Machine Learning for Resource-Constrained Computing Systems

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)
genehmigte

Dissertation

von

Martin Rapp
aus Ehingen (Donau)

Tag der mündlichen Prüfung: 31. Mai 2022
Referent: Prof. Dr.-Ing. Jörg Henkel, Karlsruher Institut für Technologie (KIT)
Korreferent: Prof. Dr. Jian-Jia Chen, Technische Universität (TU) Dortmund

Acknowledgements

First of all, I would like to express my deep gratitude to Prof. Dr.-Ing. Jörg Henkel for his continuous advice and guidance throughout my dissertation. I also greatly appreciate the freedom that he provided me in exploring and defining my own research directions and developing my own ideas. I also would like to thank Prof. Dr. Jian-Jia Chen for co-advising my dissertation and for providing valuable and constructive feedback to improve the quality of this dissertation.

I was fortunate to be a part of a supportive research environment, where many colleagues helped me with various smaller and bigger issues regarding research and teaching. My thanks go to Prof. Dr.-Ing. Anuj Pathania for helping me start my research. I would like to thank Dr.-Ing. Heba Khdr, Dr.-Ing. Lars Bauer, Dr.-Ing. Marvin Damschen, and Dr. Georgios Zervakis for always being available when I was looking for advice, for providing feedback on this dissertation, and for helping me prepare for the Ph.D. defense. I would like to thank Kilian Pfeiffer for providing comments on my dissertation.

My parents have enabled and fostered my fascination for computer science early on and have continuously supported my interests and education. For this, I am deeply grateful. My heartfelt gratitude goes to my partner Hannah for her love and support, her patience with the generally inconvenient deadlines, and for accompanying me through ups and downs – in summary, for always standing by me.

Thank You!

Martin Rapp

Abstract

Computing systems such as processors are generally constrained in their resources like power consumption, energy consumption, heat dissipation, and chip area. This makes optimizing the management of the available resources of paramount importance to achieve goals like maximum performance. In particular, system-level resource management has a major impact on the performance, power, and temperature during application execution by utilizing application mapping, application migration, and dynamic voltage and frequency scaling (DVFS).

The main challenges in resource management are coping with the high complexity of applications and platforms, coping with unseen, i.e., not known at design time, scenarios in the workload and platform configuration, achieving proactive management, and maintaining a low run-time overhead. Existing solutions based on hand-coded rules, simple heuristics, or analytical models insufficiently tackle these challenges. To this end, this dissertation focuses on employing machine learning (ML) within the resource management. ML-based solutions allow tackling the challenges by predicting the impact of potential resource management actions, by estimating hidden, i.e., unobservable at run time, properties of applications, or by directly learning a resource management policy. This dissertation presents several novel ML-based resource management techniques for different platforms, objectives, and constraints.

First, a prediction-based application migration technique for performance maximization of many-core processors with distributed shared last-level cache (LLC) under a thermal constraint is presented. It employs a neural network (NN) model to predict the impact of potential migrations on the overall system performance to determine the best one, enabling proactive management. The prediction model is trained to cope with during training unseen applications, and even copes with varying thermal constraints.

Second, a frequency boosting technique for performance maximization of homogeneous many-core processors under a thermal constraint using DVFS is presented. It is based on a novel boostability metric that integrates the sensitivities of performance, power, and temperature to voltage/frequency (V/f) changes. The sensitivities of performance and power depend on the application and cannot be observed or measured at run time. Therefore, an NN model is employed to estimate these values for unseen applications, thereby helping tackle the complexity of boosting optimization.

Third, an imitation learning (IL)-based application migration technique for temperature minimization of heterogeneous multi-core processors under quality of service (QoS) targets is introduced. It uses IL to directly learn the migration policy by learning from optimal oracle demonstrations. It employs an NN to tackle the complexity of the platform and application behavior. The NN inference is accelerated using an existing generic NN accelerator, a so-called neural processing unit (NPU).

Finally, since ML also needs to run with limited resources, a technique for resource-aware distributed on-device learning is presented to maintain a constant training throughput under fast-varying computational resource availability, e.g., due to shared resource contention. It employs structured dropout, which randomly drops parts of the NN during training. This allows to dynamically adjust the required resources for training with negligible overhead, at the cost of a slower training convergence. The Pareto-optimal per-layer dropout rates are determined using a design space exploration (DSE).

Evaluations of these techniques are performed both in simulation and on real hardware, and demonstrate significant improvements over the state of the art, at negligible run-time overhead. Ultimately, this dissertation shows that ML is a key technology to optimize system-level resource management by tackling the involved challenges listed above.

Zusammenfassung

Die verfügbaren Ressourcen in Informationsverarbeitungssystemen wie Prozessoren sind in der Regel eingeschränkt. Das umfasst z. B. die elektrische Leistungsaufnahme, den Energieverbrauch, die Wärmeabgabe oder die Chipfläche. Daher ist die Optimierung der Verwaltung der verfügbaren Ressourcen von größter Bedeutung, um Ziele wie maximale Performanz zu erreichen. Insbesondere die Ressourcenverwaltung auf der Systemebene hat über die (dynamische) Zuweisung von Anwendungen zu Prozessorkernen und über die Skalierung der Spannung und Frequenz (dynamic voltage and frequency scaling, DVFS) einen großen Einfluss auf die Performanz, die elektrische Leistung und die Temperatur während der Ausführung von Anwendungen.

Die wichtigsten Herausforderungen bei der Ressourcenverwaltung sind die hohe Komplexität von Anwendungen und Plattformen, unvorhergesehene (zur Entwurfszeit nicht bekannte) Anwendungen oder Plattformkonfigurationen, proaktive Optimierung und die Minimierung des Laufzeit-Overheads. Bestehende Techniken, die auf einfachen Heuristiken oder analytischen Modellen basieren, gehen diese Herausforderungen nur unzureichend an. Aus diesem Grund ist der Hauptbeitrag dieser Dissertation der Einsatz maschinellen Lernens (ML) für Ressourcenverwaltung. ML-basierte Lösungen ermöglichen die Bewältigung dieser Herausforderungen durch die Vorhersage der Auswirkungen potenzieller Entscheidungen in der Ressourcenverwaltung, durch Schätzung verborgener (unbeobachtbarer) Eigenschaften von Anwendungen oder durch direktes Lernen einer Ressourcenverwaltungs-Strategie. Diese Dissertation entwickelt mehrere neuartige ML-basierte Ressourcenverwaltung-Techniken für verschiedene Plattformen, Ziele und Randbedingungen.

Zunächst wird eine auf Vorhersagen basierende Technik zur Maximierung der Performanz von Mehrkernprozessoren mit verteiltem Last-Level Cache und limitierter Maximaltemperatur vorgestellt. Diese verwendet ein neuronales Netzwerk (NN) zur Vorhersage der Auswirkungen potenzieller Migrationen von Anwendungen zwischen Prozessorkernen auf die Performanz. Diese Vorhersagen erlauben die Bestimmung der bestmöglichen Migration und ermöglichen eine proaktive Verwaltung. Das NN ist so trainiert, dass es mit unbekannten Anwendungen und verschiedenen Temperaturlimits zurechtkommt.

Zweitens wird ein Boosting-Verfahren zur Maximierung der Performanz homogener Mehrkernprozessoren mit limitierter Maximaltemperatur mithilfe von DVFS vorgestellt. Dieses basiert auf einer neuartigen Boostability-Metrik, die die Abhängigkeiten von Performanz, elektrischer Leistung und Temperatur auf Spannungs/Frequenz-Änderungen in einer Metrik vereint. Die Abhängigkeiten von Performanz und elektrischer Leistung hängen von der Anwendung ab und können zur Laufzeit nicht direkt beobachtet (gemessen) werden. Daher wird ein NN verwendet, um diese Werte für unbekannte Anwendungen zu schätzen und so die Komplexität der Boosting-Optimierung zu bewältigen.

Drittens wird eine Technik zur Temperaturminimierung von heterogenen Mehrkernprozessoren mit Quality of Service-Zielen vorgestellt. Diese verwendet Imitationslernen, um eine Migrationsstrategie von Anwendungen aus optimalen Orakel-Demonstrationen zu lernen. Dafür wird ein NN eingesetzt, um die Komplexität der Plattform und des Anwendungsverhaltens zu bewältigen. Die Inferenz des NNs wird mit Hilfe eines vorhandenen generischen Beschleunigers, einer Neural Processing Unit (NPU), beschleunigt.

Auch die ML Algorithmen selbst müssen auch mit begrenzten Ressourcen ausgeführt werden. Zuletzt wird eine Technik für ressourcenorientiertes Training auf verteilten Geräten vorgestellt, um einen konstanten Trainingsdurchsatz bei sich schnell ändernder Verfügbarkeit von Rechenressourcen aufrechtzuerhalten, wie es z. B. aufgrund von Konflikten bei gemeinsam genutzten Ressourcen der Fall ist. Diese Technik verwendet Structured Dropout, welches beim Training zufällige Teile des NNs auslässt. Dadurch können die erforderlichen Ressourcen für das Training dynamisch angepasst werden – mit vernachlässigbarem Overhead, aber auf Kosten einer langsameren Trainingskonvergenz. Die Pareto-optimalen Dropout-Parameter pro Schicht des NNs werden durch eine Design Space Exploration bestimmt.

Evaluierungen dieser Techniken werden sowohl in Simulationen als auch auf realer Hardware durchgeführt und zeigen signifikante Verbesserungen gegenüber dem Stand der Technik, bei vernachlässigbarem Laufzeit-Overhead. Zusammenfassend zeigt diese Dissertation, dass ML eine Schlüsseltechnologie zur Optimierung der Verwaltung der limitierten Ressourcen auf Systemebene ist, indem die damit verbundenen Herausforderungen angegangen werden.

List of Publications

The following list enumerates papers and book chapters published by the author of this dissertation while pursuing his doctorate.

First-author publications that present major contributions to this dissertation

[1] Martin Rapp, Heba Khdr, Nikita Krohmer, and Jörg Henkel. "NPU-Accelerated Imitation Learning for Thermal Optimization of QoS-Constrained Heterogeneous Multi-Cores". In: *arXiv preprint arXiv:2206.05459* (2022).

[2] Martin Rapp, Nikita Krohmer, Heba Khdr, and Jörg Henkel. "NPU-Accelerated Imitation Learning for Thermal- and QoS-Aware Optimization of Heterogeneous Multi-Cores". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022. DOI: `10.23919/DATE54114.2022.9774681`.

[3] Martin Rapp, Ramin Khalili, Kilian Pfeiffer, and Jörg Henkel. "DISTREAL: Distributed Resource-Aware Learning in Heterogeneous Systems". In: *AAAI Conference on Artificial Intelligence (AAAI)*. 2022.

[4] Martin Rapp, Hussam Amrouch, Yibo Lin, Bei Yu, David Pan, Marilyn Wolf, and Jörg Henkel. "MLCAD: A Survey of Research in Machine Learning for CAD (Keynote Paper)". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021). DOI: `10.1109/TCAD.2021.3124762`.

[5] Martin Rapp, Mohammed Bakr Sikal, Heba Khdr, and Jörg Henkel. "SmartBoost: Lightweight ML-Driven Boosting for Thermally-Constrained Many-Core Processors". In: *Design Automation Conference (DAC)*. 2021. DOI: `10.1109/DAC18074.2021.9586287`.

[6] Martin Rapp, Anuj Pathania, Tulika Mitra, and Jörg Henkel. "Neural Network-based Performance Prediction for Task Migration on S-NUCA Many-Cores". In: *IEEE Transactions on Computers (TC)* 70.10 (2021). DOI: `10.1109/TC.2020.3023022`.

[7] Martin Rapp, Mark Sagi, Anuj Pathania, Andreas Herkersdorf, and Jörg Henkel. "Power-and Cache-Aware Task Mapping with Dynamic Power Budgeting for Many-Cores". In: *IEEE Transactions on Computers (TC)* 69.1 (2020). DOI: `10.1109/TC.2019.2935446`.

[8] Martin Rapp, Anuj Pathania, Tulika Mitra, and Jörg Henkel. "Prediction-Based Task Migration on S-NUCA Many-Cores". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019. DOI: `10.23919/DATE.2019.8714974`.

[9] Martin Rapp, Anuj Pathania, and Jörg Henkel. "Pareto-Optimal Power- and Cache-Aware Task Mapping for Many-Cores with Distributed Shared Last-Level Cache". In: *Int. Symp. on Low Power Electronics and Design (ISLPED)*. ACM/IEEE. 2018. DOI: `10.1145/3218603.3218630`.

First-author publications that present minor contributions to this dissertation

[10] Martin Rapp, Omar Elfatairy, Marilyn C. Wolf, Jörg Henkel, and Hussam Amrouch. "Towards NN-based Online Estimation of the Full-Chip Temperature and the Rate of Temperature Change". In: *Workshop on Machine Learning for CAD (MLCAD)*. 2020, pp. 95–100. DOI: `10.1145/3380446.3430648`.

[11] Martin Rapp, Ramin Khalili, and Jörg Henkel. "Distributed Learning on Heterogeneous Resource-Constrained Devices". In: *arXiv preprint arXiv:2006.05403* (2020).

[12] Martin Rapp, Sami Salamin, Hussam Amrouch, Girish Pahwa, Yogesh Chauhan, and Jörg Henkel. "Performance, Power and Cooling Trade-Offs with NCFET-based Many-Cores". In: *Design Automation Conference (DAC)*. 2019. DOI: `10.1145/3316781.3317880`.

[13] Martin Rapp, Hussam Amrouch, Marilyn C. Wolf, and Jörg Henkel. "Machine Learning Techniques to Support Many-Core Resource Management: Challenges and Opportunities". In: *Workshop on Machine Learning for CAD (MLCAD)*. ACM/IEEE. 2019. DOI: `10.1109/MLCAD48534.2019.9142064`.

Other co-authored publications

[14] Lokesh Siddhu, Rajesh Kedia, Shailja Pandey, Martin Rapp, Anuj Pathania, Jörg Henkel, and Preeti Ranjan Panda. "CoMeT: An Integrated Interval Thermal Simulation Toolchain for 2D, 2.5 D, and 3D Processor-Memory Systems". In: *ACM Transactions on Architecture and Code Optimization (TACO)* (2022).

[15] Marcel Mettler, Martin Rapp, Heba Khdr, Daniel Müller-Gritschneder, and Jörg Henkel. "An FPGA-based Approach to Evaluate Thermal and Resource Management Strategies of Many-Core Processors". In: *ACM Transactions on Architecture and Code Optimization (TACO)* (2022).

[16] Mohammed Bakr Sikal, Heba Khdr, Martin Rapp, and Jörg Henkel. "Thermal- and Cache-Aware Resource Management based on ML-Driven Cache Contention Prediction". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022.

[17] Kilian Pfeiffer, Martin Rapp, Ramin Khalili, and Jörg Henkel. "CoCo-FL: Communication- and Computation-Aware Federated Learning via Partial NN Freezing and Quantization". In: *arXiv preprint arXiv:2203.05468* (2022).

[18] Veera Venkata Ram Murali Krishna Rao Muvva, Martin Rapp, Jörg Henkel, Hussam Amrouch, and Marilyn C. Wolf. "On the Effectiveness of Quantization and Pruning on the Performance of FPGAs-based NN Temperature Estimation". In: *Workshop on Machine Learning for CAD (MLCAD)*. 2021.

[19] Mark Sagi, Martin Rapp, Heba Khdr, Yizhe Zhang, Nael Fasfous, Nguyen Anh Vu Doan, Thomas Wild, Jörg Henkel, and Andreas Herkersdorf. "Long Short-Term Memory Neural Network-based Power Forecasting of Multi-Core Processors". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2021, pp. 1685–1690.

[20] Sami Salamin, Victor M Van Santen, Martin Rapp, Jörg Henkel, and Hussam Amrouch. "Minimizing Excess Timing Guard Banding Under Transistor Self-Heating Through Biasing at Zero-Temperature Coefficient". In: *IEEE Access* 9 (2021), pp. 30687–30697.

[21] Hussam Amrouch, Martin Rapp, Sami Salamin, and Jörg Henkel. "Impact of Negative Capacitance Field-Effect Transistor (NCFET) on Many-Core Systems". In: *A Journey of Embedded and Cyber-Physical Systems.* Springer, 2021, pp. 107–123.

[22] Mark Sagi, Nguyen Anh Vu Doan, Martin Rapp, Thomas Wild, Jörg Henkel, and Andreas Herkersdorf. "A Lightweight Nonlinear Methodology to Accurately Model Multi-Core Processor Power". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39.11 (2020), pp. 3152–3164.

[23] Sami Salamin, Martin Rapp, Jörg Henkel, Andreas Gerstlauer, and Hussam Amrouch. "Dynamic Power and Energy Management for NCFET-Based Processors". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39.11 (2020), pp. 3361–3372.

[24] Behnaz Pourmohseni, Michael Glaß, Jörg Henkel, Heba Khdr, Martin Rapp, Valentina Richthammer, Tobias Schwarzer, Fedor Smirnov, Jan Spieck, Jürgen Teich, Andreas Weichslgartner, and Stefan Wildermann. "Hybrid Application Mapping for Composable Many-Core Systems: Overview and Future Perspective". In: *Journal of Low Power Electronics and Applications (JLPEA)* 10.4 (2020).

[25] Sami Salamin, Martin Rapp, Anuj Pathania, Arka Maity, Jörg Henkel, Tulika Mitra, and Hussam Amrouch. "Power-Efficient Heterogeneous Many-Core Design with NCFET Technology". In: *IEEE Transactions on Computers (TC)* 70.9 (2020), pp. 1484–1497.

[26] Sami Salamin, Martin Rapp, Hussam Amrouch, Andreas Gerstlauer, and Jörg Henkel. "Energy Optimization in NCFET-based Processors". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2020, pp. 630–633.

[27] Marvin Damschen, Martin Rapp, Lars Bauer, and Jörg Henkel. "i-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems". In: *Embedded, Cyber-Physical, and IoT Systems.* Springer, 2020.

[28] Jörg Henkel, Hussam Amrouch, Martin Rapp, Sami Salamin, Dayane Reis, Di Gao, Xunzhao Yin, Michael Niemier, Cheng Zhuo, X Sharon Hu, Hsiang-Yun Cheng, and Chia-Lin Yang. "The Impact of Emerging Technologies on Architectures and System-level Management". In: *International Conference on Computer-Aided Design (ICCAD).* IEEE. 2019.

[29] Sami Salamin, Martin Rapp, Hussam Amrouch, Girish Pahwa, Yogesh Chauhan, and Jörg Henkel. “NCFET-aware Voltage Scaling”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE. 2019.

[30] Jörg Henkel, Heba Khdr, and Martin Rapp. “Smart Thermal Management for Heterogeneous Multicores”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 132–137.

Research at the Chair for Embedded Systems

The Chair for Embedded Systems (CES) at the Karlsruhe Institute of Technology (KIT) has its core expertise in design and architectures for embedded systems. The initial focus of the CES was on hardware/software co-design. This research area studies design methodologies that simultaneously consider hardware and software to optimize for a design metric such as minimizing power under constraints such as area or performance. One method to achieve this is to employ application-specific instruction-set processors (ASIPs), which are designed and manufactured specifically for the executed application [C1]. More flexibility is attained by introducing run-time reconfiguration to dynamically adapt the instruction set to the application requirements [C2].

Research related to multi-/many-core processors has also been a continual focus of the CES, reacting to predictions about a persistent increase in the number of CPU cores per processor. This includes research on networks-on-chip (NoCs) [C3], which were proposed as a scalable alternative to the common bus-based communication in processors. The CES studied this both at the architectural level, e.g., by developing a configurable NoC to adapt to changing application requirements [C4], and at the system level, e.g., by developing NoC-aware scheduling to minimize the traffic [C5]. Another branch of research within the scope of multi-/many-core processors addresses power management, which is necessary to cope with increasing power densities and corresponding elevated temperatures in modern technology nodes due to the failure of Dennard's scaling. For instance, distributed agent-based management techniques have been studied that distribute the global power budget to the individual cores based on the application requirements [C6]. Within this scope, an early learning-based technique has been presented, which employs economic learning to let per-core agents "buy" power budget [C7]. Modern technology nodes also aggravate degradation effects. The CES has developed several techniques to cope with such degradation effects to increase the reliability of computing, most prominently in the context of DFG SPP 1500 "Dependable Embedded Systems" [C8]. As the underlying effects affect individual transistors, this research spans several levels of abstraction from the device (transistor) level to the system level.

Approximate computing studies techniques to optimize power, area, or performance by tolerating certain errors in computations. The CES contributed to this research area by designing approximate functional units like adders [C9] and by developing embedded design automation (EDA) methodologies to design accelerators comprising many approximate functional units [C10].

Recently, the CES has performed research on the internet of things (IoT) and embedded ML. IoT systems perform computations at several hierarchy levels from the distributed devices that interact with their environment, to cloud servers that offer virtually unlimited computational resources but have high communication latency, and at edge devices at the boundary between the two. IoT systems require resource management in the form of computational offloading [C11] to decide which computation is performed at which level in the hierarchy. In particular, IoT devices that employ ML models to process sensor data, have been studied. Several techniques in the scope of embedded ML have been developed to support the computationally heavy operations of ML in IoT systems [C12]. This research direction is currently being extended towards ML training on embedded devices, such as in federated learning (FL) systems.

DFG Transregio TR89 – Invasive Computing The central idea of DFG Transregio TR89 "Invasive Computing" (*InvasIC*) is to enable resource-aware programming, where the resource requirements of an application are expressed by the programmer. This involves research at all levels of the computing stack, at application and compiler level, where such requirements must be expressed and represented in a systematic way, at the operating system (OS)-level, where the resources are managed, and even at the hardware level, where fine-grained resource allocation to applications needs to be supported. This project is a collaborative project between the three German universities Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Karlsruher Institut für Technologie (KIT), and Technische Universität München (TUM). The project started in 2010 and is currently in its third funding phase from July 2018 to June 2022.

Within *InvasIC*, sub-project B3 studies power and thermal management of invasive multi-core processors [C13]. The challenge thereby is that power and temperature are affected jointly by all running applications, requiring global optimization. Several techniques have been developed to this end. For instance, the work in [C14] jointly selects the degree of parallelism, mapping, and V/f levels of applications under a thermal constraint by exploiting design-time knowledge of the performance and power of applications at various configurations. Another outcome of this research is Thermal Safe Power (TSP) [C15], which translates a global thermal constraint into per-core power budgets depending on the number and location of active cores, which can be enforced decentrally. TSP has been extended to consider heterogeneous processors by formulating a power density constraint and reallocate surplus power density to other cores [C16].

This dissertation contributes to sub-project B3. The main goal of B3 in the third funding phase is to reduce the dependency of resource management on design-time hardware and application models. This includes unknown workloads, i.e., applications, and to a limited degree also unknown hardware parameters, such as the cooling. The techniques developed in this dissertation contribute to this goal by being designed to cope with unknown scenarios at run time. This is achieved by employing ML modeling that generalizes to different scenarios than has been used in the training.

[C1] Jörg Henkel. “A Low Power Hardware/Software Partitioning Approach for Core-Based Embedded Systems”. In: *Design Automation Conference (DAC)*. IEEE. 1999.

[C2] Lars Bauer, Muhammad Shafique, Simon Kramer, and Jörg Henkel. “RISPP: Rotating Instruction Set Processing Platform”. In: *Design Automation Conference (DAC)*. 2007.

[C3] Jörg Henkel, Wayne Wolf, and Srimat Chakradhar. “On-Chip Networks: A Scalable, Communication-Centric Embedded System Design Paradigm”. In: *International Conference on VLSI Design*. IEEE. 2004.

[C4] Mohammad Abdullah Al Faruque, Thomas Ebi, and Jörg Henkel. “Run-Time Adaptive On-Chip Communication Scheme”. In: *International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2007.

[C5] Mohammad Abdullah Al Faruque, Rudolf Krist, and Jörg Henkel. “ADAM: Run-Time Agent-Based Distributed Application Mapping for On-Chip Communication”. In: *Design Automation Conference (DAC)*. IEEE. 2008.

[C6] Thomas Ebi, Mohammad Abdullah Al Faruque, and Jörg Henkel. “TAPE: Thermal-Aware Agent-Based Power Econom Multi/Many-Core Architectures”. In: *International Conference on Computer-Aided Design-Digest of Technical Papers*. IEEE. 2009.

[C7] Thomas Ebi, David Kramer, Wolfgang Karl, and Jörg Henkel. “Economic Learning for Thermal-Aware Power Budgeting in Many-Core Architectures”. In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. 2011.

[C8] Jörg Henkel, Lars Bauer, Joachim Becker, Oliver Bringmann, Uwe Brinkschulte, Samarjit Chakraborty, Michael Engel, Rolf Ernst, Hermann Härtig, Lars Hedrich, et al. “Design and Architectures for Dependable Embedded Systems”. In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2011.

[C9] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. “A Low Latency Generic Accuracy Configurable Adder”. In: *Design Automation Conference (DAC)*. IEEE. 2015.

[C10] Jorge Castro-Godínez, Sven Esser, Muhammad Shafique, Santiago Pagani, and Jörg Henkel. “Compiler-Driven Error Analysis for Designing Approximate Accelerators”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018.

[C11] Farzad Samie, Vasileios Tsoutsouras, Lars Bauer, Sotirios Xydis, Dimitrios Soudris, and Jörg Henkel. “Computation Offloading and Resource Allocation for Low-Power IoT Edge Devices”. In: *World Forum on Internet of Things (WF-IoT)*. IEEE. 2016.

[C12] Farzad Samie, Sebastian Paul, Lars Bauer, and Jörg Henkel. “Highly Efficient and Accurate Seizure Prediction on Constrained IoT Devices”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018.

[C13] Jörg Henkel, Heba Khdr, Santiago Pagani, and Muhammad Shafique. “New Trends in Dark Silicon”. In: *Design Automation Conference (DAC)*. IEEE. 2015.

[C14] Heba Khdr, Santiago Pagani, Muhammad Shafique, and Jörg Henkel. “Thermal Constrained Resource Management for Mixed ILP-TLP Workloads in Dark Silicon Chips”. In: *Design Automation Conference (DAC)*. ACM. 2015.

[C15] Santiago Pagani, Heba Khdr, Jian-Jia Chen, Muhammad Shafique, Minming Li, and Jörg Henkel. “Thermal Safe Power (TSP): Efficient Power Budgeting for Heterogeneous Manycore Systems in Dark Silicon”. In: *IEEE Transactions on Computers (TC)* 66.1 (2017), pp. 147–162. DOI: 10.1109/TC.2016.2564969.

[C16] Heba Khdr, Santiago Pagani, Ericles Sousa, Vahid Lari, Anuj Pathania, Frank Hannig, Muhammad Shafique, Jürgen Teich, and Jörg Henkel. “Power Density-Aware Resource Management for Heterogeneous Tiled Multicores”. In: *IEEE Transactions on Computers (TC)* 66.3 (2016), pp. 488–501.

Contents

1 Introduction

Computing systems like processors are generally resource-constrained [31, 32, 33, 34]. For instance, their power consumption, energy consumption, heat dissipation, and chip area are often limited. This makes optimizing the management of the available resources of paramount importance to achieve goals like maximum performance or minimum temperature. Optimizing the resource usage can be tackled at several stages of the design and operation of processors. This includes the hardware design, which optimizes the implementation of processors in hardware w.r.t. performance, power, area, etc., at different abstraction levels such as technology, circuits, microarchitecture, etc. At run time (operation time), *system-level resource management*, which is the focus of this dissertation, manages the available hardware resources in software at the operating system (OS) level by dynamically assigning the resources to applications. Resource management utilizes dynamic voltage and frequency scaling (DVFS) and application mapping/migration, and thereby has a major impact on the achievable performance, dissipated power, reached temperature, and consumed energy. The classical approaches for resource management either perform optimization at design time, rely on hand-coded rules or simple heuristics, or use analytical models of applications and the platform. They can not sufficiently tackle the challenges in resource management of dynamic workloads, as will be discussed further in Section 1.2. Novel approaches to resource management are required instead. To this end, this dissertation investigates machine learning (ML)-based optimization.

ML has demonstrated great success in various domains [35] like language processing [36], robotics [37], etc. ML aims at automatically learning abstract concepts from a typically large set of demonstrations. This is beneficial because it is easier in these domains to (manually) provide the target output for individual inputs than explicitly create rules or algorithms that work with arbitrary inputs. For instance, in image classification [38], which is a flagship use case of ML, it is much easier to annotate a large set of images with information about their contents than to explicitly formulate rules or algorithms that determine the content of an image based on its pixel values. This applies also to many problems in resource management, where it is for instance easier to determine the optimal application migration in a given single scenario than to create algorithms or rules to determine the optimal management in all possible scenarios. Recently, researchers have started to apply ML also to various problems outside the typical flagship use cases, and even in the embedded design automation (EDA) context, such as chip design [39].

The focus of this dissertation is to study how ML can help in optimizing resource-constrained computing systems. The majority of the dissertation studies methods on how to embed ML into system-level resource management. Finally, since ML algorithms themselves also need to run on the limited resources, this dissertation studies techniques on how to perform ML with limited computational resources.

1.1 Constrained Resources in Computing Systems

There are several constrained resources in computing systems. First, the *chip area* is limited to reduce production costs. This limits the amount of memory for caches and the number of central processing unit (CPU) cores that can be placed on the chip. Assigning the chip area to these architectural components is done during hardware design and can not be changed at run time. Therefore, run-time resource management can only manage the area by assigning cores to applications, which are consequently considered as the limited resource at the system level. In the scope of many-core processors, mostly a one-thread-per-core model [40] is employed, such that every core executes zero or one thread. Cores are managed by application mapping, which assigns cores to applications at their start time, and application migration, which changes the assignment of cores to applications during their execution. Cores are not necessarily homogeneous. They may differ in their microarchitecture, e.g., simple in-order cores and complex out-of-order cores [41, 42], or in their cache latency [43]. Consequently, there is in general a limited number of cores with different characteristics that need to be managed.

In addition, *peak power* and *energy* are commonly limited [32]. The power supply is designed to deliver a certain peak power that must not be exceeded. Energy is limited for instance in battery-driven operation. These limits may even change over time, e.g., in the case of a device that is powered by renewable energies. The main method for system-level resource management to distribute the available power and energy to applications is DVFS, which dynamically at run time adjusts the voltage/frequency (V/f) levels of cores to adjust the trade-off between performance and power. In addition, heterogeneous cores also offer different power and performance to applications, which allows distributing power and energy to applications by application mapping and migration.

Finally, the *heat dissipation* of a processor is limited by the cooling system [44]. Since the failure of Dennard scaling [45], voltage scaling could not keep up with technology scaling, resulting in increasing power density. High power density increases the on-chip temperature, which accelerates circuit degradation and may even permanently damage the chip [33, 34]. These negative effects can be reduced by enforcing an upper limit on the temperature. Resource management affects the temperature via DVFS and application mapping/migration. DVFS changes the temperature via the power consumption. Application migration can migrate threads from hot to colder cores. Application mapping can leave cores idle between active cores with high power consumption to dissipate the heat.

1.2 Challenges in Resource Management

Resource management mostly operates on discrete decisions, such as assigning CPU cores or selecting discrete V/f levels. It is *NP*-complete in the general case [46]. In addition, the

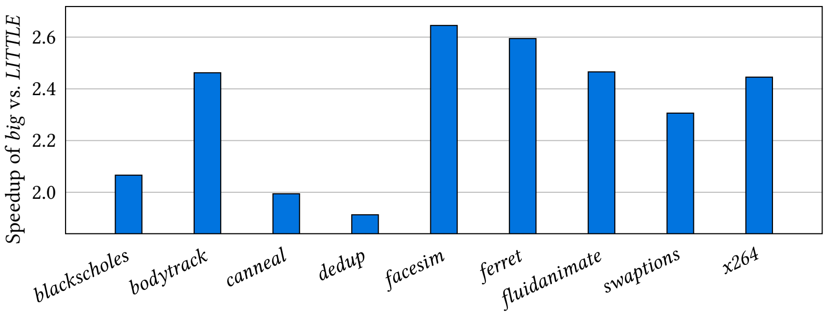


Figure 1.1: Performance of the *big* cluster vs. *LITTLE* cluster on Arm big.LITTLE for various applications.

search spaces are large. For instance, when running 16 threads on a many-core processor with 64 cores, where each core supports 31 V/f levels, there are $\binom{64}{16} \approx 4.9{\cdot}10^{14}$ possible mappings with $31^{16} \approx 7.3{\cdot}10^{23}$ possible V/f level settings, each, resulting in a total of over 10^{38} possible configurations. Therefore, it is most often infeasible to find optimal solutions and the resource management should find solutions as close as possible to the optimum instead. There are four main challenges to resource management.

Challenge: Platform and Application Complexity Many factors affect the power consumption (and thereby the temperature) and performance of an application. These include the application characteristics, which are determined by the instruction mix and inter-instruction dependencies, the CPU microarchitecture, the cache architecture, the interconnect, the random access memory (RAM), etc. Fig. 1.1 presents an example to demonstrate the role of the application characteristics and CPU/memory microarchitecture in the application performance. The figure shows that the speedup of different *PARSEC* [47] applications when running on the *big* cluster compared to running on the *LITTLE* cluster on an Arm big.LITTLE processor[1] varies significantly. First, the speedup comes from a different microarchitecture and cache architecture of the two clusters. The *big* cluster uses out-of-order processors, in contrast to the in-order processors of the *LITTLE* cluster. In addition, the *big* cluster supports higher V/f levels and has larger caches. Secondly, different applications show different speedup, as they benefit differently from the features of the *big* cluster, e.g., due to different memory usage patterns that affect the efficacy of the caches, or different instruction-level parallelism that affects the efficacy of out-of-order execution.

The platform complexity is aggravated by technology scaling, which allows to build more complex microarchitectures and house more cores in a processor due to tighter integration. Explicitly modeling every detail of modern many-core processor architectures is not practicable for resource management due to their sheer complexity. First, it requires

[1] The experimental setup of the heterogeneous Arm big.LITTLE multi-core processor is described in Section 3.2.

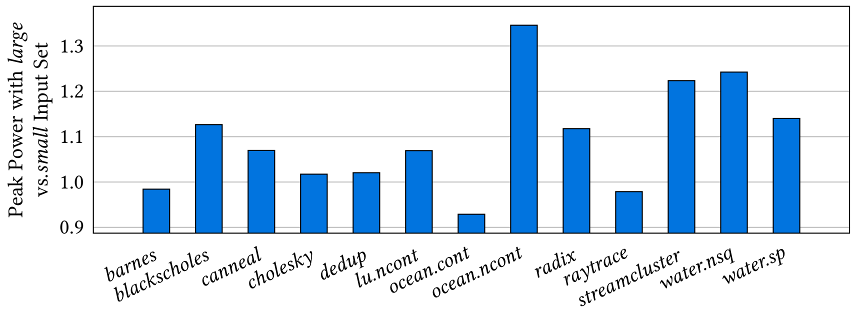


Figure 1.2: Peak power of different *PARSEC* and *SPLASH-2* applications on a homogeneous many-core processor with the *simmedium*/*large* input data compared to the *simsmall*/*small* input data.

detailed knowledge of the architecture, which may be kept confidential by manufacturers. Second, such accurate models would be too slow for run-time use, limiting their use to design-time simulations. When introducing simplifications in the form of abstractions, it is difficult to still maintain a relatively high accuracy, requiring a detailed understanding of all involved aspects to avoid introducing too large errors.

In summary, a resource management technique needs to optimize a complex platform running complex applications, without requiring detailed knowledge about their internal structure.

Challenge: Unseen Scenarios The applications that are executed at run time are in general not known at design time. Such systems are called open systems [48], where a priori unknown applications arrive at a priori unknown times. However, as already shown in Fig. 1.1, there is a large variation between applications. Furthermore, even the characteristics of the same application binary differ when processing different input data. This is illustrated in the example in Fig. 1.2, which plots the peak power consumption of several *PARSEC* and *SPLASH-2* [49] applications when operating on the *simmedium*/*large* input data relative to the *simsmall*/*small* input data, running on a homogeneous many-core processor[2]. For instance, the power consumption of *ocean.ncont* increases by 35 %, while the power consumption of *ocean.cont* decreases by 7 %, just by changing the input data, even though the same binary is executed. In addition, several applications running in parallel further affect each other's execution via thermal coupling of the cores they are running on and contention on the shared resources like memory.

The set of currently running applications is called the workload. Overall, there are infinitely many possible workloads that may be executed on the platform, i.e., it is impossible to explicitly consider all possible workloads at design time when developing a resource management technique. Therefore, the workload that is executed at run time must be considered as *unseen*, i.e., not known, at design time.

[2] The experimental setup of the homogeneous many-core processor is described in Section 3.1.1.

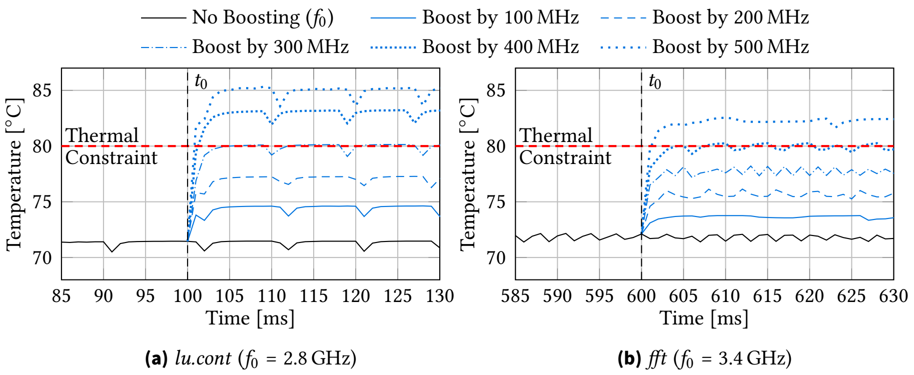


Figure 1.3: Without proactively considering the impact of upscaling (boosting) V/f levels, violations of the thermal constraint cannot be avoided.

In summary, a resource management technique needs to cope with unseen applications and workloads that have not been observed at design time. In addition, the characteristics of the platform may not be known at design time. For instance, the cooling characteristics may degrade over time, or change with the environment temperature.

Challenge: Proactive Management Resource management typically needs to satisfy constraints, such as a thermal or performance constraint. It is in general not clear in advance whether executing a certain action (e.g., scaling V/f levels) would violate the constraint. Fig. 1.3 shows an example, in which the *SPLASH-2 lu.cont* (a) or *fft* (b) application is running on the same homogeneous many-core processor as in Fig. 1.2, subject to a thermal constraint of 80 °C. Initially, the applications are operated at f_0 = 2.8 GHz and 3.4 GHz, respectively. Both applications show a thermal margin of around 8 °C. At t_0 = 100 ms and 600 ms, respectively, several V/f level boosts between 100 MHz and 500 MHz are considered to increase the performance. Without prior knowledge, it is not clear which of the V/f levels are thermally safe. In fact, the thermally-safe V/f level increase differs between the two applications, despite almost the same initial thermal margin. Switching to an unsafe level, e.g., boost *lu.cont* by 500 MHz, results in a thermal violation after less than 1 ms, which is too fast to react, and must be avoided. Increasing the V/f levels step by step until the temperature is exceeded is slow, especially if the thermal margin is large, and still results in a thermal violation when testing the first unsafe level. This problem is aggravated by the fact that typically many applications are executing in parallel, where each application has separate characteristics, requiring to operate each application at a different optimal V/f level, but all jointly affect the peak temperature. Similar challenges are observed with application migration, where different cores offer different properties (microarchitecture, cache latency, thermal properties, etc.). Trying all possible migrations to find the best is impracticable due to their potentially large number, and migrations that cause abrupt performance or temperature violations must be avoided.

In summary, proactive resource management needs to consider the impact of an action before executing it. However, this requires predictions, which are difficult to obtain, and there is usually a large number of potential actions that need to be considered.

Challenge: Run-Time Overhead The goal of resource management is to optimize the resource usage by the applications. Since the state of platform and applications changes continuously over time, optimal management actions also change continuously. Therefore, resource management is usually invoked periodically to be able to adapt to such changes. The optimization performed by the resource management itself also requires computational resources at run time, called the run-time overhead. In particular, one or several cores are occupied for some time by resource management optimization, and the optimization may itself dissipate power, consume energy, and heat up the cores. All these resources are not available for the applications.

In particular, there is a trade-off between the achievable quality of the management decisions and the overhead. First, achieving faster adaptability to changes requires a faster control epoch, i.e., executing the optimization more often. Second, obtaining management actions closer to the optimum requires a more complex optimization, which requires more resources. In both cases, the more resources are spent on the optimization, the better is the management but fewer resources are available to the applications.

In summary, resource management needs to optimize the usage of time-varying resources by the applications while maintaining a low overhead.

1.3 Machine Learning for Resource Management

This section discusses how ML helps solve the challenges described above. This dissertation considers mostly neural network (NN) models, which recently dominate the field of ML due to their capabilities to learn both low-level features in their first layers and high-level representations in subsequent layers, coining the term *deep learning* [50].

Managing the Platform and Application Complexity ML models help managing the high complexity in the platform and applications because 1) ML can train models that learn complex functions, and 2) ML reduces the requirement for detailed system knowledge, as discussed in the following.

The capabilities of ML models, especially NN models, to learn complex functions is one of the main reasons for their recent success. Therefore, such ML models should also be capable of learning the complex interdependencies of performance, power, temperature, application characteristics, microarchitecture, etc.

As discussed earlier, one of the main challenges with high complexity is that building analytical models requires detailed knowledge of the potentially confidential internals of the platform. ML helps because it allows building models with significantly less domain knowledge. While it is still required to identify the relevant parameters to use as features, detailed knowledge about their impact is not required. Instead, information about these internals is extracted from the system by observing its behavior at different scenarios to create training data, essentially treating it as a black box. Thereby, ML models allow to trade the requirement for detailed domain knowledge for a requirement to probe the system at different scenarios.

Coping with Unseen Scenarios ML algorithms are designed to generalize to unseen data. There are 1) established processes to measure the generalization and 2) established training methods to improve generalization, as discussed in the following. When employing ML for resource management, we can utilize these established processes and methods to cope with unseen workloads or even different hardware platforms.

A key concept in ML is the separation of training and test data, where the former is used to create a model and the latter is used to evaluate the model [35]. This separation is the basis for an established process to evaluate the generalization capabilities of a trained model. The key challenge is to maintain statistical independence between training and test data. The major unseen component considered in this dissertation is the workload, i.e., the executed applications. Therefore, in the scope of this dissertation, the split of training and test data is always performed at the granularity of applications, where some applications are used exclusively for training and others are used exclusively for testing.

ML training aims at preventing overfitting of the model, which manifests itself in high accuracy on the training data, yet poor generalization (low accuracy on the test data). Several methods are typically used during NN training to improve generalization, such as dropout [51] or regularization [50]. Both techniques affect only the NN training process and have no overhead at run time.

Achieving Proactive Management The main challenge with proactive management is that the impact of an action is difficult to estimate in advance. However, in hindsight, after having executed a certain action, the impact is very straightforward to quantify and can easily be measured. This allows to create training data to train an ML model. In addition, NNs have proven effective in many control tasks, such as playing video games [52], where the impact of an action needs to be learned (implicitly or explicitly). Finally, the impact of a set of actions often follows a continuous trend. For instance, the performance and power continuously increase with higher V/f levels. This allows to use the interpolation capabilities of NNs, which learn continuous functions on continuous feature and output spaces, and thereby reduces the amount of required training data.

Maintaining a Low Run-Time Overhead While ML algorithms may be very compute-intensive, there are several properties of ML algorithms that also help achieve a low run-time overhead. These are 1) a flexible trade-off between overhead and accuracy, 2) moving the most expensive computations to the design time, and 3) easy use of hardware acceleration, as discussed in the following.

ML models allow us to make a fine-grained trade-off between overhead and accuracy. This applies both to training (creating the model) and inference (using the model). In particular, the regular structure of NNs allows to statically scale the size (depth: number of layers, and width: neurons or filters per layer) of the models for different trade-offs between overhead and accuracy. Moreover, the regular structure of computations during training allows to dynamically skip some computations to adjust to a dynamic resource availability but at the cost of slower convergence.

The separation into distinct training and inference phases allows to further reduce the overhead. Since the main concern is the run-time overhead, more complex computations can be afforded at design time. Inference needs to be done at run time, while training can also be shifted to design time. For most ML algorithms, the training is more complex than inference. Performing training at design time enables to reduce the run-time overhead significantly.

Finally, the computations involved in NN inference are dominated by multiply-accumulate (MAC) operations with a high degree of data parallelism. These computations can be implemented efficiently in hardware. In addition, these computations are always similar for different NNs, independently of the task the NN was trained for. This enables to employ a single generic NN accelerator to speed up many different policies with different NNs, instead of requiring a separate specific accelerator for each policy, which would require manufacturing a new chip.

1.4 Patterns how to Apply ML to Resource Management

This section discusses the main patterns how ML can be employed in resource management. In abstract terms, resource management can be modeled as the problem of taking actions (e.g., V/f scaling, application migration, etc.) based on observations of the state of platform and applications (e.g., temperature, hardware performance counters, cache miss rate, application arrival rate, etc.) in order to optimize for a certain objective (e.g., minimize temperature, maximize performance) under constraints (e.g., not violate a critical temperature, maintain quality of service (QoS) targets). This dissertation distinguishes and studies three alternative methods on how to employ ML in resource management that are described in the following sections, respectively.

1.4.1 Predict Impact of Resource Management Actions

The first pattern is to predict the impact of potential management actions before executing any. The impact is quantified in terms of performance, power, temperature, etc., depending on the objective and constraints. Based on the predicted impact, only one action is selected and executed. This pattern is depicted in Fig. 1.4. First, a set of candidate actions is created. Representations of these actions are passed to an ML model. The ML model predicts how each candidate action would impact the performance, power, or temperature of the platform or application. Based on these predictions, a greedy algorithm selects the best action and executes it. As the output of the model is a physical property, the ML model is a regression model that can be trained using supervised learning. Many actions needs to be observed or emulated to evaluate their impact to create the training data.

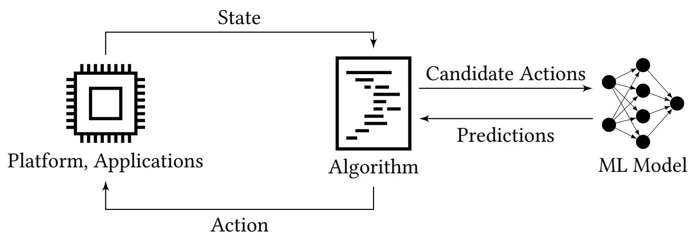


Figure 1.4: ML Usage Pattern 1: Employing ML to predict the impact of a potential management action.

This approach has several advantages. First, proactive management is straightforward to achieve because no action is executed without predicting its impact in advance. Second, the learned model is easily reusable for different objectives or constraints because it is trained to predict the impact of any action, independently of whether this action is beneficial, and, hence, independently of objectives and constraints. Finally, it is rather straightforward to create training examples because one training example can be created directly by observing how a certain management action affects the platform and application state.

The major disadvantage of this pattern is that many inference calls to the model are required, which may negatively affect the run-time overhead. Consequently, the main challenge in this pattern is to select a suitable set of candidate actions that is not too large, which would increase the overhead, but also does not discard the best actions.

1.4.2 Estimate Hidden Properties of the Platform or Applications

The second pattern is to employ an ML model to estimate hidden properties of the platform or applications. Hidden properties are properties that can not be measured directly. This includes for example temperature, which is only measurable at a few distinct locations on the chip, where thermal sensors are located, but also abstract properties of running applications, like their performance sensitivity on V/f level changes. Knowing hidden properties make determining the best actions much easier. This pattern is depicted in Fig. 1.5.

The model can be trained with supervised learning. Obtaining the labels requires measuring the hidden properties. This may be possible either in special measurement setups (e.g., our work in [10] uses a thermal camera to measure the full-chip thermal heatmap), in simulation, or by indirect measurement of hidden properties (e.g., the work presented in Chapter 6 profiles applications at design time at different V/f levels to measure the sensitivity of their performance to the V/f levels).

One of the advantages of this pattern is, similar to the previous one, that the model training is independent of the objective and constraints, which allows reusing the model and training data for different objectives and constraints. Secondly, in contrast to the previous pattern, only one inference call is required, which allows for a lower overhead. The overhead then largely depends on the complexity of the subsequent optimization

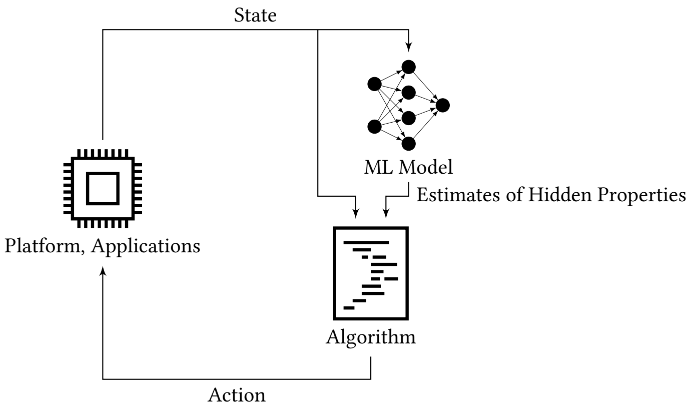


Figure 1.5: ML Usage Pattern 2: Employing ML to estimate hidden properties of the platform or applications.

algorithm. However, as explained earlier, creating the training data may be challenging, since it requires access to the hidden properties to create labels.

1.4.3 Directly Learn Resource Management Policy

The third pattern is to directly learn management actions, i.e., the model output is directly the next action to take. This pattern is visualized in Fig. 1.6. The two main methods on how to train such a model are reinforcement learning (RL) and imitation learning (IL).

RL learns which actions to take by trial and error [35]. RL is designed for Markov decision processes (MDPs), where an agent (partially) observes the state of its environment, selects an action, and executes it to affect the environment. The goal of the agent is to maximize its reward, which is determined based on the state of the environment and the selected action, and should reflect the objectives and constraints. This model is directly applicable to resource management. The main advantage of RL is that it learns at run time, which enables adaptive management. However, this is also its main weakness because run-time learning introduces instability into the learning process, resulting in suboptimal actions being selected, and it comes with a high run-time overhead.

In contrast, IL trains a model only once at design time based on oracle demonstrations that describe the optimal action for various scenarios. Oracle demonstrations are usually created by brute-force testing of many possible actions to find the best one w.r.t. the given objective and constraints. This process may be rather inefficient since many actions must be tested for just one training example. IL trains a model based on these oracle demonstrations with the goal that the model generalizes also to different scenarios. At run time, IL only performs inference of the model, i.e., low overhead. Therefore, IL promises to combine the optimality of the oracle demonstrations with a low overhead. Both RL and IL train models for a specific objective and constraints, which prevents reusing these models for different objective or constraints.

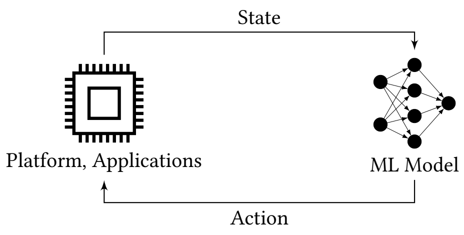


Figure 1.6: ML Usage Pattern 3: Employing ML to directly select management actions.

1.5 Dissertation Contributions

The key contributions of this dissertation are:

- An application mapping and DVFS technique has been developed, using a classical approach based on simple heuristics, for performance maximization under a thermal constraint. It targets many-core processors with physically-distributed, yet logically-shared last-level cache (LLC), resulting in non-uniform cache access latency among the cores. This technique enables us to assess the strengths and weaknesses of classical heuristic resource management. The obtained insights are used to develop ML-based solutions.
- An ML-based application migration technique has been developed, following ML Usage Pattern 1 (Fig. 1.4), for performance maximization under a thermal constraint. An NN model is developed and trained to predict the performance impact of migrating a thread to a different core, enabling proactive management. The prediction is based on hardware performance counters readings, to be able to cope with various characteristics (and, hence, different optimal migrations) of unseen applications. The model is used to determine at each control epoch the best migration w.r.t. the overall performance. Thereby, the technique dynamically adjusts the mapping of the application to maintain peak performance even under changing workload and execution phases.
- A frequency boosting technique has been developed, following ML Usage Pattern 2 (Fig. 1.5), for performance maximization under a thermal constraint. This technique proactively boosts or throttles applications based on a novel metric, called boostability, which integrates the sensitivity of performance and power to V/f changes, and the sensitivity of the hotspot temperature to power into a single value. An NN is used to estimate the sensitivity of performance and power of the running applications to V/f changes, which cannot be measured directly. Predictions of these properties enable proactive management and help tackle the complexity of the problem. Similar to the previous technique, the features of the model are based on hardware performance counters, enabling to cope with unseen applications.
- A technique using DVFS and application migration has been developed, following ML Usage Pattern 3 (Fig. 1.6), for temperature minimization under QoS targets of

heterogeneous clustered processors like Arm big.LITTLE. It uses IL with an NN to tackle the complexity of the platform and application behavior by learning an application migration policy from optimal oracle demonstrations. This technique has been evaluated on a real smartphone chip (*Kirin 970* on *HiKey 970* board). To reduce the run-time overhead, NN inference is performed using its existing generic NN accelerator, the neural processing unit (NPU).

- A technique for resource-adaptive on-device training has been developed. It enables to dynamically adjust the required computational resources for training an NN at negligible overhead. The targeted use case is dynamic resource availability for NN training, e.g., if the training needs to share resources with other running applications. The core idea is to dynamically drop filters of convolutional layers, to trade off resource requirements for convergence speed. The Pareto-optimal per-layer dropout rates are determined using a design space exploration (DSE). This technique is evaluated in a federated learning (FL) setting, where many devices jointly perform distributed training of a large NN model.

1.6 Dissertation Outline

The next chapter discusses the related work for system-level resource management and resource-aware learning. Chapter 3 introduces the experimental framework used to evaluate the techniques developed in this dissertation. These comprise both simulation-based setups and a physical setup with real hardware. Chapters 4 to 8 present the techniques developed in this dissertation. Chapter 4 develops and studies the classical heuristic resource management technique for application mapping and DVFS. The application migration technique based on predicting the impact of potential migrations with ML is presented in Chapter 5. Chapter 6 presents the boosting technique that performs optimization by predicting the sensitivity of performance and power with ML. Chapter 7 introduces the IL-based application migration and DVFS technique for heterogeneous multi-core processors. The technique for resource-adaptive on-device training by dynamically dropping parts of an NN during training is presented in Chapter 8. Finally, Chapter 9 concludes this dissertation and discusses future research directions.

2 Related Work

This chapter discusses related work in three categories. The first category of techniques performs classical resource management, i.e., not based on ML. The second category targets ML-based resource management techniques. The third category discusses resource-aware ML.

2.1 Classical Resource Management

Classical resource management either performs optimization at design time, is based on simple heuristics, or employs analytical models of applications or the platform. The following sections discuss each of these categories.

2.1.1 Resource Management with Design-Time Optimization

If the workload (executed applications and arrival times) and the platform (processor, cooling, etc.) are known at design time, resource management can already be optimized at design time. The main advantages are that virtually unlimited computational resources are available for the optimization, and that alternative management decisions can be tried (profiled). Coskun et al. [53] present a design-time application mapping technique formulated as an integer linear program (ILP) that minimizes the peak temperature, thermal gradients, or energy under performance constraints. Khdr et al. [54] jointly select the degree of parallelism, mapping, and V/f levels of applications under a thermal constraint with dynamic programming. Olsen et al. [55] use design-time profiling to determine the optimal parallelism and thermal-aware mapping to maximize the performance.

The main drawback of design-time optimization is that it is rather unrealistic to assume that the workload is known at design time, e.g., due to varying user activity or input data, as has been discussed in Section 1.2. All these techniques cannot cope with unseen scenarios.

2.1.2 Resource Management with Simple Heuristics

The second class of resource management techniques rely on hand-coded rules or other simple heuristics to make decisions. Processor manufacturers developed frequency boosting techniques like Intel *TurboBoost* [56] or AMD *Turbo Core*, which upscale the V/f levels of all active cores simultaneously if a thermal margin exists, and throttle all cores in case a thermal violation is imminent. These techniques boost the cores without specific knowledge of the applications running on the cores. Resource management at the operating system level, like Android / Linux resource management [57] comprises scheduling and DVFS. Most schedulers are designed for homogeneous processors. In contrast, Global Task Scheduling (GTS) aims at increasing the energy efficiency of heterogeneous processors by migrating mostly-idle applications to the *LITTLE* cluster. DVFS is performed by governors [58], such as *powersave* for power minimization or *ondemand* for a trade-off between power and performance. These policies employ simple hand-coded rules. For instance, *ondemand* tries to keep the CPU utilization between a predefined lower and upper bound by scaling the V/f levels.

Sha et al. [59] propose to operate cores at oscillating V/f levels that are determined such that the maximum transient temperature is below the thermal constraint. Pathania et al. [60] present a run-time application mapping algorithm targeting many-core processors with static non-uniform cache access (S-NUCA) architecture. They greedily minimize the average LLC latency of applications, aiming at maximizing the performance, by prioritizing cores close to the center of the many-core processor. However, they do not take into account application characteristics. Kanduri et al. [61] present a run-time application mapping and power budgeting algorithm for performance maximization under a thermal constraint. They develop a heuristic algorithm to sparsely map threads of an application. In addition, they propose a power budgeting controller and a boosting controller, which determines the application to boost based on the current temperatures of the cores they are running on, i.e., boost the coldest core.

The main advantage of techniques that rely on hand-coded rules or simple heuristics is that they tend to have low complexity and low overhead. However, this comes at the cost of ignoring relevant information like specific application or platform characteristics. Therefore, these techniques cannot cope with the complexity of platform and applications and the resulting management decisions are suboptimal. A plethora of techniques has been developed to improve resource management by using analytical modeling of the platform and applications, which are discussed in the next section.

2.1.3 Analytical Modeling

Most works based on analytical modeling are specific to a certain processor architecture (e.g., homogeneous or heterogeneous) and use models of the platform (e.g., power or temperature). However, the works differ in how they handle applications, where some works assume that application characteristics are known in advance at design time and others target unknown (at design time) applications.

Known Applications Many works target known applications, i.e., use design-time information about the running applications like power consumption or execution time. In contrast to techniques that perform design-time optimization, the arrival times of the applications are not assumed to be known, which mandates run-time optimization.

Some works target network-on-chip (NoC)-based many-core processors with message passing between threads of an application. Ng et al. [62] maximize the performance of a NoC-based many-core processor by application migration. They migrate threads of a single application close to each other and maintain coherent regions of idle cores to map new applications. Their optimization requires information about the execution time and communication volume of the applications. Zhu et al. [63] perform application mapping making a trade-off between temperature and communication latency, which is formulated as a graph problem. They also require models of applications' execution time, communication volume, and power consumption. Wang et al. [64] use application mapping and DVFS to maximize the performance under a thermal constraint. They map threads of an application spatially compactly with idle cores in between. The mapping and DVFS heuristics require knowledge of the communication volume and power consumption, respectively.

Other works target shared-memory architectures. Wang et al. [65] determine the V/f levels that maximize the performance under a power budget. Their approach relies on application-specific models for power consumption and performance depending on the V/f level. Khdr et al. [66] perform aging-constrained performance optimization. They also require design-time performance models of applications, in addition to aging models of the platform. Liu et al. [67] formulate the problem of performance maximization as maximizing the sum of instructions per second (IPS) of all applications. Hence, high-IPS applications are boosted more than the ones with low-IPS. Their optimization requires models of the power consumption and throughput of applications. In addition, focusing only on the absolute value of the IPS ignores relevant other information, like the applications' V/f sensitivity of the performance, which is required to achieve optimal management. Therefore, several techniques consider the V/f sensitivity of the performance, instead. Hankendi et al. [68] maximize the overall system performance under a power constraint considering the sensitivity of the performance, which needs to be known in advance. In addition, they require knowledge of the resource demand of applications. Wang et al. [69] consider the sensitivity of the performance of the applications (known a priori) to determine the V/f levels of their corresponding cores. Particularly, they execute memory-intensive applications at lower V/f levels than compute-intensive applications.

The limitation of these techniques is their dependency on the existence of models (performance, power, etc.) of the applications at design time, which is rather unrealistic, and thereby, they cannot be employed for unknown applications.

Unknown Applications Finally, some works target a priori unknown applications by using only information about the running applications that can be collected at run time. Several works develop analytical performance and power prediction models. Van Craeynest et al. [70] predict the performance of a thread when migrated to another CPU type in a heterogeneous processor. They model the impact of the microarchitecture on individual parts of the cycles per instruction (CPI) stack. However, they do not consider power or temperature. Pricopi et al. [71] extend this concept to also predict power consumption. They first estimate how miss event rates are affected by the microarchitecture and then estimate the performance and power based on the estimated miss event rates. However, they do not present a resource management technique to make use of the predictions.

Bhat et al. [72] perform thermal management of a heterogeneous processor with models of power and temperature. They build analytical power models based on domain knowledge, e.g., physics-based leakage models, to decide when to boost or throttle cores. Using a simple linear performance model, they throttle the applications with the lowest sensitivities, in case of expected thermal violations, thereby minimizing the performance reduction. However, they incompletely make use of these models. For instance, they ignore power and temperature to decide which application to boost or throttle.

These works ignore parts of the complexity in resource management, by not performing resource management itself, or by ignoring application characteristics in parts of the optimization, ultimately leading to suboptimal decisions.

2.2 Learning-Based Resource Management

Recently, a trend towards ML-based resource management can be observed [73]. The following sections discuss techniques that learn the management policy with RL or IL, and techniques that use ML to learn properties of the platform and applications.

2.2.1 Learn Management Policy with Reinforcement Learning

Several works have employed RL for resource management. Some employ table-based centralized Q-learning. Das et al. [74] optimize the reliability under QoS targets using both application migration and DVFS. However, they do not cope with several applications running in parallel. Shafik et al. [75] employ Q-learning for DVFS. The learned policy is not intended to generalize to different workloads. Instead, workload changes are detected and a re-exploration is triggered to update the learned policy. They use a small Q-table to speed up the repeated exploration. Dinakarrao et al. [76] use Q-learning to minimize the energy consumption under reliability constraints by using DVFS. Kwon et al. [77] minimize the energy per cycle by controlling per-cluster DVFS. Liu et al. [78] apply application mapping and DVFS to minimize the temperature under QoS targets. However, they analyze intermediate compiler-level representations of applications, and, hence, their technique is only applicable to known applications. In addition, they do not cope with several applications running in parallel.

A major limitation of centralized table-based Q-learning is that the number of different states and actions is limited by the affordable storage. In addition, large Q-tables slow down the convergence substantially because commonly, only one entry is updated per time step. Therefore, several works approximate the Q-function or the policy function. Lu et al. [79] perform migration for temperature minimization based on per-core temperature measurements. They approximate the Q-function with a radial basis function composition. Other works employ NNs for approximation. Gupta et al. [80] use deep reinforcement learning (DRL) to decide the number of active cores and their V/f levels in a heterogeneous multi-core processor to minimize the energy consumption. Yang et al. [81] minimize the temperature via mapping applications to a core at arrival time.

Other works avoid with large state and action spaces and slow convergence by splitting the centralized agent into many distributed agents, e.g., per core or per application. However, special attention needs to be paid to achieve convergence and cooperativeness between agents. Chen et al. [82] maximize the performance under a global power budget using per-core RL. The individual agents are coordinated using a global heuristic to assign per-core power budgets. Donyanavard et al. [83] employ RL at the core level. A high-level coordinator translates the system goal, e.g., minimizing power, into core-level target IPS. Then, the core-level agents select the V/f level to manage the core IPS accordingly.

RL-based resource management suffers from several problems. It requires to combine objective and constraints into a single scalar reward, which does not reflect their different properties and may lead to suboptimal actions. This is known as reward hacking [84]. Moreover, RL performs training at run time. This may be computationally expensive, preventing a low-overhead implementation, especially if NNs are used. In addition, this may result in instability, due to the requirement to perform exploration at run time, and may even lead to catastrophic forgetting, where the performance of the policy degrades for certain scenarios. Both cases lead to suboptimal management decisions.

2.2.2 Learn Management Policy with Imitation learning

IL has recently gained attention for resource management. Gupta et al. [85] train an ML model to predict the optimal number of active cores and per-cluster V/f levels to minimize the energy. Kim et al. [86] propose an IL technique for DVFS to minimize the energy under a QoS targets. They compare their IL-based technique to an RL-based technique and demonstrate that IL achieves a significantly better management at a lower overhead. However, they train a separate policy per application, and, hence, their technique cannot cope with unknown applications. Mandal et al. [87] use IL to select the types, number, and V/f levels of active cores for several optimization goals, such as minimize the energy under a QoS targets. Finally, Sartor et al. [88] propose a hierarchical IL technique to select the number of active cores and the per-cluster V/f levels to maximize the energy efficiency of a heterogeneous multi-core processor under QoS targets.

These works divide the application execution into phases and record hardware performance counters, performance, and power for each phase at each configuration (number of active cores, V/f levels, etc.). Oracle demonstrations are created by finding the optimal sequence of per-phase configurations. A greedy approach can be used in some cases, e.g., when minimizing energy. More complex heuristics are required in other cases, e.g., when minimizing energy under a performance constraint, as the locally optimal configuration of a phase not necessarily is part of the global optimal sequence of configurations. These techniques initially perform training only on the optimal sequence of configurations, i.e., train the policy to imitate the oracle. They use the *DAgger* algorithm [89] to make the policy more robust. *DAgger* adds a training example whenever the policy differs from the oracle. This example contains information about how to recover from the misprediction.

A major limitation of these techniques is that they only work for power/energy/performance optimization but can not cope with temperature. The reason is that power, performance, and energy of a phase depend only on the used configuration in this phase. This is not the case with temperature, which is subject to both spatial (heat transfer) and temporal (heat capacity) effects that do not exist in power/energy. The reached temperature during a phase additionally depends on all configurations of all previous phases. Consequently, the oracle policy does not work for temperature optimization and would require an exponential number of traces, which is infeasible. In addition, the power sensors required for creating the oracle are often not available in real-world processors.

2.2.3 Learn Properties of the Platform or Applications

Several works employ supervised learning to learn properties of the platform or applications. The models are used to estimate properties that can not be measured at run time, predict the impact of a management action before executing it, or to forecast future events. This section discuss techniques for each of these categories.

Estimate Hidden Properties of the Platform or Applications As discussed in Section 1.4.2, ML can be used to estimate hidden properties of the platform or applications, thereby reconstructing the current, partially observable system state. Analog sensors for power and temperature are costly to implement and, therefore, usually only a few sensors are placed on a chip, making the platform state only partially observable. Bircher et al. [90] employ a simple linear model to estimate the current power from performance counter readings. Sadiqbatcha et al. [91] develop a recurrent NN to estimate the current temperature of the thermal hotspots at run time from processor performance counter readings. Other works aim at estimating application properties. Gupta et al. [92] estimate the sensitivity of the performance of an application to V/f changes with a linear model based on performance counter readings. Walker et al. [93] propose a run-time power model that can be used to estimate the sensitivity of the power to V/f levels.

These works do not perform any resource management but only aim at providing estimates of the platform or application properties. As the available observations to the developed models and required estimates depend on the platform and the management objectives, constraints, and optimization, these models would require adjustment when integrated into a resource management technique.

Predict Impact of Management Actions Another set of works employs ML to predict the impact of potential management actions, as described in Section 1.4.1. The impact of a management action depends on the action but also on potential workload changes that may happen simultaneously to executing the management action. To avoid the challenges of forecasting workload changes, many techniques predict how the system metrics would be now if another action would have been selected, i.e., assume that the workload does not change within the next control step. Consequently, control steps must be short enough to react to changing workloads.

For application scheduling in a high-performance system, Zhang et al. [94] predict how the temperature of a processor will change if a certain application is started. They use application characteristics (performance counters, kernel counters, etc.), as well as CPU-specific features that describe the processor's current properties.

Several works target a more fine-grained control using application migration. Ge et al. [95] predict the steady-state peak temperature of a core after an application migration. They train a very small NN with a single hidden layer. This work targets a homogeneous multi-core processor, i.e., no heterogeneity between cores. Therefore, no workload dependency needs to be learned as the power consumption of an application is the same on all cores, i.e., not affected by migration. Kim et al. [96] predict the performance and power of an application when migrated to another core of a heterogeneous processor. They employ a cascaded NN, where the first NN predicts the expected performance counter values on the target architecture, and the second NN predicts the power and performance based on the performance counter values. Both NNs are small fully-connected feedforward NNs.

As the works in the previous category, these works do not perform any resource management but only aim at providing predictions.

Forecast Future Events Finally, several works aim at forecasting future events, such as changes in the workload or environment. Such forecasts may provide relevant inputs to a resource management technique. Coşkun et al. [97] forecast future workload behavior, such as temperature, core utilization, and instructions per cycle (IPC). They assume repeating patterns in the workload. Abad et al. [98] use an NN to forecast the temperature in a multi-core processor based on information about the workload, as well as information about the V/f level and cooling fan speed.

Similar to the previous two categories, these works do not perform any resource management themselves but only aim at providing relevant inputs.

2.3 Resource-Aware Learning

The majority of works on resource-aware machine learning focuses on resource-aware *inference* [99, 100, 101]. These techniques dynamically adapt the resource requirements of the inference but are not applicable to training. Resource-aware *training* is only recently getting increasing attention, mostly in the context of distributed learning, where the available resources for training vary between devices and additionally on each device over time, as will be discussed in Chapter 8.

Some techniques dynamically drop training data. Li et al. [102] present *FedProx*, which allows devices that participate in distributed training to drop training examples that could not be processed with the available resources. However, dropping data greatly reduces the achievable model accuracy.

Several techniques train separate models on each device, and use distributed learning to improve these individual models. For instance, Shi et al. [103] optimize the hyperparameters of the per-device models with distributed model-based optimization. However, they do not synchronize the learned concepts by the models. Recently, techniques based on *Federated Distillation* [104, 105, 106] have been presented that allow each device in a distributed system to train an NN model with different topology that fits best its available resources. However, synchronizing knowledge between these different models is rather inefficient because the NNs do not necessarily share a common structure, reducing the achievable accuracy. In addition, such techniques cannot cope with time-varying resource availability.

Instead of training independent NN topologies on the devices, several techniques perform training only on a dynamic subset of a single NN. In contrast to *Federated Distillation*, the obtained parameter updates during training can be merged easily. The size of the subset determines a trade-off between the required resources and the achievable convergence speed. Horváth et al. [107], Yu et al. [108], and Diao et al. [109] select subsets of the NN in a hierarchical way, where smaller subsets are fully contained in larger subsets. For instance, Diao et al. [109] introduce a shrinkage ratio s that determines the ratio of removed hidden channels in convolutional layers to reduce the resource requirements of the NN. The same parameter s is applied repeatedly to all layers to obtain several NN subsets with decreasing resource requirements. However, using hierarchical subsets is restricted to a small number of supported subsets to avoid accuracy losses [99]. This prevents fine-grained adjustability of resource requirements for training, which leads to wasted computational resources.

This limitation can be avoided by selecting subsets randomly, i.e., use non-hierarchical subsets. Xu et al. [110] randomly remove neurons before training on slow devices at the beginning of local training. Graham et al. [111] study the suitability of dropout [51], which is commonly used as a regularization method, to reduce resource requirements. They observe that computations can only be saved if dropout is done in a structured way, i.e., a contiguous set of neurons is dropped for all samples of a mini-batch. Caldas et al. [112] perform structured dropout before starting local training and train a repacked smaller network on the devices. However, they use the same dropout rate for all devices that participate in distributed training, which needs to be selected according to the device with the lowest resources, and leads to wasted resources on other devices. In addition, they use a single dropout rate for all layers, which is suboptimal and reduces the achievable convergence speed. All these works select the trained subset at the beginning of training, and do not support changing it during training. This does not allow to adapt to changing resource availability during training on the devices.

None of these techniques supports heterogeneous and changing resource availability for local training on the devices, while still achieving a high model accuracy.

3 Experimental Framework

The techniques developed in this dissertation are evaluated experimentally. Two different setups are used to evaluate resource management policies, a simulation-based setup and a setup based on real hardware. Finally, a third setup based on FL is employed to evaluate resource-aware on-device ML techniques.

3.1 Simulation-Based Setup

The simulation-based setup uses *HotSniper* [113], which combines the *Sniper* [114] multi-/many-core simulator for performance simulation with *McPAT* [115] for periodic interval-based power simulation, and *HotSpot* [44] for periodic interval-based thermal simulation. *Sniper* enables multi-program simulation of multi-threaded applications with full modeling of shared resource contention, such as on cache capacity or memory bandwidth. The tight integration with *McPAT* and *HotSpot* enables resource management policies to access information about the power and temperature during execution. We always simulate the full execution of the benchmarks [116]. The main advantage of the simulation-based setup is its flexibility to simulate different systems (e.g., different hardware architectures).

Resource Management API and Simulation Automation During this dissertation, extensions for *HotSniper* have been developed and published as open source contribution[1]. The main extensions are a resource management API and simulation automation flow. The resource management API enables to implement custom policies for application mapping, application migration, or DVFS by implementing well-defined C++ interfaces. The simulation automation flow enables to run multiple different simulations (e.g., different workloads, different resource management policies) in batch mode. It extends the configuration file syntax of *HotSniper* to automatically switch between different configuration values in each simulation. In addition, it provides a Python API for automated evaluation of the simulation results.

CPI Stacks to Represent Application Characteristics A cycle stack [117] divides the number of CPU cycles for executing a sequence of instructions into several components. The base cycle count is the number of cycles that would have been required if the CPU never stalled, i.e., loads and stores finish immediately, no branch mispredictions, no interdependencies between instructions, etc. Every additional cycle is assigned to the architectural

[1] `https://github.com/anujpathania/HotSniper`

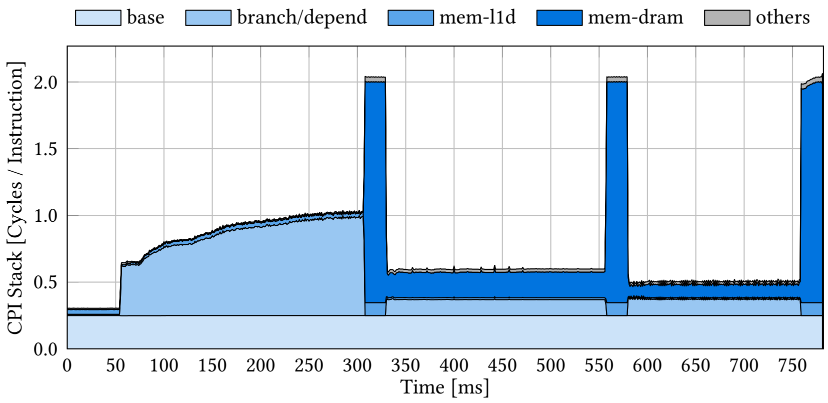


Figure 3.1: The CPI stack of *SPLASH-2 fft* clearly shows its execution phases and provides insights into what causes them (e.g., instruction dependencies or many DRAM accesses).

component that causes it. This comprises data dependencies between instructions, bottlenecks in the CPU (e.g., limited issue width, limited number of reservation stations), branch mispredictions, and the memory hierarchy (e.g., TLB, L1 cache, L2 cache, LLC, RAM). A CPI stack normalizes the cycle stack to the number of executed instructions. CPI stacks are a valuable tool for system designers because they visualize performance bottlenecks and represent the characteristics of the applications. Furthermore, CPI stacks can be obtained from common performance counters at run time with minimal overhead [70]. *HotSniper* directly provides CPI stacks per each thread during execution.

Fig. 3.1 shows the CPI stack of *SPLASH-2 fft* over time when running at a constant frequency of 3.0 GHz on the architecture described in Section 3.1.1. A certain minimum number of cycles are always required to execute any instructions, called the base component in a CPI stack. The other components clearly show the application execution phases. Initially, data and branch dependencies between instructions cause many stall cycles in the CPU pipeline. Later, *fft* is more memory-intensive with three short distinct phases of high dynamic random access memory (DRAM) activity. A CPI stack not only allows to detect the execution phases of *fft* with different characteristics, but also presents an explanation of the causes, which is relevant information for application-aware optimization in the resource management.

Thermal Modeling This setup uses *HotSpot* to perform thermal modeling of the chip and the cooling system, comprising the thermal interface material, heatspreader, and heat sink. *HotSpot* uses an RC-thermal network where temperature and heat flow correspond to voltage and current in an electrical circuit, respectively [44]. The many-core processor and its cooling system are modeled by N thermal nodes. The first n nodes correspond to the n cores of the many-core processor along with their associated components like private caches, LLC banks, or NoC routers. The remaining $N - n$ nodes describe the

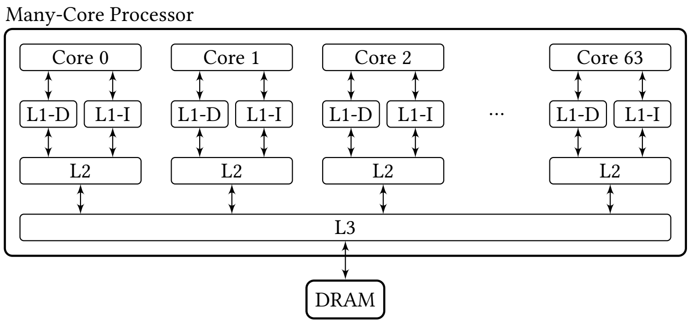


Figure 3.2: The 8×8 bus-based homogeneous many-core architecture employed in the simulation-based setup.

cooling system. The changes in temperatures $\mathbf{T}' = [T_i']_N$ of all nodes are described by N differential equations:

$$\mathbf{A}\mathbf{T}' + \mathbf{B}\mathbf{T} = \mathbf{P} + T_{amb}\mathbf{G} \tag{3.1}$$

Thereby, the matrix $\mathbf{A} = [A_{ij}]_{N\times N}$ describes the dynamic thermal behavior, i.e., thermal capacitance values, the vector $\mathbf{T} = [T_i]_N$ contains the current temperatures, the matrix $\mathbf{B} = [B_{ij}]_{N\times N}$ describes the thermal conductivity between nodes, the vector $\mathbf{P} = [P_i]_N$ contains the current power consumption of all nodes, T_{amb} is the ambient temperature, and the vector $\mathbf{G} = [G_i]_N$ describes the thermal conductivity between the nodes and ambient.

For a constant power consumption, the temperature asymptotically approaches the *steady-state temperature* $\mathbf{T}_{steady} = [T_{steady,i}]_N$, which is described by

$$\mathbf{T}_{steady} = \mathbf{B}^{-1}\mathbf{P} + \mathbf{B}^{-1}T_{amb}\mathbf{G} \tag{3.2}$$

where matrix $\mathbf{B}^{-1} = [B_{ij}^{-1}]_{N\times N}$ is the inverse of matrix $\mathbf{B}$. The steady-state temperature is much cheaper to compute than transient temperatures. This is as the main required computation is a single matrix-vector multiplication. As low run-time overhead is always a concern in resource management, the steady-state temperature may be used to lower the complexity of the optimization. However, this may come at the risk of transient thermal violations that are not captured by the steady-state temperature. Pagani et al. [118] have shown that thermal overshoot can happen when the distribution of power consumption on the chip changes, even if the steady-state temperature is always thermally safe. It is important to notice here that while several techniques in this dissertation consider the steady-state temperature within their optimization, evaluations are always performed on the transient temperature, and the observed thermal violations are negligible.

3.1.1 Homogeneous Many-Core Processor

HotSniper is configurable to simulate many different architectures. The first studied architecture is a homogeneous bus-based 64-core many-core processor with the *gainestown*

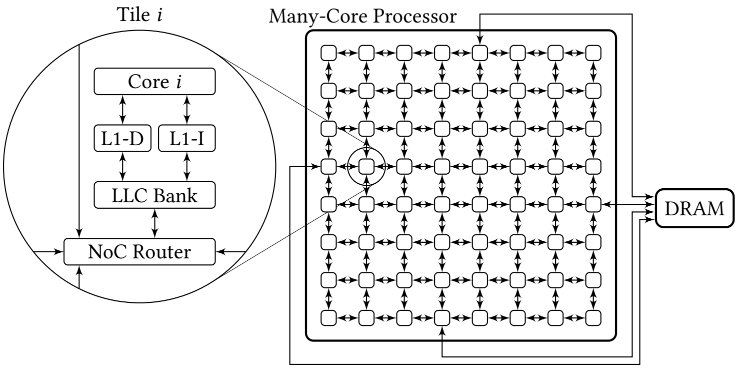


Figure 3.3: The 8×8 NoC-based many-core architecture with S-NUCA LLC employed in the simulation-based setup.

microarchitecture. It is depicted in Fig. 3.2. This architecture is close to many commercially-used desktop and server processors [119]. The 32 KB L1-D, 32 KB L1-I, and 256 KB L2 caches are private per core. The 8 MB L3 cache is shared among all cores. The many-core processor is modeled in the 14 nm FinFET technology node. The most recent technology that is supported by *McPAT* is 22nm FinFET. We, therefore, estimate the power consumption at 22nm and scale it to 14nm FinFET with factors based on [33]. DVFS selects frequencies between 1.0 GHz and 4.0 GHz in steps of 100 MHz. The cooling is modeled with the default parameters of *HotSpot*.

3.1.2 S-NUCA Many-Core Processor

The second studied processor employs an S-NUCA architecture [43], which is depicted in Fig. 3.3. S-NUCA is also employed in commercially available many-core processors [120]. In an S-NUCA architecture, banks of the LLC are physically distributed among all tiles of the many-core processor, yet still logically form a single large cache that is shared among all cores. Each bank of the LLC is co-located with a CPU core within a tile. The main advantage of S-NUCA is that it allows for a high scalability in the number of cores because distributing the LLC and routing the memory traffic on the NoC prevents a single LLC bottleneck. To achieve such scalability and enable a simple hardware implementation, the mapping of memory address to LLC bank is *static*, i.e., the mapping is based on the lower-level bits of the address tag. All cores are connected by a NoC, with a NoC router in each tile. Prior work [60] has shown that this architecture is *heterogeneous* w.r.t. LLC latency, where cores closer to the center of the many-core processor experience a lower LLC latency than cores closer to the corners.

The simulated many-core processor comprises 64 cores aligned in an 8×8 grid. The per-core private L1 data and instruction caches have a size of 16 KB each and an access latency of 3 cycles. The shared LLC uses S-NUCA policy and has a total size of 8 MB where each core holds an LLC bank of size 128 KB with a latency of 8 cycles. The NoC uses

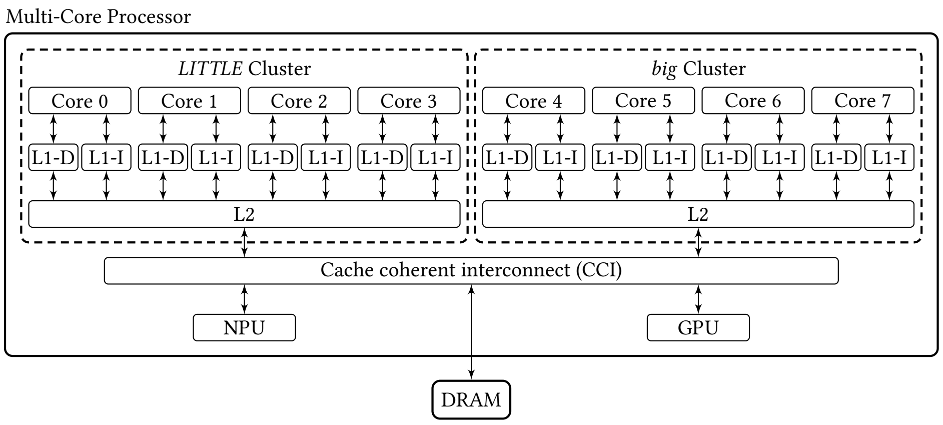


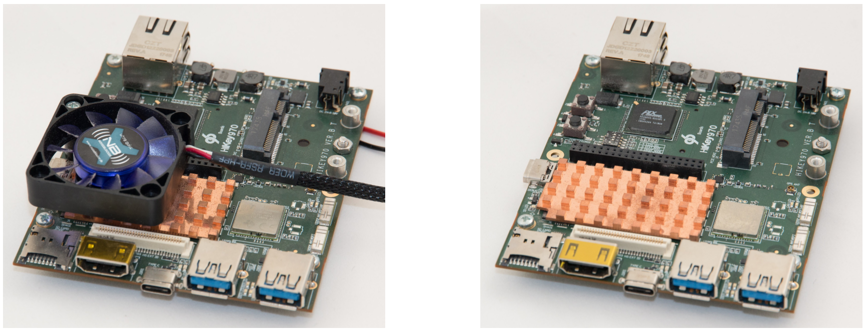
Figure 3.4: The Arm big.LITTLE architecture employed in the physical setup with a *HiKey 970* board.

XY-routing with a latency of 1.5 ns (6 CPU cycles @ 4 GHz) per hop and a link width of 256 bits. DVFS sets core frequencies in multiples of 100 MHz from a minimum of 1.0 GHz up to a maximum of 4.0 GHz. The many-core processor is modeled to be fabricated in the 14nm FinFET technology node. As with the homogeneous architecture, we use *McPAT* to estimate power at 22nm and scale it to 14nm. The area of each core is 0.81 mm^2. We use the default *HotSpot* cooling parameters. Idle cores consume 0.3 W because their associated globally-shared LLC banks need to stay active.

3.2 Physical Setup with Real Hardware

In addition to the simulation-based setup, we also use a physical setup that is based on a *HiKey 970* [121] board with a *HiSilicon Kirin 970* smartphone system-on-chip (SoC). Fig. 3.4 shows the SoC architecture. It implements the common Arm big.LITTLE architecture, which combines powerful *big* cores (in our platform four Arm Cortex-A73 cores [41]) with energy-efficient *LITTLE* cores (in our platform four Arm Cortex-A53 cores [42]). Thereby, the system comprises two *clusters* with very different characteristics. Arm big.LITTLE is the prevalent architecture in mobile SoCs. However, there are also recent advances to employ big.LITTLE architectures in desktop and server processors, like Intel Alder Lake-S and AMD Zen 5.

The SoC on *HiKey 970* is manufactured in 10nm. It supports per-cluster DVFS with frequencies up to 1.84 GHz and 2.36 GHz, respectively. Furthermore, it comes with an NPU to accelerate the inference of NNs. However, the NPU does not support NN training. The board is placed in an air-conditioned room to maintain a constant ambient temperature. Two different cooling settings are supported. The first setting is forced-convection cooling by adding a fan with constant power (Fig. 3.5a). The second setting is passive air cooling with the provided heat sink (Fig. 3.5b). The on-chip temperature is monitored with the



(a) With a fan **(b)** Without a fan

Figure 3.5: The *HiKey 970* board with (a) and without (b) a fan.

on-board thermal sensor with a frequency of 20 Hz. The board runs Android 8.0. We use the Linux *CPU affinity* feature to pin threads to cores to implement custom applications mapping and migration, and use the Linux *governor* subsystem, specifically the *userspace* governor, to implement custom DVFS policies.

3.3 Federated Learning Setup

Techniques for resource-aware on-device ML are evaluated in an FL setting [122]. FL has emerged as an alternative to traditional centralized training of an ML model on a single server, which suffers from several problems. First, training data is often obtained at end devices, such as smartphones, internet of things (IoT) nodes, etc. Collecting all training data at the server is costly, as the raw data would need to be sent all over the network to that centralized entity [123]. Second, the training might use users' private data, mandating that the data must not leave the users' devices. In contrast, FL performs *distributed training* on many participating devices, where each device holds its own private training data, and knowledge is exchanged between the devices by synchronizing the parameters of an NN model. However, training a deep NN model is resource-hungry in terms of computation, energy, time, etc. Hence, the available resources for training are limited. In summary, FL is a suitable use case to study resource-aware learning techniques because it performs training of an NN model on devices such as smartphones or IoT nodes, which are resource-constrained in terms of computations, power, energy, etc.

Fig. 3.6 shows the FL-based setup, which simulates one server and N distributed devices that act as clients. Each device i holds its own local training data X_i. We target a synchronous coordination scheme, which divides the training into many rounds. At the beginning of a round, the server selects n devices to participate in the training. Each selected device downloads the recent model from the server and trains it with its local data, considering the available resources for training. After training, each device sends weight updates back to the server. The server combines all received weight updates to a

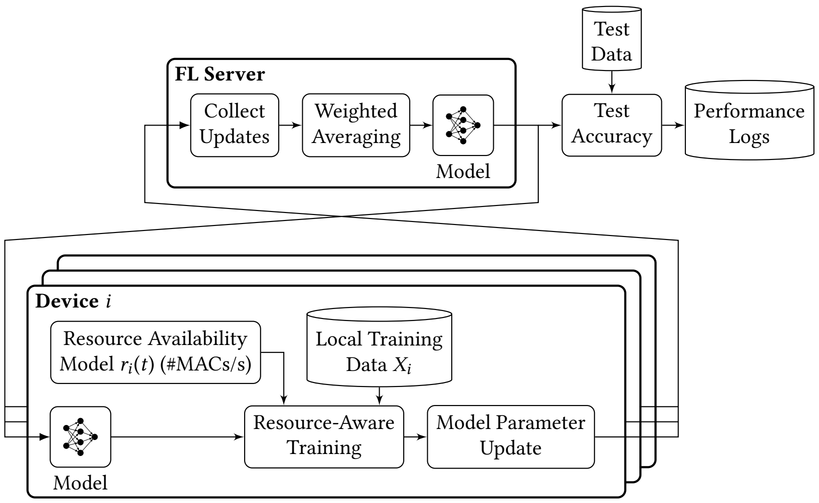


Figure 3.6: FL-based setup to evaluate techniques for resource-aware ML.

single update by weighted averaging. Updates from devices that take too long to perform the training (stragglers) arrive at the server too late and are discarded. At the end of a round, the server has obtained a new model. The accuracy of this model is measured in each round with the help of test data that is independent of the devices' data.

The devices are subject to time-varying limited computational resource availability for training. To which degree the availability of a certain resource affects the local training time of an NN depends on the NN and hyperparameters, but also on the deep learning implementation (the used library) and the underlying hardware [124]. We abstract from such specifics of the hardware and software implementation, and from the constrained physical resource to keep this setup applicable to many systems by representing the resource availability in the number of MACs that a device can calculate per time given its specifications, implementation, and available resources. MAC operations are the fundamental building block of NNs (e.g., fully-connected and convolutional layers) and account for the great majority of operations [125]. The resource availability $r_i(t)$ depend on the device i, and the current time t. Resources may change at any time, i.e., also within an FL round. Resources are not modeled to be known ahead of time.

The setup offers several points of customization. Different NN topologies and data sets can be studied. On the server, the device (client) selection and weighted averaging algorithm can be changed. On the devices, the resource availability model and the resource-aware training algorithm can be customized.

4 Classical Heuristic Resource Management

Sophisticated application mapping and power budgeting policies are required to efficiently utilize the increasing number of cores in many-core processors [31]. An application mapping policy determines for each thread of a multi-threaded application the core that it is executed on, while the power budgeting policy sets a limit to the power consumptions of the cores in order to prevent thermal violations. Both policies need to be matched to attain best performance. This section introduces a classical heuristic application mapping and power budgeting policy for many-core processors with S-NUCA architecture, as introduced in Section 3.1.2. The gained insights into the advantages and shortcomings of classical heuristic management are used in the development of ML-based solutions in Chapters 5 to 7.

4.1 Analysis of the S-NUCA Many-Core Architecture

We first present an analysis of LLC latency and temperature on an S-NUCA architecture.

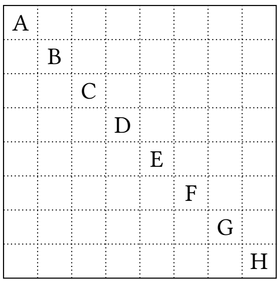
4.1.1 Last-Level Cache Access Latency

The latency of a single LLC access is affected by the hop count between the thread's core and the LLC bank, which corresponds to the Manhattan distance (MD). The MD between two points **a** and **b** is the sum of absolute differences of their Cartesian coordinates:

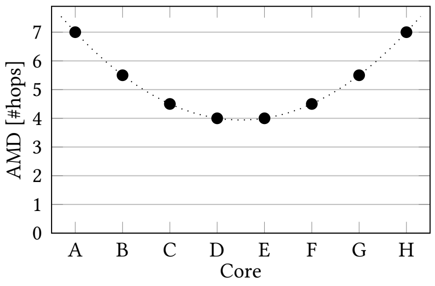
$$\mathrm{MD}(\mathbf{a}, \mathbf{b}) = \sum_i |a_i - b_i| \tag{4.1}$$

The fine-grained interleaving of memory addresses to LLC banks causes a thread to access the LLC banks on all cores equally likely irrespective of the core it is executing on [60]. Thereby, the *average* LLC latency correlates with the average Manhattan distance (AMD)

This chapter is mainly based on [7, 9].



(a) Core Locations



(b) LLC Access Characteristics

Figure 4.1: AMD of cores to the LLC banks is lowest for the cores closest to the center of the many-core processor. Cores C and F close to the center experience only a small increase in the AMD.

of the core that the thread is executing on to all tiles (LLC banks). The AMD of a core with location (x, y) on a many-core processor with $X \times Y$ cores is defined by

$$\mathrm{AMD}(x, y) = \frac{1}{X \cdot Y} \sum_{i=1}^{X} \sum_{j=1}^{Y} \mathrm{MD}\left(\begin{pmatrix} i \\ j \end{pmatrix}, \begin{pmatrix} x \\ y \end{pmatrix}\right) \tag{4.2}$$

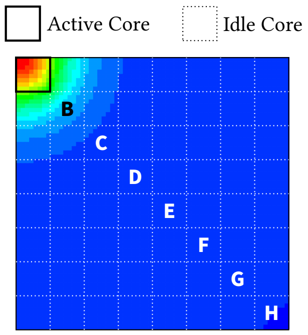
Cores that have a lower AMD to all other cores (LLC banks) have a lower average LLC latency [60]. For quadratic $n \times n$ many-core processors, i.e., $n = X = Y$, Eq. (4.2) can be rewritten as

$$\mathrm{AMD}(x, y) = \frac{1}{n} \left\| \begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} \frac{n+1}{2} \\ \frac{n+1}{2} \end{pmatrix} \right\|^2 + \frac{n^2 - 1}{2 \cdot n} \tag{4.3}$$

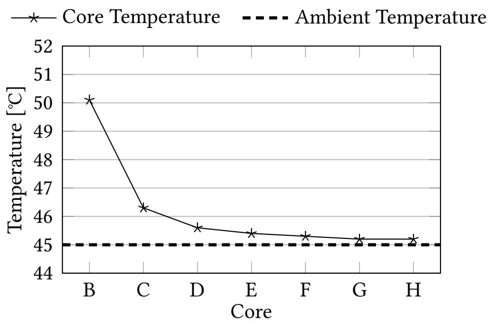
The AMD of cores increase quadratically with their Euclidian distance to the center of the many-core processor. This is also shown in Fig. 4.1. *The LLC latency in S-NUCA is heterogeneous, i.e., lowest close to the center of the many-core processor and higher closer to the corners.* Furthermore, the AMD to LLC banks increases steeply near the corner of the many-core processor (cores A and H), but changes only slightly near the center (cores C and F). *Therefore, we make the observation that cores near the center of the many-core processor experience only a slight increase in LLC latency.* These observations are not restricted to S-NUCA many-core processors. Similar observations also hold true for directory-based many-core processors where the cache directory is distributed along the cores [126].

4.1.2 Thermal Aspects

The second important factor that affects the performance of applications running on a thermally-constrained many-core processor is the power. Due to increasing power densities in smaller technology nodes, active cores of a many-core processor are constrained by



(a) Core Locations



(b) Core Temperatures

Figure 4.2: Heat conductance across the many-core processor falls off approximately exponentially. Cores D to H have similar temperatures even though their distances to the active core differ significantly.

a power budget that aims to ensure thermally safe operation [127]. The most commonly used power budget is Thermal Design Power (TDP), which is a constant per-chip power budget determined at design time. It is unaware of the number and location of active cores (the mapping of threads to cores). Hence, it needs to be pessimistic to be able to prevent thermal violations. Pagani et al. [128] present Thermal Safe Power (TSP) as an alternative to overcome this limitation by considering the actual mapping. TSP provides a per-core power budget for a given mapping that prevents thermal violations in the steady state. By considering the actual mapping, TSP can provide a higher power budget than TDP while still being thermally safe.

Pinning threads to spatially close cores leads to a reduced power budget to avoid thermal hotspots, whereas spreading these threads far across the many-core processor leads to better heat distribution and thereby to a higher power budget. The increased power budget can potentially increase the performance of an application by allowing to operate the cores at higher V/f levels. Fig. 4.2 shows that the heat conductance on the many-core processor approximately falls off exponentially with the distance. Cores B to H in Fig. 4.2 are all idle, but their temperatures differ significantly. The farther a core is from the active core, the lower is its temperature. However, once a sufficient distance is reached, the changes in temperature are insignificant. Therefore, the temperatures of cores D to H are nearly identical even though their distances to the active core differ significantly. *We observe that maximizing the power budget requires to spatially distribute active cores on the many-core processor. However, if active cores have sufficiently high distance to each other, further increasing it only slightly increases the power budget.*

4.1.3 Trade-off Between Last-Level Cache Latency and Power Budget

As described in Sections 4.1.1 and 4.1.2 and illustrated in Fig. 4.3, minimum LLC latency and maximum power budget are obtained by contradictory mapping decisions. While the former requires clustering threads near the center of the many-core processor, the

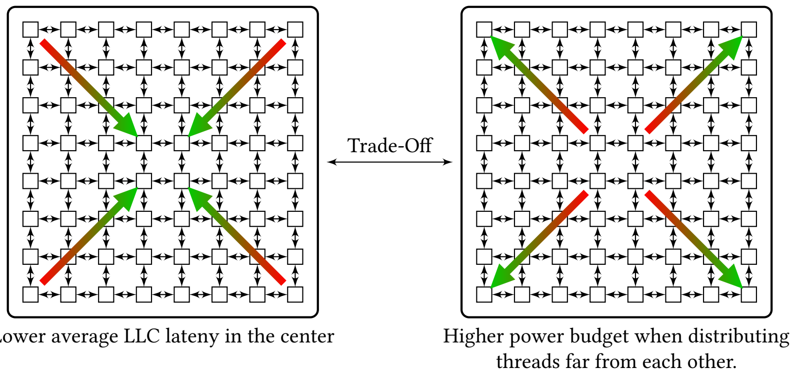

Trade-Off
Lower average LLC lateny in the center
Higher power budget when distributing threads far from each other.

Figure 4.3: Trade-off between power budget and LLC latency in S-NUCA many-core processors.

latter requires maximizing the spatial distances between threads by distributing them across the whole chip. Since both metrics cannot be simultaneously optimal, there exist multiple Pareto-optimal mappings. *In a Pareto-optimal mapping, the power budget cannot be improved without sacrificing LLC latency and vice versa.* Based on the insights from Figs. 4.1 and 4.2, it is very likely that there exist many near-optimal mappings with respect to both optimization goals. *To maximize application performance, a trade-off between pinning threads as far as possible from each other in order to maximize the power budget and pinning threads as near as possible to the center in order to minimize the LLC latency needs to be found.* This trade-off has not been observed before. We demonstrate it using the following motivational examples.

4.2 Motivational Examples

This section presents motivational examples for application mapping and dynamic power budget reallocation, which are the two means employed in this section.

Application Mapping Fig. 4.4 shows the execution times of different four-threaded *PARSEC* applications on a 64-core S-NUCA many-core processor under three different mappings. TSP is used to calculate a uniform per-core power budget for each mapping. Mapping 4.4a (Corner) uses cores close to the corners to gain a high distance between active cores and, thereby, maximize the power budget, reaching 1.37 W. Using the exact corner cores would result in a lower power budget since the cores in the corners have fewer neighboring cores that can absorb heat. Mapping 4.4b (Center) uses cores as close as possible to the center of the many-core processor to minimize the LLC latency, reaching AMD = 4 hops. Mapping 4.4c (Intermediate) uses inner, but not the center cores to achieve a trade-off between a low LLC latency (AMD = 4.5 hops) and a high power budget (1.35 W). DVFS enables cores to run at different V/f levels allowing them to restrict the

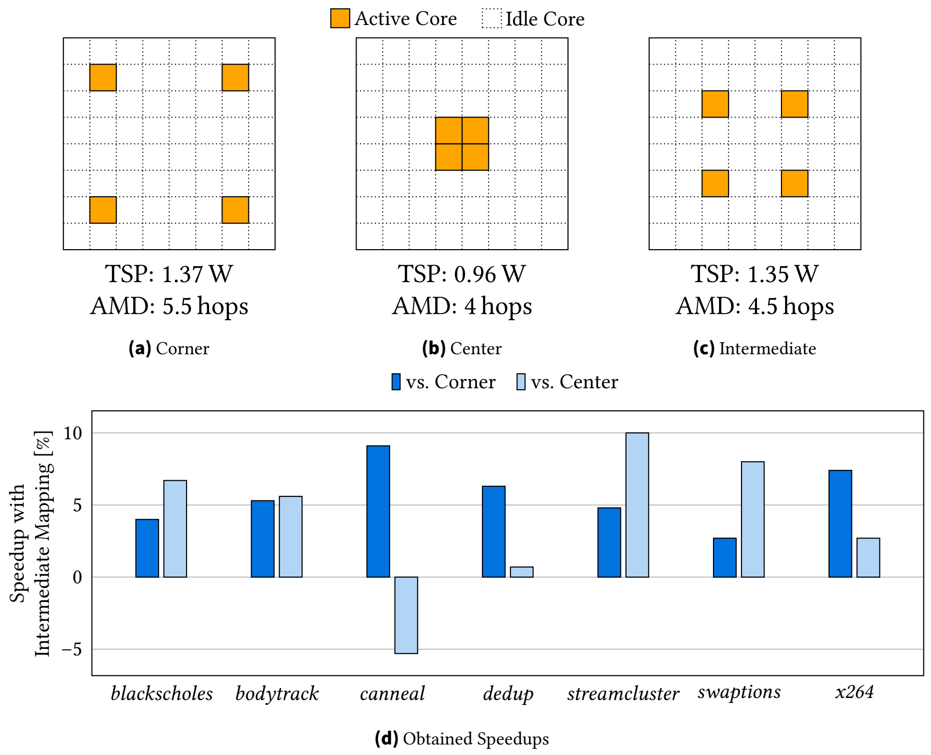


Figure 4.4: Impact of mapping on the performance of *PARSEC* applications on an S-NUCA many-core processor. Neither mapping threads to the corner cores nor mapping them to the center cores results in the best performance. Instead, an intermediate mapping yields the highest performance for almost all *PARSEC* applications.

power consumption by sacrificing performance. The V/f levels of the cores are adapted dynamically at run time to not exceed the power budget.

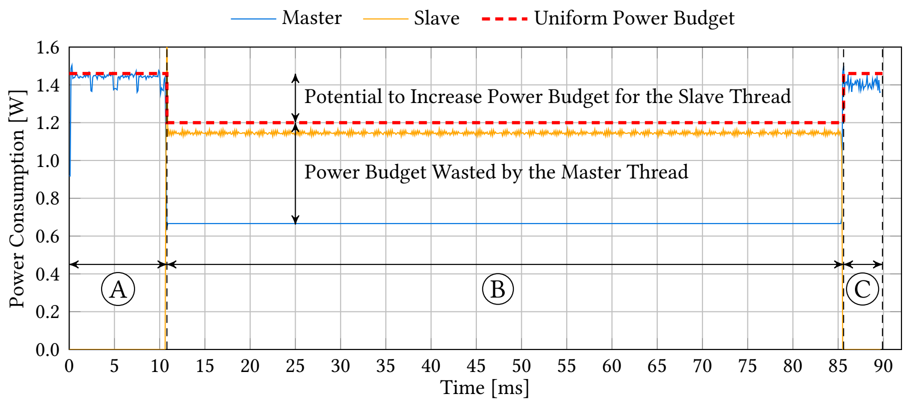
Fig. 4.4d shows the speedup achieved with the intermediate mapping over the other two mappings. The intermediate mapping almost always provides the highest performance. Mapping to the corners suffers from high LLC latency and mapping to the center suffers from a low power budget, while mapping to intermediate cores has both a relatively high power budget and a low LLC latency. The average speedup with the intermediate mapping among all applications is 5.7 % (or 4.1 %) compared to mapping to corners (or center).

Different *PARSEC* applications experience different performance with different application mappings due to their different characteristics. The intermediate mapping provides the best performance for all applications except *canneal*, for which a mapping to the center is the best. *Canneal* is a memory-intensive application and has the most LLC accesses per second among all *PARSEC* applications. Its performance is dominated by the memory access latency. Therefore, the LLC latency has a much higher impact on its performance than the power budget which eliminates all scope for any trade-off. *Overall, this example demonstrates that the application mapping needs to make a trade-off between maximizing the power budget and minimizing the LLC latency.*

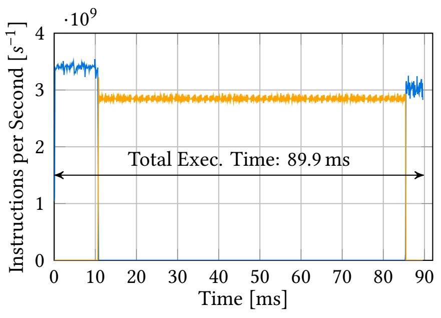
Dynamic Power Budget Reallocation The execution of applications typically shows distinct phases [129]. Fig. 4.5a shows the execution of two-threaded *blackscholes.* Its execution comprises three phases denoted by A, B, and C. In Phase A, the master thread prepares the work while the slave thread has not yet been spawned. Phase B starts when the master thread spawns the slave thread. The slave thread processes the prepared data while the master thread is idle. After the slave thread has finished execution, the master thread resumes and completes the application execution in Phase C. Fig. 4.5a also shows the power budgets of the threads if a uniform TSP budget is used. During Phase A, the slave thread is not yet spawned and consequently, the master thread receives a high power budget. During Phase B, master and slave threads both are mapped to a separate core, reducing the power budget to prevent thermal violations. During Phase C, the master thread is the only thread and its power budget returns to the same level as in Phase A. This example uses two threads for the sake of simplifying illustrations. The same observations also hold true for higher parallelism levels (higher number of slave threads).

With a uniform power budget, such as provided by TSP, both master and slave threads are provided the same power budget in Phase B. However, the behaviors of these two threads differ strongly. The slave thread is compute-intensive and would benefit from a higher power budget, while the master thread is idle and cannot make use of its assigned power budget. A non-uniform power budget would be beneficial instead. The power budget of the master thread can be reduced to a minimum without sacrificing performance, thereby allowing to increase the slave's power budget while still being thermally safe. Figs. 4.5b and 4.5c show the IPS with the uniform and non-uniform power budgets, respectively. *Reallocating* the unused power budget from the master thread to the slave thread results in a higher IPS for the slave thread and ultimately in a 5.0 % reduction in the response time of the whole application. Fig. 4.5d shows the observed speedups for all *PARSEC* applications that support a parallelism level of two threads. Like observed with the mapping of threads to cores, different applications respond differently to different policies. The highest speedup is observed with *fluidanimate* (7.8 %), the lowest with *canneal* (−0.4 %). The reason why *canneal* does not show any speedup is that it is memory-intensive and cannot consume high power since the cores are stalled often while waiting for the memory. Therefore, the power budget does not restrict *canneal* at all and reallocation has no effect. The average speedup for these five applications is 5.0 %.

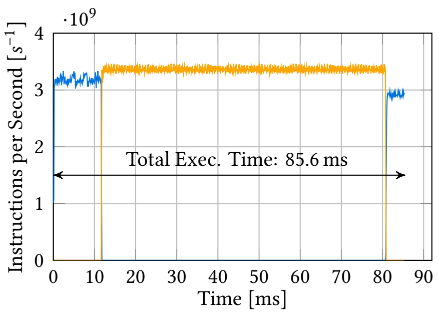
Not only the threads of a single application can behave heterogeneously, but also threads of different applications may show heterogeneous characteristics. Fig. 4.6 shows the peak power consumption of a single thread for different *PARSEC* applications, when mapped to the center of the many-core processor and operated at the maximum V/f level. Their peak power consumption varies strongly. It is apparent that not all applications can utilize high power budgets, e.g., *canneal* or *x264.* Fig. 4.7 shows an example of how this observation can be exploited to boost the performance. *Canneal*, which is a memory-intensive application,



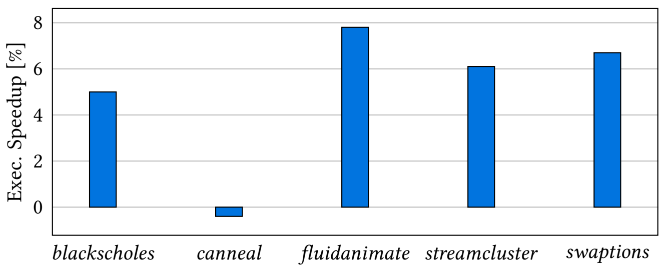
(a) Power Consumption of *Blackscholes* with Uniform Power Budget



(b) IPS with Uniform Power Budget



(c) IPS with Non-Uniform Power Budget



(d) Obtained Speedups with Non-Uniform Power Budget

Figure 4.5: Using a uniform power budget for all threads results in wasted power budget. (a) shows that the *blackscholes* master thread cannot use its power budget in Phase B, i.e., for most of the time. Reallocating this power budget to the slave thread results in a higher performance (b and c). (d) shows the observed speedups with a non-uniform (reallocated) power budget over a uniform power budget for all *PARSEC* applications that support a parallelism level of two threads.

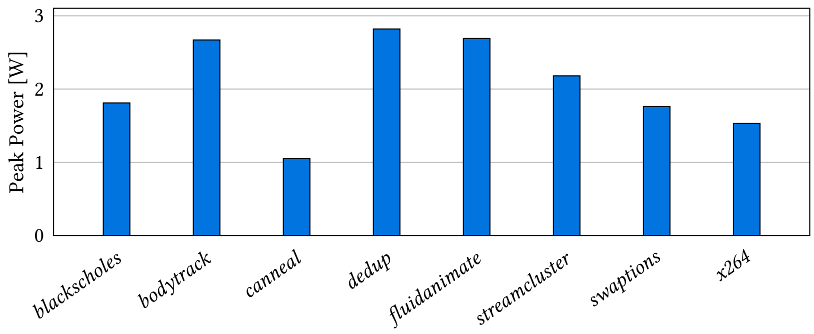


Figure 4.6: The peak power consumption of a single thread of different *PARSEC* applications differs strongly.

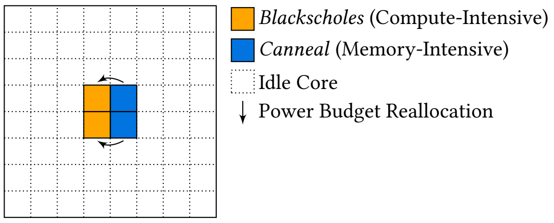

🟧 *Blackscholes* (Compute-Intensive)
🟦 *Canneal* (Memory-Intensive)
⬚ Idle Core
↓ Power Budget Reallocation

Figure 4.7: Reallocation of unused power budget between heterogeneous applications increases the performance. The response time of the compute-intensive *blackscholes* reduces from 97.8 ms to 95.4 ms while the response time of the memory-intensive *canneal* does not change.

cannot make use of its assigned power budget. *Blackscholes*, on the other hand, is a compute-intensive application and thereby could benefit from a higher power budget. To prevent the impact of inhomogeneous LLC latency, all threads are pinned to cores with the same AMD class. Reducing the power budget for the threads of *canneal* to a value close to its actual power consumption allows to increase the power budget of the threads of *blackscholes*. In this example, threads of the same application still receive the same power budget. This allows us to separate the effects of power budget reallocation between different applications and between threads of the same application. The performance of *canneal* is unaffected by this reallocation since it was not able to make use of its power budget in the first place. At the same time, the response time of *blackscholes* was reduced from 97.8 ms to 95.4 ms (2.5 %). Therefore, the overall performance is increased by reallocating the unused power budget. Similar observations also hold true for other combinations of applications, where one application is memory-intensive and another application is compute-intensive. *These examples show that power budget reallocation is required to improve the performance over a uniform power budget.*

4.3 Novel Contributions

The novel contributions of this chapter are as follows:

- The work presented in this chapter is the first to explore the trade-off between power budget and LLC latency for application mapping on S-NUCA many-core processors. We develop a method to determine all Pareto-optimal mappings using an ILP.
- We present the first run-time application mapping algorithm that improves the performance by exploiting this trade-off while considering previously mapped applications.
- This algorithm is extended by a run-time component that dynamically reallocates power budgets between threads to react to different application execution phases and application heterogeneity. Both algorithms works in tandem.

4.4 Problem Definition

This chapter targets many-core processors with S-NUCA architecture, operating in an open system [48], where new applications arrive in an unpredictable order at a priori unknown times. We employ the one-thread-per-core model well suited for many-core processors [40], as introduced in Section 1.1. We further assume rigid applications whose threads cannot be migrated during their execution. The objective is to maximize the performance without exceeding the thermal constraint T_{crit}. The thermal constraint is enforced by using per-core power budgets. This is achieved by application mapping and power budget reallocation between active cores. The mapping is agnostic of the applications, as the characteristics of an incoming application are not known before starting it. The only available information about applications is their parallelism level, i.e., required number of cores.

Section 4.5 first introduces a Pareto-optimal mapping based on an ILP. This is too computationally expensive to compute at run time and instead serves as a baseline for the run-time heuristic mapping presented in Section 4.6. Finally, Section 4.7 presents the run-time power budget reallocation algorithm.

4.5 Pareto-Optimal Application Mapping

Application-agnostic mapping of k threads to n cores has two conflicting optimization goals: maximizing the power budget by TSP (and thus enabling operation at higher V/f levels) and minimizing the maximum AMD (and thus reducing the LLC latency). As explained in Section 4.1.3, these two metrics cannot both be simultaneously optimal. Hence, the goal of this section is to find all Pareto-optimal mappings. We first present a 0-1 ILP to find the optimal mapping with respect to the power budget only, which is then extended to also consider the LLC latency, and thus, find the Pareto-optimal mappings.

Only Power Budget The ILP formulation is based on the thermal model introduced in Section 3.1. We consider the steady-state temperatures, as described by Eq. (3.2). None of the steady-state core temperatures $T_{steady,i}$ may exceed the thermal constraint T_{crit} for thermally safe operation[1]. We additionally consider a global power budget P_{TDP}. Idle cores consume a constant power P_{idle}.

The optimization variables of the ILP are $\mathbf{x} = [x_i]_n$, $x_i \in \{0, 1\}$, where $x_i = 1$ indicates that core i is active (assigned to an application), whereas $x_i = 0$ means that core i is idle. A mapping and its associated power budget need to fulfill three constraints:

- Exactly k cores need to be selected.
- The global power budget P_{TDP} must not be exceeded.
- There must be no thermal violations in the steady-state, i.e., the per-core power budgets P_{TSP} must not be exceeded.

Selecting exactly k cores is satisfied if $\sum_{i=1}^{n} x_i = k$. For the remaining two constraints, we introduce a helper variable $H > 0$ that allows us to express them as linear inequalities. H is defined by

$$P_{TSP} = P_{idle} + \tfrac{1}{H} \tag{4.4}$$

The value of $\frac{1}{H}$ corresponds to the difference between the power budget of active cores and the power consumption of idle cores (static and leakage power). This value is the same for all active cores, because we employ uniform TSP for the mapping. The power budget is maximized if the helper variable H is minimized. The total power consumption of the chip comprises k active cores, each consuming at most the power budget P_{TSP}, and $n - k$ idle cores, each consuming the idle power P_{idle}. The total power consumption must not exceed the global power budget P_{TDP}:

$$k \cdot P_{TSP} + (n - k)P_{idle} \leq P_{TDP} \tag{4.5}$$

Combining Eqs. (4.4) and (4.5) and solving for H results in:

$$H \geq \frac{k}{P_{TDP} - n \cdot P_{idle}} \tag{4.6}$$

The steady-state is thermally safe if $\forall i : T_{steady,i} \leq T_{crit}$. Using Eqs. (3.2) and (4.4), we obtain:

$$T_{steady,i} = \sum_{j=1}^{N} B_{ij}^{-1} \cdot (P_{idle} + x_j \cdot \frac{1}{H} + T_{amb} \cdot G_j) \leq T_{crit} \tag{4.7}$$

Solving for H:

$$H \geq \frac{\sum_{j=1}^{n} B_{ij}^{-1} \cdot x_j}{T_{crit} - \sum_{j=1}^{n} B_{ij}^{-1} \cdot (P_{idle} + T_{amb} \cdot G_j)} \tag{4.8}$$

[1] While we consider the steady-state temperature in the optimization, we measure the transient temperature in the experimental evaluation in Section 4.8.1 and demonstrate few violations.

Putting all three constraints together, the power-budget-maximizing application mapping and its power budget is obtained by solving the following ILP:

$$\text{Minimize } H$$

such that:

$$\sum_{i=1}^{n} x_i = k \tag{4.9}$$

$$H \geq \frac{k}{P_{TDP} - n \cdot P_{idle}} \tag{4.10}$$

$$H \geq \frac{\sum_{j=1}^{n} B_{ij}^{-1} \cdot x_j}{T_{crit} - \sum_{j=1}^{n} B_{ij}^{-1} \cdot (P_{idle} + T_{amb} \cdot G_j)} \quad \forall i \in [1 \ldots n] \tag{4.11}$$

Power Budget and LLC Latency This ILP does not yet account for the non-uniform LLC latency in S-NUCA many-core processors. For a sufficiently high power budget, the performance of *PARSEC* applications is dominated by the core with the maximum AMD in this application's mapping since the thread pinned to this core forms a bottleneck [60]. Therefore, we use the maximum AMD of all cores that execute the threads of an application as a metric to optimize in the application mapping. Since the AMD values of the cores are discrete and symmetric, there exist only a small number of unique values [60]. For example, the 64-core many-core processor introduced in Section 3.1.2 has only 9 unique AMD values. This enables us to find all Pareto-optimal mappings by fixing the maximum AMD AMD_{max} in the mapping and maximizing the power budget for this value of AMD_{max}. Therefore, we add a fourth constraint to the ILP that restricts the available cores in to cores with a smaller AMD than the upper limit AMD_{max}.

$$x_i = 0 \quad \forall i : \text{AMD}(i) > AMD_{max} \tag{4.12}$$

The resulting ILP is solved several times, once for each unique AMD limit AMD_{max}. This process enables us to find all Pareto-optimal mappings.

Already Running Applications The above ILP implicitly makes the assumption that the many-core processor is idle when k new cores need to be assigned to an incoming application, which is in practice almost never the case. Let C_U denote the set of cores that are already assigned to other applications. The ILP needs to be modified slightly. Instead of finding a mapping of k threads, a mapping of $k + |C_U|$ threads is searched while enforcing that $x_i = 1$ if core $i \in C_U$. Fixing the already used cores accounts for application rigidity.

Solving this ILP is too computationally expensive for use at run time. For instance, determining the optimal mapping of 8 cores on a 64-core many-core processor takes 8.7 s using *MATLAB* on an *Intel Core i5-3470*. This is much higher than the average application execution time of 340 ms determined in the experimental evaluation of this chapter. Calculating the optimal mappings at design time and storing them in a lookup table (LUT) is not feasible either since there are too many possible scenarios of already

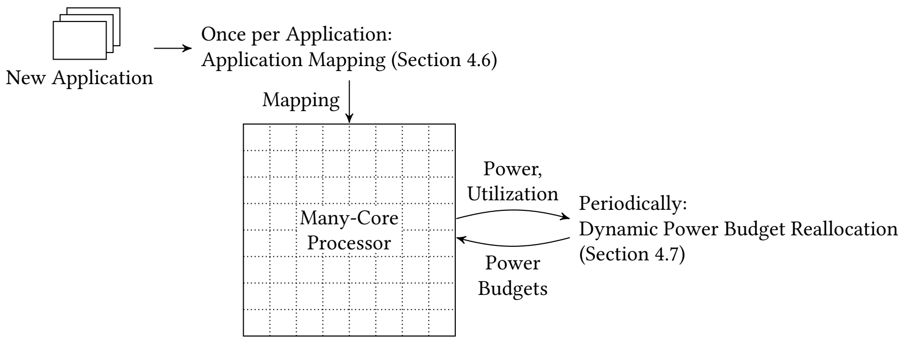


Figure 4.8: *PCGov* comprises two parts: application mapping and dynamic power budget reallocation. The application-agnostic mapping is only called once per application. The dynamic power budget reallocation periodically sets the power budgets according to the current power consumptions and core utilizations.

active threads and number of requested threads. This creates the need for an efficient heuristic run-time algorithm to find mappings close to the Pareto-optimal ones, so-called near-Pareto-optimal mappings. A near-Pareto-optimal has only an insignificantly lower power budget than the Pareto-optimal mapping with the same maximum AMD of the used cores. The heuristic algorithm is presented in the next section and its efficiency is evaluated empirically against the optimal solutions found by the ILP.

4.6 Run-Time Application Mapping Algorithm

This section presents the run-time application mapping algorithm. It forms the first part of the proposed algorithm *PCGov*, which is depicted in Fig. 4.8. Dynamic power budget reallocation forms the second part and is described in Section 4.7. The mapping algorithm needs to be application-agnostic. This is as the mapping is to be decided when a new application arrives, and, hence, before the application starts execution. Since we target unknown applications, no information except their number of threads is available. The mapping algorithm comprises two steps. The first step finds near-Pareto-optimal mappings. The second step selects one of these mappings that is expected to maximize the performance.

4.6.1 Find Near-Pareto-Optimal Mappings

Algorithm 4.1 presents the heuristic algorithm to find near-Pareto-optimal mapping candidates MC. Each mapping candidate should contain k cores. The set of cores that are already in use by other applications is denoted as C_U. The algorithm finds a near-Pareto-optimal mapping M for each unique AMD value AMD_{max} where only cores that do not exceed AMD_{max} are used. The k cores of each mapping M are selected *greedily* starting with an empty mapping. The available cores are the idle cores that do not exceed the AMD limit. For each of these cores, the resulting power budget is calculated, one at a time,

Algorithm 4.1 Find Near-Pareto-Optimal Mapping Candidates

```
$MC \leftarrow \emptyset$
for all $AMD_{max} \in$ UNIQUE($AMD(i) \mid i \in [1 \dots n]$) do
    $M \leftarrow \emptyset$                                              ▷ start with empty mapping
    while $|M| < k$ do                                                ▷ greedily select $k$ cores
        $A \leftarrow \{i \mid \text{AMD}(i) \leq AMD_{max}\} \setminus M \setminus C_U$                    ▷ available cores
        $P_{TSP,all} \leftarrow \{\text{TSP}(M \cup \{i\} \cup C_U) \mid i \in A\}$
        $M \leftarrow M \cup \{\arg\max_i \; P_{TSP,all}(i)\}$                ▷ greedily add a core to $M$
    $MC \leftarrow MC \cup \{M\}$
return $MC$
```

in combination with the already selected cores. The core that results in the highest power budget is added to the mapping M. This is repeated until k cores have been selected.

Calculating TSP for all available cores can be done in $\mathcal{O}(n^2)$ by sharing intermediate results across the individual calculations. Since this is to be done k times for each of the $\mathcal{O}(n)$ unique AMD values, the total complexity for finding all these near-Pareto-optimal mappings is $\mathcal{O}(k \cdot n^3)$.

4.6.2 Select One of the Near-Pareto-Optimal Mappings

After the near-Pareto-optimal mappings have been determined, one of them needs to be selected. The impact of power budget and LLC latency on the performance differs for different applications as shown in Fig. 4.4d. Selecting the mapping that maximizes the performance for an application can only be accomplished with detailed profiling, which is not available when the application arrives. Hence, the goal of the application-agnostic algorithm *PCGov* is to select the mapping that is expected to result in high performance for *most* applications.

Fig. 4.9 shows the Pareto-curve for mapping twelve threads on a 64-core many-core processor. The highest performance of *blackscholes* or *streamcluster* is obtained with neither the mapping with highest power budget nor the one with lowest LLC latency, but with mappings that perform a trade-off between both metrics. Fig. 4.4d shows that similar observations hold true for almost all other *PARSEC* applications. We use a weighted sum of power budget TSP(M) and maximum AMD of the selected cores $\max_{i \in M}$ AMD(i) to decide the selected mapping $M^\star$.

$$M^\star = \underset{M \in MC}{\arg\max} \left(\text{TSP}(M) - \alpha \cdot \max_{i \in M} \text{AMD}(i) \right) \tag{4.13}$$

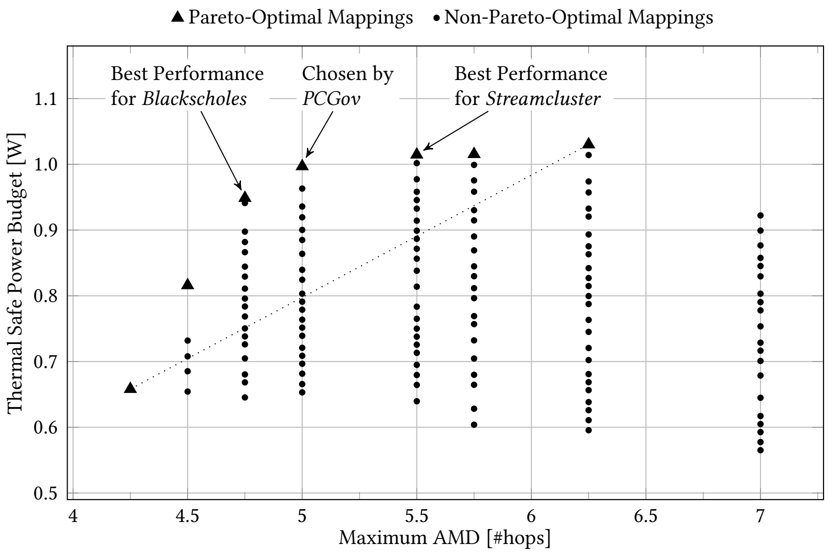


Figure 4.9: Pareto curve of power budgets and maximum AMDs of different mappings of twelve threads to a 64-core many-core processor. The best performance for *blackscholes* and *streamcluster* is attained with mappings that perform a trade-off between power budget and AMD. Our algorithm *PCGov*, which is designed to be application-agnostic, selects the mapping that equally balances power budget and AMD.

The parameter α balances the impact of power budget and AMD on the selection. Without application profiling, we aim to make a balanced trade-off between power budget and LLC latency. We calculate α for each set of mapping candidates separately:

$$\alpha = \frac{\max_{M \in MC} \text{TSP}(M) - \min_{M \in MC} \text{TSP}(M)}{\max_{M \in MC} \max_{i \in M} \text{AMD}(i) - \min_{M \in MC} \max_{i \in M} \text{AMD}(i)} \tag{4.14}$$

This equation makes the selection independent of the scales that are used. Thereby, *PCGov* selects the mapping with the highest distance to the dotted line in Fig. 4.9 that connects the mapping with the highest power budget and the mapping with the lowest maximum AMD. Selecting one of the mapping candidates has a time complexity of $O(n)$, which results in the total complexity $O(k \cdot n^3)$ for application mapping. This algorithm is to be run only once whenever a new application arrives.

4.7 Run-Time Dynamic Power Budget Reallocation

This section describes the run-time power budget reallocation algorithm, which forms the second part of *PCGov*. It observes the current core utilization and power consumption values of threads and adjusts their power budgets accordingly.

Fig. 4.10 shows the finite state machine (FSM) that is associated with each thread. It comprises three states based on the core utilization and power consumption: *Idle*, *Compute-*

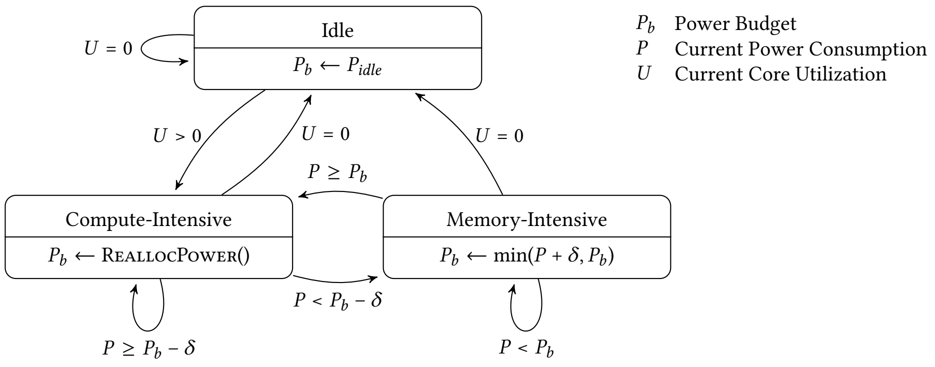

Idle
$P_b \leftarrow P_{idle}$
$U = 0$
$U > 0$
$U = 0$
$U = 0$
$P \geq P_b$
Compute-Intensive
$P_b \leftarrow$ ReallocPower()
Memory-Intensive
$P_b \leftarrow \min(P + \delta, P_b)$
$P < P_b - \delta$
$P \geq P_b - \delta$
$P < P_b$
P_b Power Budget
P Current Power Consumption
U Current Core Utilization

Figure 4.10: FSM to track the state of a thread and reallocate power budgets. Each state has its own policy to determine the power budget.

Intensive, and *Memory-Intensive*. A thread stays in the *Compute-Intensive* state as long as its power consumption potentially could exceed the power budget if the core ran at a higher V/f level. The DVFS control loop throttles the core to prevent a power budget violation. Hence, the observed power consumption P will be slightly lower than the power budget P_b. If a thread cannot exceed its associated power budget, i.e., if its power consumption is significantly lower than its power budget, the FSM transitions to the *Memory-Intensive* state. The parameter δ prevents oscillations between states by compensating small fluctuations in the power consumption. A thread transitions back to the *Compute-Intensive* state if its power consumption increases again. If a thread is idle, i.e., its core utilization U is zero, the FSM transitions to the *Idle* state.

The policy to determine the power budget depends on the current state. Idle threads are blocked and wait for an event before resuming operation. Their power budget is set to the bare minimum P_{idle}, which forces the DVFS control loop to choose the lowest possible V/f level for the associated core. This results in the highest possible surplus power budget available for other threads. The power consumption of memory-intensive threads does not reach their power budget. Hence, their power budget can safely be reduced to a value slightly higher than their current power consumption to free the otherwise wasted power budget. *This does not degrade their performance even though their power budget is reduced*, but allows to reallocate the otherwise wasted power budget to threads that can make use of it. Compute-intensive threads are the threads whose performance could increase with a higher power budget. Hence, their power budget should be maximized.

It is important to notice that power budget that was freed on any thread cannot be simply added to the power budget of another thread. Care must be taken to prevent thermal violations. *Therefore, the sum of all per-core power budgets may change if power is reallocated to ensure thermal safety.* Algorithm 4.2 increases the power budget for compute-intensive threads in a thermal-aware fashion. First, the set C_C of cores that receive power budget is determined. These are all cores that execute *compute-intensive* threads. Then, the

Algorithm 4.2 REALLOCPOWER: Reallocate Power Budgets

```
$C_{I+M} \leftarrow \{i \mid$ Core $i$ is *idle* or runs a *memory-intensive* thread$\}$
$C_C \leftarrow \{i \mid$ Core $i$ runs a *compute-intensive* thread$\}$
**for all** $i \in C_{i+M}$ **do**
    $P_{b,I+M}(i) \leftarrow P_b(i)$                    ▷ Read power budgets from FSMs
$P_C \leftarrow \mathrm{TSP}(C_C \mid P_{b,I+M})$  ▷ Comp. TSP for cores $C_C$ given power budget of other cores
**for all** $i \in C$ **do**
    $P_b(i) \leftarrow P_C$ ▷ Broadcaset power budgets of *compute-intensive* threads to corr. FSMs
```

thermally safe power budget for these cores is recalculated using TSP while considering the already determined power budget for the other cores $P_{b,I+M}$. TSP supports thermal nodes with known power consumption. To use this, we temporarily assume that the power consumptions of idle and memory-intensive threads exactly match their power budgets and, therefore, overestimate their power consumption at most by δ.

The FSMs of the threads are mostly independent, which allows for a distributed implementation. One FSM instance is run on every core. These FSMs independently track the state of their associated threads. However, the calculation of power budgets for compute-intensive threads needs information exchange between the FSMs. To do this, all FSMs in the *Idle*- and *Memory-Intensive*-state send their current power budget to a central manager that calculates the power budget for *compute-intensive* threads applying Algorithm 4.2 and broadcasts it to all associated FSMs.

Each FSM of one thread has a time complexity $O(1)$. Since TSP is to be recalculated after reallocating the power budget, Algorithm 4.2 has a complexity of $O(n^2)$. However, Algorithm 4.2 does not need to be run in every DVFS-epoch. This is only needed if the number of threads in the system changes or if the state of the FSM of a thread changes.

4.8 Experimental Evaluation

The experimental evaluation uses the setup described in Section 3.1.2 with slight modifications. The many-core processor is modeled to be fabricated in the 10 nm technology. The ambient temperature is 45 °C and the thermal constraint T_{crit} is 80 °C. TDP is set at 30 W. The power consumption of an idle core is set at 0.2 W due to the use of its associated LLC bank even when the core itself is idle. The parameter δ is set at 50 mW. We create and execute 3 random workloads that each consist of 20 randomly selected *PARSEC* applications with *simsmall* inputs. We use 8 of the 13 *PARSEC* applications: *blackscholes*, *bodytrack*, *canneal*, *dedup*, *fluidanimate*, *streamcluster*, *swaptions* and *x264*. The application arrival rates follow a Poisson-distribution with varying average arrival rate to trigger different system utilization values.

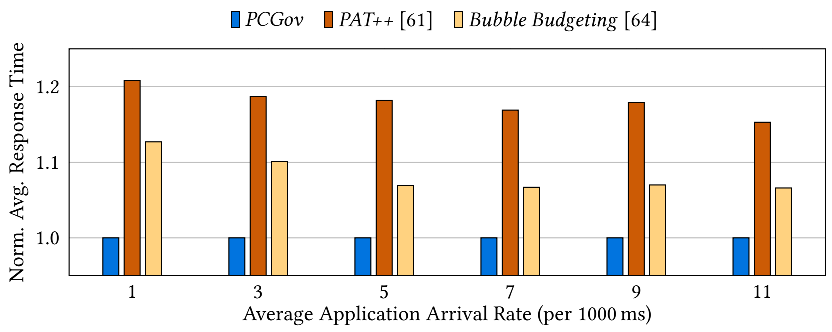
4.8.1 Comparison to the State of the Art

The proposed run-time application mapping and power budgeting algorithm *PCGov* is compared to the state-of-the-art algorithms *Bubble Budgeting* [64], and *PAT++* [61], since these algorithms address the same problem. This is the main experiment in this chapter.

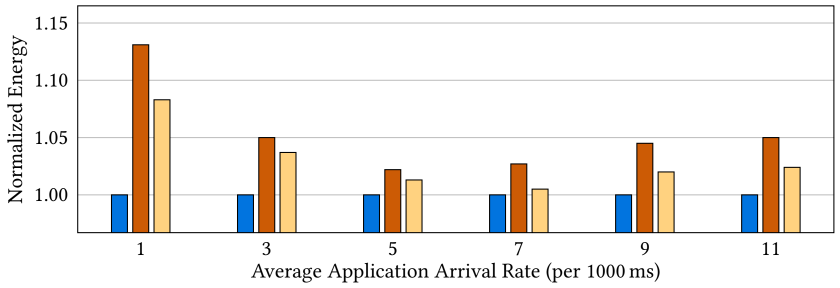
Bubble Budgeting [64] is a run-time technique that aims to maximize performance under the constraint of thermal safety, using application mapping and DVFS. The goal of the mapping policy is to map threads of an application spatially compactly with idle cores (*bubbles*) in between. The idle cores are used to dissipate heat to prevent thermal hotspots. Threads are assumed to communicate point-to-point, where compactness reduces the communication latency. The mapping is extended by a power budgeting algorithm that uses a heuristic to determine the V/f levels of all threads and refines this heuristic in a second step. To do this, the power consumption of threads at different V/f levels need to be known in advance, which is not the case in open systems, as we study here.

PAT++ [61] is a run-time application mapping and power budgeting technique with similar goals and application models as *Bubble Budgeting*. Authors propose a heuristic algorithm to sparsely map threads of an application within a small region of the core. The motivation is similar to the previous approach, where idle cores dissipate heat but increase the point-to-point communication latency between threads. *PAT++* further tries to maximize the distance between threads of different applications to minimize the mutual thermal impact of applications. This application mapping algorithm is extended by both a power budgeting controller and a boosting controller. The power budget is calculated using *pessimistic* TSP, which only considers the number of active cores instead of the full mapping. The power budgeting controller reallocates surplus power budget according to applications' priorities and network characteristics. Remaining thermal margin is exploited by the boosting controller that allows threads to exceed their power budget for short periods of time. The boosting technique proposed in [61] determines the application to boost based on the current temperatures of the cores they are running on, i.e., boost the coldest core. Both *Bubble Budgeting* and *PAT++* neglect that different applications have different characteristics which largely affect the benefits of boosting. We adapted *Bubble Budgeting* and *PAT++* to the used platform.

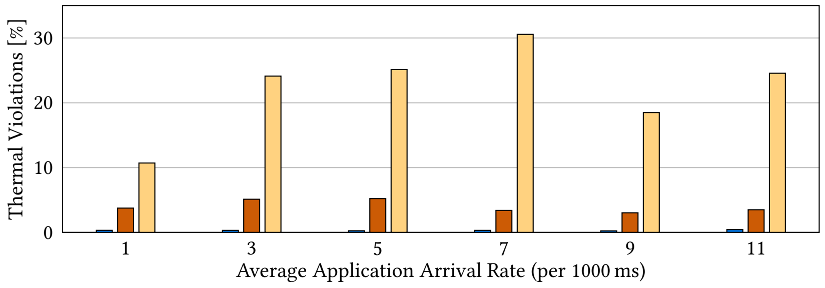
Fig. 4.11a compares the average response time of the applications for different arrival rates for each of the three application mapping and DVFS policies. In these experiments, the average (peak) system utilization varies from 4.3 % (34 %) for an arrival rate of 1 per 1000 ms up to 34 % (91 %) for an arrival rate of 11 per 1000 ms. The response times of individual applications vary significantly from 24 ms up to 2.3 s. In order to prevent long-running applications from dominating the average response time, we compare the geometric means of the response times. *PCGov* always achieves the lowest response time and, hence, in the highest performance with up to 21 % and 13 % improvement over *PAT++* and *Bubble Budgeting*, respectively. This is because *PCGov* considers not only the power budget but also the LLC latency. *PAT++* and *Bubble Budgeting* consider power budget and application communication but assume message passing, which is not the case in S-NUCA many-core processors. Hence, these approaches tend to map applications to the



(a) Performance



(b) Energy



(c) Temperature

Figure 4.11: Comparison of *PCGov*, *PAT++* [61] and *Bubble Budgeting* [64] policies for random multi-program workloads at different arrival rates. *PCGov* achieves a significantly lower response time, lower energy consumption, and fewer thermal violations than *PAT++* and *Bubble Budgeting*.

corners of the many-core processor and thus result in high LLC latencies and reduced performance. These observations hold true for varying arrival rates.

Fig. 4.11b compares the average energy consumption per application for execution of the workloads. The energy consumption shows similar trends as the performance. *PCGov*

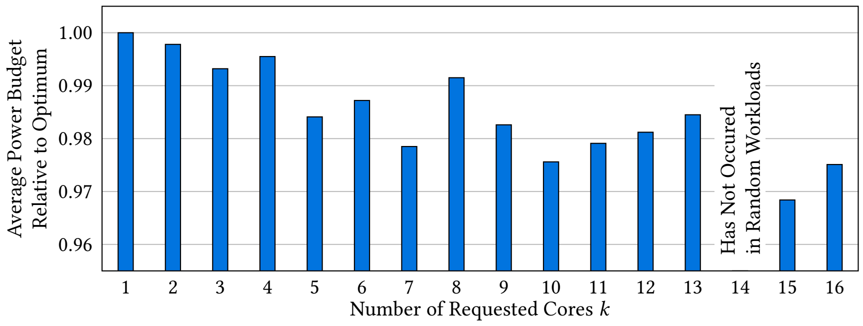


Figure 4.12: Average power budgets of the near-Pareto-optimal mapping candidates obtained by *PCGov* are very close to the Pareto-optimal mappings obtained by the ILP.

always achieves the lowest energy consumption for all arrival rates with up to 13 % and 8.3 % improvement over *PAT++* and *Bubble Budgeting*, respectively.

Fig. 4.11c shows how often thermal violations occur during execution, comparing the capabilities of the policies to prevent thermal violations. This mainly studies the effectiveness of the power budgeting and DVFS algorithm, not of the application mapping. The power budgeting policy of *PCGov* is designed to prevent thermal violations in the steady state by considering the actual mapping. The resulting power budget is enforced on a per-core basis. This results in very rare thermal violations in the transient temperature of less than 0.4 % of the time. Since we employ a reactive approach, thermal violations due to sudden changes in the power consumption of threads cannot be fully avoided. *PAT++* uses a per-chip power budget, which cannot prevent thermal hotspots. Hence, the duration of thermal violations is much higher (up to 5.2 %). The thermal model used by *Bubble Budgeting* determines the power budget of a core by considering its local neighborhood. However, the model does not consider the location of the core on the chip. The cores in the corners of the many-core processor have a significantly lower power budget compared to other cores since they have fewer neighbors that can dissipate heat. *Bubble Budgeting* cannot consider this and, hence, causes frequent thermal violations (up to 31 %), mostly in the corner cores.

Summarizing, *PCGov* achieves substantial improvements over both *PAT++* [61] and *Bubble Budgeting* [64]. It significantly reduces response time, energy consumption, and thermal violations.

4.8.2 Evaluation of the Mapping Candidates of *PCGov*

The first step of the mapping algorithm of *PCGov* finds near-Pareto-optimal mappings. We evaluate how close these mappings are to the Pareto-optimal mappings by extracting all near-Pareto-optimal mappings that were obtained while executing the random workloads. These are all the mapping candidates *MC*, not just the selected mappings. We compare the power budget of each of these mappings to the power budget of the corresponding

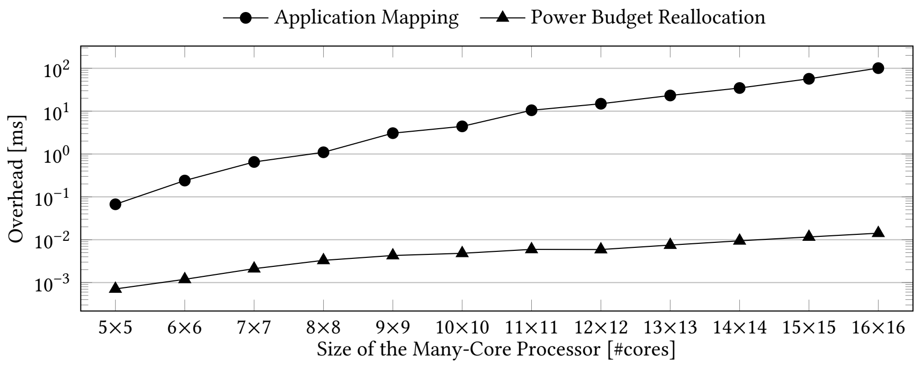


Figure 4.13: Average overhead per execution for application mapping and dynamic power budget reallocation depending on the size of the many-core processor.

Pareto-optimal mapping with the same maximum AMD. The Pareto-optimal mappings are obtained using the ILP described in Section 4.5. Fig. 4.12 shows the results per each number of requested cores k. Requesting $k = 14$ cores did not occur in the random workloads. For small values of k, the mappings obtained by the greedy algorithm are very close to optimal or even optimal. For larger values, the quality drops slightly. This is because the search space increases combinatorially with the number of requested cores k, and, hence, the likelihood for a greedy algorithm to find the optimal solution decreases. Overall, the greedy algorithm determines near-Pareto-optimal mappings with a negligible loss in power budget of less than 3.2 % compared to the Pareto-optimal mappings, at an overhead that is several orders of magnitude lower, as will be shown in the next section.

4.8.3 Run-Time Overhead

The run-time overhead of *PCGov* is evaluated w.r.t. two aspects, which are the impact on the performance, and the impact on power and temperature.

Performance Impact Fig. 4.13 shows the average run-time overhead of *PCGov*. The overhead is split into two components: application mapping and power budget reallocation. Application mapping is done only once per application, while power budget reallocation is done periodically. The overhead is reported per execution of the algorithms, i.e., per application mapping and per power budget reallocation epoch, respectively.

The overhead of application mapping is measured by creating random scenarios, where the number and location of already active cores is set randomly at a utilization of 50 % and a random number of requested cores k is selected between 1 and 16. The average overhead increases with the number of cores from 67 µs on a 5×5 many-core processor up to 101 ms on a 16×16 many-core processor when running on a single core. It is important to note that mapping is only performed once per application. The average execution time of applications in the experiments in Section 4.8.1 is 340 ms. This results in a relative overhead of 30 % for application mapping on a 16×16 many-core processor, which may be impractical. However, up to 10×10 cores, the overhead is less than 1.3 %. We conclude that *PCGov* is suitable for many-core processors with up to 10×10 cores, but may be too slow for larger processors. On the studied 8×8 many-core processor, application mapping takes 1 ms on average which results in an overhead of 0.3 %.

When measuring the overhead of the power budget reallocation, it needs to be considered that Algorithm 4.2, which calculates the power budget for *compute-intensive* applications, does not need to be executed in every epoch. The less the power consumption of active applications fluctuates, the less often this algorithm is executed. However, this depends on the executed workloads. To account for this, we extract power and utilization traces from the workloads executed in Section 4.8.1 to stimulate the power budgeting algorithm and measure its execution time. Algorithm 4.2 had to be executed in less than 10 % of all power budgeting epochs. The overhead increases with the size of the many-core processor from 0.7 µs on a 5×5 many-core processor up to 14.2 µs on a 16×16 many-core processor. Considering the power budget reallocation epoch of 100 µs, this results in a relative overhead of less than 0.1 % for all studied sizes of the many-core processor, since Algorithm 4.2 is executed on only one of the cores. The average overhead on the studied 8×8 many-core processor is 3.3 µs.

Power and Thermal Impact *PCGov* is executed at run time, i.e., in parallel to the executed workload. Therefore, we need to also investigate its impact on power and temperature. Executing application mapping (Algorithm 4.1 and selection of the best candidate) on a single core results in a power consumption of up to 1.4 W. This is comparable to the power consumption of actual applications, which ranges from 1.1 W to 2.8 W, as has been analyzed in Fig. 4.6. Therefore, it needs to be considered by the dynamic power budget reallocation algorithm to maintain thermal safety. However, since the power budget algorithm is agnostic of applications and simply considers threads independently, no changes to the power budget reallocation algorithm are required. The thread that decides the mapping can be treated exactly as a thread of any other application.

The power budget reallocation algorithm is executed periodically and, therefore, also runs in parallel to the executed applications. As discussed earlier, the average execution time of this algorithm is 3.3 µs every 100 µs on a single core. The peak power consumption during its execution is 1.7 W. Since the control epoch is small compared to thermal time constants of a processor cooling system, the thermal impact can be treated as constant and we only need to consider the average power consumption over the full control epoch, i.e., including 96.7 µs idle time. The average power consumption is 0.25 W, which is very close to the idle power consumption of 0.2 W. Therefore, the power budget reallocation algorithm has a negligible thermal impact on the many-core processor.

4.8.4 Impact of the Hysteresis Parameter δ

The goal of employing a hysteresis parameter δ to the FSM is to reduce the run-time overhead of the power budget reallocation. Without δ, i.e., $\delta = 0$ mW, the power budget

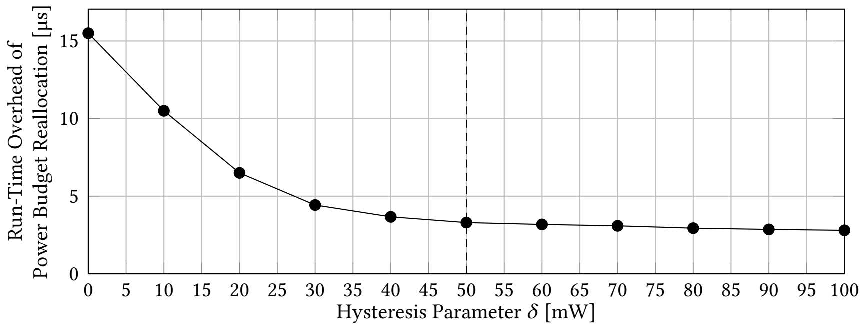


Figure 4.14: Impact of varying the hysteresis parameter δ on the performance and dynamic power budget reallocation overhead. Setting δ at 50 mW allows to filter most of the noise in the power consumption.

of compute-intensive applications would have to be recalculated every time the power consumption of a memory-intensive application changes. This is the case in every control step due to noisy application and platform behavior leading to small fluctuations in the power consumption. The hysteresis δ avoids this by creating a corridor for the power consumption. As long as the power consumption stays within this corridor, the power budgets do not have to be recalculated. The wasted power budget, i.e., the difference between power budget and actual power consumption, is bounded by δ. Summarizing, δ should be high enough to filter fluctuations in the power consumption, but as low as possible to bound the wasted power budget.

Fig. 4.14 shows the average overhead of power budget reallocation for different values of the hysteresis δ. The overhead is highest for $\delta = 0$ mW where it is 16 µs per control epoch when executed on a single core. The overhead reduces strongly with increasing δ, but only slightly decreases above 50 mW. Therefore, we choose $\delta = 50$ mW because this value allows to filter out fluctuations in the power consumption, but still is small compared to the absolute power consumption of applications of up to 2.8 W as shown in Fig. 4.6.

4.9 Summary

This chapter presented a run-time application mapping and power budgeting algorithm called *PCGov*. This algorithm exploits a trade-off between power budget and LLC latency for application mapping on many-core processors with distributed shared LLC. It is based on the observation that optimizing for power budget and optimizing for LLC latency result in contradictory mapping decisions. Optimizing for only one of the two does not provide the best application performance. Instead, a trade-off between the two is most beneficial. Furthermore, *PCGov* exploits heterogeneity in the power consumption both within threads of the same application and between threads of different applications. Experimental evaluation with random multi-program workloads shows that *PCGov* is very effective in avoiding thermal violations while maximizing the performance. Furthermore, the overhead of *PCGov* on a 64-core many-core processor is negligible.

The employed algorithms use simple heuristics that ignore relevant application characteristics. The application mapping ignores any application characteristics and instead aims at achieving a good performance on average. However, different applications depend differently on the power budget and LLC latency, leading to a different trade-off per application. This results in different optimal mappings, as can be seen in Figs. 4.4 and 4.9. The optimal mapping can only be determined by considering the application characteristics. In addition, different cooling would shift the trade-off, rendering the heuristic selection in Eq. (4.14) suboptimal. In the extreme case, a very strong cooling would result in thermally-safe operation with all mappings, rendering the heuristic ineffective. Solving this would require additionally considering the cooling in the mapping heuristic. While the power budget reallocation heuristic considers some characteristics of the applications, i.e., their power consumption, it only reallocates unused power budget. More optimization would be feasible when throttling memory-intensive applications to boost other applications even more. However, such optimization would again depend on application characteristics, such as the sensitivity of power and performance on the V/f levels.

The general challenge with classical heuristic resource management is that it can only consider a limited amount of information. Moreover, the used information must be directly observable. This is generally not the case with application characteristics. Application characteristics contain many dimensions, such as instruction mix or intensiveness on the different cache levels, interconnect, DRAM, etc. Abstract characteristics like memory-intensiveness are only indirectly observable via several hardware performance counters. Therefore, classical heuristic management is not capable to achieve near-optimal management. This requires detailed modeling of the platform and applications, which can either be done with analytical models or ML models. The following sections present techniques that tackle the shortcomings of classical heuristic management.

5 Prediction-Based Application Migration

This chapter presents ML-driven prediction-based application migration to maximize the performance on thermally-constrained S-NUCA many-core processors. The presented technique *PCMig* employs the pattern to predict the impact of a migration before executing it, as introduced in Section 1.4.1. Employing an ML performance prediction model to dynamically adjust the mapping according to the observations of the application characteristics at run time enables coping with the limitations of the classical heuristic *PCGov* introduced in the previous chapter, which ignores relevant application characteristics. This chapter also develops analytical modeling based on CPI stacks, similar to the works in [70, 71, 130], as a comparison to ML-based modeling.

5.1 Motivational Examples

This section presents three motivational examples, demonstrating the relevance of application characteristics, system load changes, and execution phases to find the optimal application migrations.

Application Characteristics As discussed in Section 4.1, application mapping on S-NUCA many-core processors needs to make a trade-off between the power budget, which affects the sustainable V/f levels, and the AMD of the cores to the LLC banks, which affects the average LLC latency. The performance-maximizing trade-off between the power budget and the AMD depends on the executed thread's characteristics. Fig. 5.1 shows the performance of *PARSEC blackscholes* master and slave threads at different AMD and power budget values. The performance of both threads depends on both factors. However, their characteristics differ. Lines *a* in Fig. 5.1 show their dependency on the AMD. With increasing AMD, the master thread's performance significantly drops, whereas the slave threads' performance is unaffected. The master thread is more memory-intensive and performs more LLC accesses per instruction, whereas the slave threads are more compute-intensive. This results in the master thread's higher sensitivity to the AMD. In contrast, Lines *b* in Fig. 5.1 show the impact of the power budget. The master thread's performance

This chapter is mainly based on [6, 8].

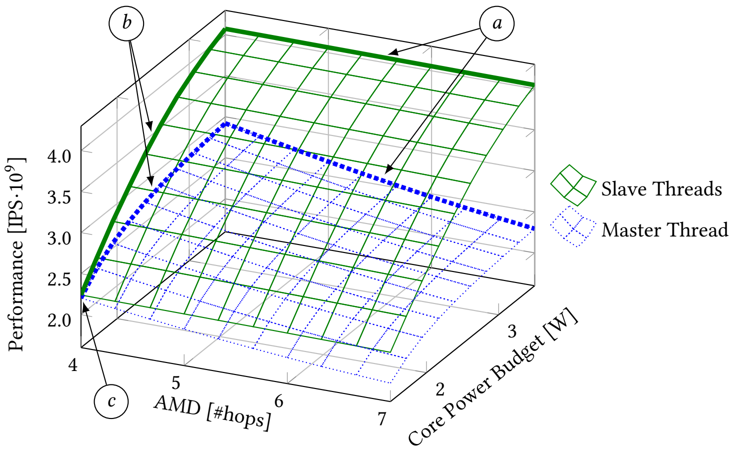


Figure 5.1: The performances of *blackscholes* master and slave threads depend differently on the power budget and AMD due to their different characteristics.

saturates early and increasing the power budget has little benefit, whereas the slave threads can exploit a high power budget and benefits strongly from a higher power budget. The master thread is memory-intensive, and, therefore, its performance does not depend as strongly on the V/f level, and additionally, its power consumption is lower which renders high power budgets not beneficial, as the peak V/f levels are reached early. The slave threads, on the other hand, are compute-intensive. High V/f levels strongly increase the performance and the power consumption is high, which makes a high power budget beneficial because it enables operation at higher V/f levels. *This examples shows that thread characteristics must be considered to determine the performance-maximizing trade-off between power budget and LLC latency.*

However, thread characteristics are not available until the application has started execution. Therefore, initial application mapping cannot consider these characteristics, and, hence, cannot always select the optimal mapping. Application migration is required after the application has started to adjust the mapping accordingly.

System Load Changes The application arrivals and departures, as well as the applications themselves, are not known in advance in open systems [48]. The resulting dynamic workload changes require readjustments of the mapping using application migration to maintain peak performance. Fig. 5.2 shows an example to illustrate this. *Canneal*, which is memory-intensive, is mapped to cores close to the center of the many-core processor to minimize the LLC latency, which is the optimal mapping for it. During its execution, an instance of *bodytrack* arrives and is mapped in a way that trades off power budget and LLC latency, which is also the optimal mapping w.r.t. the already mapped *canneal*. After *canneal* terminates, migrating *bodytrack* to now free cores closer to the center of the many-core processor improves its performance by 4 % over a rigid mapping. Mappings

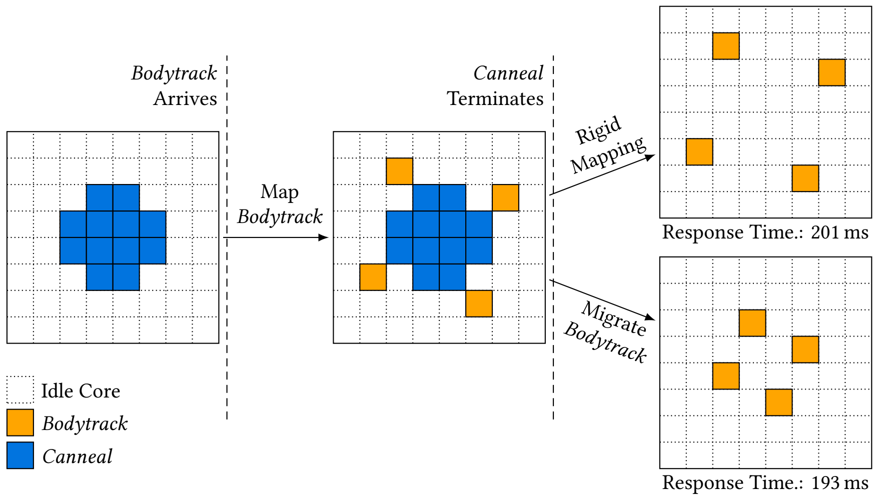


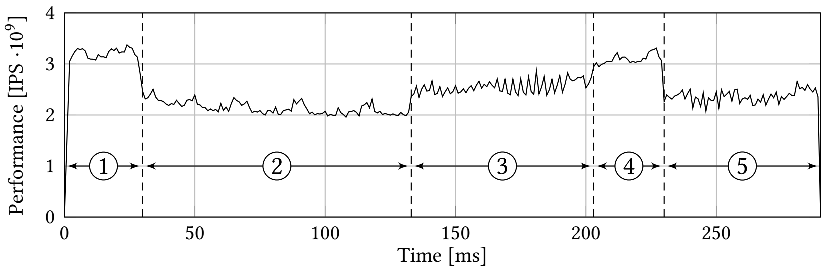
Figure 5.2: Application migration after a system load change, e.g., due to a terminating application, improves the performance.

that have been optimal once, can become suboptimal when other applications finish. Similar observations hold true for new arriving applications. *This example shows that system load changes require adjusting the mapping via migration.*

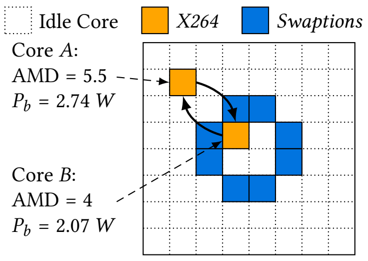
Application Execution Phases As discussed in the first motivational example, different threads have different characteristics. Furthermore, even the characteristics of a single thread may change over time with its execution phases. Fig. 5.3 presents an example in which two applications are executed on the many-core processor. *Swaptions* is mapped to cores close to the center but with a gap in the center to prevent a thermal hotspot. We consider two mappings for single-threaded *x264* as depicted in Fig. 5.3b. Core *A* is far from *swaptions* and, therefore, has a high power budget but also has a high AMD. Core *B* is in the center of the many-core processor and has minimal AMD but has a lower power budget due to the proximity to *swaptions*. Fig. 5.3a shows that *x264* has several distinct execution phases. These phases come from different frames being encoded, resulting in different computational and memory access patterns. Fig. 5.3c shows the performance per phase with each mapping. The performance-maximizing mapping alternates between Core *A* and Core *B*. Operating *x264* at its best mapping improves its per-phase performance by up to 14 %. *This example shows that the peak performance can only be achieved by dynamically adjusting the application mapping to their changing characteristics.*

5.2 Challenges and Novel Contributions

There are two main challenges in performance maximization on thermally-constrained S-NUCA many-core processors. As has been demonstrated in the previous motivational



(a) Execution phases of *x264* (if running on Core *A*)



(b) Mappings of the applications

Phase	1	2	3	4	5
Core *A*: IPS $\cdot 10^9$	**3.22** (+9.6 %)	2.13	2.54	**3.08** (+6.1 %)	2.36
Core *B*: IPS $\cdot 10^9$	2.94	**2.43** (+13.7 %)	**2.64** (+4.1 %)	2.90	**2.53** (+7.0 %)

(c) Average IPS of the phases under different mappings

Figure 5.3: Migration *x264* according to its execution phases increases the per-phase performance by up to 14 %.

examples, this goal can only be achieved through application migration. The first challenge is the inherent complexity in the performance of applications. The performance of a thread depends on its power budget (which depends on the cooling and thermal constraint), on the AMD of the core that it is running on, and on the characteristics of the thread, i.e., its compute- and memory-intensiveness. The second challenge is that proactive management is required because it is not clear in advance whether a migration would increase or decrease the performance. In addition, there are many possible migrations at any point in time, as any thread may be migrated to any free core.

These challenges are tackled with an ML-based prediction model that predicts the impact of a migration. The novel contributions of this chapter are as follows:

- The work presented in this chapter is the first to explore the performance potential of thread migrations on a many-core processor with distributed shared LLC.
- An analytical and an NN model are created to predict the power budget and LLC-based performance impact of thread migrations. Both models are compared w.r.t. accuracy and overhead, demonstrating the superiority of the NN-based model.
- A lightweight run-time application migration algorithm *PCMig* is proposed that uses the performance-prediction model to improve the many-core processor's overall performance.

5.3 Problem Definition

As demonstrated in the motivational examples, the performance of a thermally-constrained S-NUCA many-core processor can only be maximized through application migration. The challenge is how to decide which thread to migrate at which point in time to which core. Migration changes the AMD of the migrated thread. Additionally, the change in the mapping changes the thermal state and, therefore, also changes the thermally-safe power budgets of all threads. Both AMD and power budget ultimately affect the performance of the threads.

This chapter follows the pattern to predict the impact of a migration before executing it, as introduced in Section 1.4.1, in order to assess whether a certain migration candidate should be executed. With accurate predictions, selecting the best migration is straightforward. As demonstrated in Section 5.1, the impacts of AMD and power budget on the performance of a thread heavily depend on its characteristics.

Problem definition for performance prediction The goal of performance prediction is to estimate the IPS of a thread after a migration of this thread or another thread. IPS may be a misleading metric in multi-threaded workloads [131], e.g., due to the presence of spinlocks. In this case, other metrics like *application heartbeats* [132] could be employed instead that do not suffer from such inaccuracies but may require changes to the applications. However, for our studied applications, IPS strongly correlates with the actual performance, which justifies using it as a proxy variable to estimate the performance impact of application migrations. It is important to mention already here that the evaluation is not performed based on the IPS of applications, but by measuring the response time of applications. The migration is characterized by the initial AMD and power budget, and the AMD and power budget after the migration. The thread is characterized by its current cycle stack. As explained in Section 3.1, the cycle stack can be obtained from run-time profiling information obtained by hardware performance counters. In summary, the following inputs are available for predicting the performance of a thread:

- AMD and power budget before migration
- AMD and power budget after migration
- Current cycle stack of the thread
- Current V/f level

We develop in this chapter several models for performance prediction on many-core processors with distributed shared LLC. First, we build NN models that directly predict the performance (IPS) of a thread after the migration. We train different models that make different trade-offs between accuracy and overhead. Secondly, we build an analytical model to serve as a comparison. We separately model the impact of AMD, the impact of V/f level, and power. Similar to the state of the art [70], we base the analytical model on the abstract concept of CPI stacks. Finally, we compare the overhead and prediction accuracy of the NN models and the analytical model.

5.4 Neural Network-based Prediction Model

This section describes how the NN-based IPS prediction models are created, including the feature selection, training data generation, and the NN topology.

5.4.1 Feature Selection

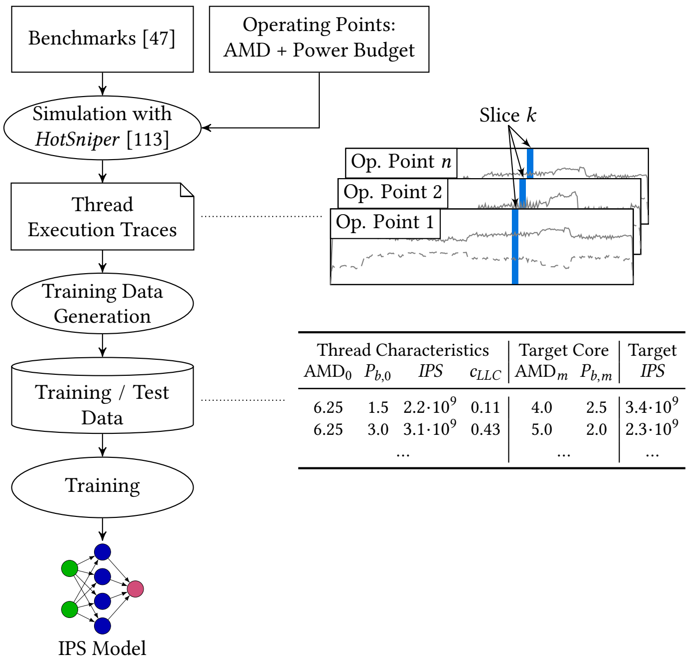
An important step in designing any ML model is the feature selection. We require features for the target core and the thread characteristics. As discussed in Section 4.1, cores in an S-NUCA many-core processor differ in their AMD and the thermally-safe power budget, which is determined by the mapping after migration. Therefore, we use these two values as features for the target core.

The thread is characterized by features obtained from its current CPI stack. Firstly, we use the current IPS before migration as a feature. This allows to capture that threads with a high IPS on the source core are more likely to have a high IPS on the target core. However, only IPS is not sufficient, as can be seen in Fig. 5.1. *Blackscholes* master and slave threads have similar performance (IPS) when executed at a center core (AMD = 4 hops) at a power budget of 1.5 W (Point *c* in Fig. 5.1). Migrating them to other cores with different AMD and power budget affects the two threads very differently. This is due to the different LLC access characteristics of the two threads. We capture such differences in the thread characteristics by adding $c_{LLC} := \mathrm{CPI}_{LLC} / \mathrm{CPI}_{total}$ as a feature. This feature captures the relative performance loss because of LLC accesses. A higher value corresponds to a higher susceptibility to AMD changes, and lower susceptibility to power budget changes. The c_{LLC} values of *blackscholes* master and slave threads at Point *c* in Fig. 5.1 are 0.49 and 0.09, respectively. The current IPS and c_{LLC} only gain expressiveness if the AMD and power budget of the current core that executes the thread are known. Therefore, we add the AMD and power budget of the current core as features. In total, we use the following six features for the NN model:

1. AMD of the current core AMD_0
2. Power budget of the current core (with the current mapping) $P_{b,0}$
3. AMD of the target core AMD_m
4. Power budget of the target core (with the mapping after the migration) $P_{b,m}$
5. Current IPS of the thread
6. Current c_{LLC} of the thread

5.4.2 Training Data Generation

Fig. 5.4 provides an overview of training and test data generation. We execute multi-threaded *PARSEC* applications at different *operating points* and record traces of their IPS and CPI stacks. Thereby, an operating point is a combination of an AMD value and a power budget value. We use 30 operating points, which are combinations of five different



Thread Characteristics				Target Core		Target
AMD_0	$P_{b,0}$	*IPS*	c_{LLC}	AMD_m	$P_{b,m}$	*IPS*
6.25	1.5	$2.2{\cdot}10^9$	0.11	4.0	2.5	$3.4{\cdot}10^9$
6.25	3.0	$3.1{\cdot}10^9$	0.43	5.0	2.0	$2.3{\cdot}10^9$
		…		…		…

Figure 5.4: Training data generation for the NN-based IPS prediction models.

mappings (five AMD values from 4 hops to 7 hops) and six power budgets between 1.0 W and 3.5 W. Thereby, we obtain 30 multi-threaded traces per application.

These traces may have different lengths, even for the same application, because the operating point affects the execution time of an application. We cut each trace into 100 slices and align the slices of different traces of the same application. Consequently, the k-th slices of different traces of the same application execute the same instructions. We create one training example from each pair of slices. This training example represents a hypothetical migration from the operating point of the first slice to the operating point of the second one. Thereby, up to $30 \cdot 29 = 870$ training examples are obtained from each slice of each thread of each application. We filter out slices when a thread is idle, resulting in 1,026,000 valid training examples in total.

5.4.3 Neural Network Topology

We use small fully-connected feedforward NNs with few hidden layers. Due to the smooth shapes of the functions to learn (see Fig. 5.1), we use a *sigmoid* activation function for hidden layers. The lowest prediction error is achieved with two hidden layers with 24 and 12 neurons, respectively. We use L2 regularization to improve the generalization and use the *Adam* optimizer [133] to improve the training convergence.

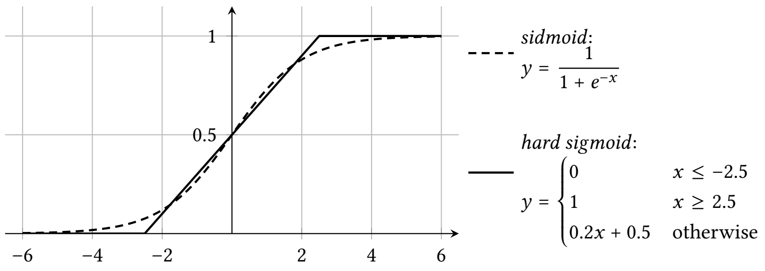


Figure 5.5: Comparison of the two activation functions *sigmoid* and an approximate version *fast sigmoid* that does not require calculating an expensive exponential term.

As discussed previously, the run-time overhead is an important factor to consider. Computing a *sigmoid* function requires calculating an exponential term, which is computationally expensive. We, therefore, also test the *hard sigmoid* function, depicted in Fig. 5.5. It is a piecewise linear variant of the *sigmoid* function, and has a much lower overhead [13]. We also tested to use only one hidden layer and tested reducing the number of neurons in the hidden layer. This greatly reduces the overhead but also reduces the accuracy. The results for accuracy and run-time overhead of different topologies are described in the evaluation of this chapter.

5.5 Analytical Prediction Model

Performance and power prediction based on CPI stacks is a common technique in the literature [70, 71, 130]. Therefore, we base our analytical performance prediction model on CPI stacks. We first separately investigate the impact of AMD and V/f level changes. Then, we build a power model to estimate the power consumption at a given V/f level and CPI stack. All these models are based on an analysis of the many-core processor's architecture to obtain formulas with parameters that are fitted using benchmark applications. Finally, we combine all three models.

The CPI stack comprises two classes of components. These are components related to CPU and L1 caches and components related to the NoC, LLC, and DRAM. CPU and L1 caches operate at the same V/f level, which can be changed using DVFS. Consequently, the same operation always takes the same number of CPU cycles, and related CPI stack components are independent of the AMD and the CPU V/f level. The CPI stack components of the other components are affected by the CPU V/f level and the mapping (AMD). Fig. 5.6 visualizes these observations using the *bodytrack* master thread.

5.5.1 Impact of the Last-Level Cache Access Latency

Only the LLC-related components of the CPI stack depend on the AMD. Calculations in the CPU and accesses to the L1 cache do not traverse the NoC. DRAM accesses originate in the LLC. Therefore, the average DRAM access latency (on top of the LLC latency) is

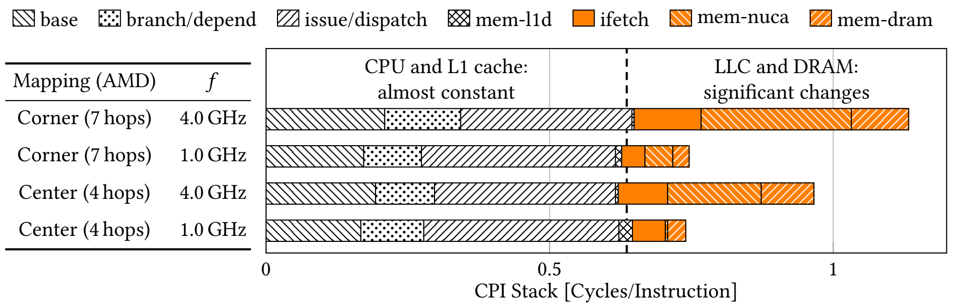


Figure 5.6: Impact of the AMD and V/f level on the CPI stack of the *bodytrack* master thread. The components related to the CPU and L1 cache are not affected by the AMD or V/f level. In contrast, the components related to LLC and DRAM change significantly.

independent of the mapping of the thread. The latency of a single LLC access comprises two parts. First, the memory access itself has a latency l_i^0. Second, each access to the LLC needs to traverse the NoC twice. As explained in Section 4.1, the average hop count to access the LLC from a core is characterized by its AMD. Let l_{hop} denote the latency of a single hop on the NoC. The average latency of an LLC access is described by:

$$l_i^{avg}(\text{AMD}) = l_i^0 + 2 \cdot \text{AMD} \cdot l_{hop} \tag{5.1}$$

If a sequence of N instructions contains m accesses to the LLC, the CPI stack component when operating at frequency f can be calculated as:

$$\begin{aligned} \text{CPI}_i(\text{AMD}, f) &= \frac{m \cdot f \cdot l_i^{avg}(\text{AMD})}{N} \qquad (5.2) \\ &= \frac{m \cdot f \cdot 2 \cdot l_{hop}}{N} \cdot \left(\frac{l_i^0}{2 \cdot l_{hop}} + \text{AMD} \right) \qquad (5.3) \end{aligned}$$

Let $\alpha_i := \frac{l_i^0}{2 \cdot l_{hop}}$ be one constant per LLC-related CPI stack component i. Migrating a thread from a core with AMD_1 to another core with AMD_2, both operating at the same frequency f does not affect m, N, and l_{hop}, and consequently changes CPI_i as follows:

$$\text{CPI}_i(\text{AMD}_2, f) = \frac{\alpha_i + \text{AMD}_2}{\alpha_i + \text{AMD}_1} \cdot \text{CPI}_i(\text{AMD}_1, f) \tag{5.4}$$

Solving Eq. (5.4) for α_i results in:

$$\alpha_i = \frac{\text{CPI}_i(\text{AMD}_1, f) \cdot \text{AMD}_2 - \text{CPI}_i(\text{AMD}_2, f) \cdot \text{AMD}_1}{\text{CPI}_i(\text{AMD}_2, f) - \text{CPI}_i(\text{AMD}_1, f)} \tag{5.5}$$

Eq. (5.5) can be used to calculate α_i from two traces of the same application at the same V/f level but different AMD. The parameters α_i are ultimately obtained by averaging over many pairs of traces of different benchmark applications. This is repeated for every LLC-related component of the CPI stack. It is important to notice that Eqs. (5.4) and (5.5) implicitly consider shared resource contention because these effects are already considered in the CPI stacks.

Memory-level parallelism (MLP) is a relevant parameter when estimating the performance of out-of-order cores [70]. In our case, the microarchitectures of source and target core are identical, the only architectural difference is the average LLC latency. By working on the CPI stack, MLP is already considered. For instance, the performance of a workload with high MLP only weakly depends on the LLC latency. Such a workload has small CPI stack components for the caches and memory, and, therefore, predicted changes in the CPI stack by our model are also small.

5.5.2 Impact of the Voltage/Frequency Level

The time spent waiting for the LLC and DRAM is not affected by the CPU V/f level because the V/f levels of NoC, LLC, and DRAM are constant and independent of the CPU V/f level. Consequently, changing the V/f level of a CPU core linearly affects the CPI for these components, as can be seen in Eq. (5.3). All other components of the CPI stack are not affected by a V/f level change.

However, our experiments demonstrated that simple linear behavior does not accurately describe the system. *Sniper* [114], which is used to perform experiments in this chapter, reports a single CPI stack component for fetching instructions. This is a combination of L1-I, LLC, and DRAM accesses. The L1-I-related cycles are not affected by a V/f level change, whereas the others are affected linearly. To capture the resulting combination of linear and constant terms, we model the V/f level dependency as follows:

$$\mathrm{CPI}_i(\mathrm{AMD}, f) = (\beta_i \cdot f + (1-\beta_i)) \frac{m \cdot 2 \cdot l_{hop}}{N} \cdot (\alpha_i + \mathrm{AMD}) \tag{5.6}$$

The parameter β_i describes the relative sensitivity of CPI_i on the frequency f. Changing the CPU V/f level does not affect the executed instruction mix (m and N), and does not affect the V/f level of the NoC, LLC, and DRAM (l_{hop} and α_i). Therefore, changing the V/f level from f_1 to f_2 affects CPI_i according to the following equation:

$$\mathrm{CPI}_i(f_2) = \frac{\beta_i \cdot f_2 + (1-\beta_i)}{\beta_i \cdot f_1 + (1-\beta_i)} \cdot \mathrm{CPI}_i(f_1) \tag{5.7}$$

Solving Eq. (5.7) for β_i results in:

$$\beta_i = \frac{\mathrm{CPI}_i(f_2) - \mathrm{CPI}_i(f_1)}{\mathrm{CPI}_i(f_2) - \mathrm{CPI}_i(f_1) + f_2 \cdot \mathrm{CPI}_i(f_1) - f_1 \cdot \mathrm{CPI}_i(f_2)} \tag{5.8}$$

Eq. (5.8) can be used to calculate β_i from two traces of the same application at the same core (same AMD) but different V/f level. The parameters β_i are ultimately obtained by averaging over many pairs of traces of different benchmark applications. This is repeated for every component of the CPI stack that is related to the NoC, LLC, or DRAM.

5.5.3 Power Model

Since we need to predict the performance of a thread under a certain power budget, which is required to enforce the temperature constraint, we need a power model. A major requirement for building this model is low overhead, i.e., relatively low complexity. We, therefore, avoid modeling detailed behavior of individual microarchitectural parts of the processor like *McPAT* [115] and instead, base our model on the well-established activity-based power model:

$$\begin{aligned} P(V,f,\alpha) &= P_{leak} \quad + P_{dynamic} & (5.9) \\ &= I_{leak} \cdot V + \alpha \cdot (V - V_{th})^2 \cdot f & (5.10) \end{aligned}$$

The supply voltage V and the frequency f have an approximately linear relationship [134]:

$$f(V) = a \cdot V + b \tag{5.11}$$

Combining Eqs. (5.10) and (5.11) results in a cubic equation for the total power with parameters $\gamma^{(0)}(\alpha)$ to $\gamma^{(3)}(\alpha)$ that depend on the activity α.

$$P(f,\alpha) = \gamma^{(0)}(\alpha) + \gamma^{(1)}(\alpha) \cdot f + \gamma^{(2)}(\alpha) \cdot f^2 + \gamma^{(3)}(\alpha) \cdot f^3 \tag{5.12}$$

The CPI stack of a set of instructions is a decomposition of the execution into different actions. For instance, when waiting for the L1-D cache, the CPU stalls, while the L1-D cache is active. We reflect this decomposition also in the power model by assigning a switching activity α_i to every component of the CPI stack. This is equivalent to assigning a different set of parameters $\gamma_i^{(0)} := \gamma^{(0)}(\alpha_i), \ldots, \gamma_i^{(3)} := \gamma^{(3)}(\alpha_i)$ to every component. The total power is a linear combination of these activities and their relative duration:

$$P = \sum_i \frac{\mathrm{CPI}_i}{\mathrm{CPI}_{total}} \left(\gamma_i^{(0)} + \gamma_i^{(1)} \cdot f + \gamma_i^{(2)} \cdot f^2 + \gamma_i^{(3)} \cdot f^3 \right) \tag{5.13}$$

The parameters $\gamma_i^{(0)}$ to $\gamma_i^{(3)}$ can be obtained by fitting this power model to application traces at different V/f levels.

5.5.4 Algorithm for Analytical Performance Prediction

Migrating a thread affects the AMD and power budget of the core it is running on. Algorithm 5.1 shows the performance prediction algorithm that combines the three analytical models that have been constructed in the previous sections. Taking the impact of AMD into account is straightforward by applying the AMD model $\mathcal{F}_{AMD}$. A change in the AMD and power budget might necessitate switching to another V/f level to keep the power

Algorithm 5.1 Performance Prediction using CPI Stacks

```
$\text{CPI}_{\text{AMD}} \leftarrow \mathcal{F}_{\text{AMD}}(\text{CPI}_0, \text{AMD}_0 \rightarrow \text{AMD}_m)$        ▷ impact of AMD
for all $f_m \in$ BinarySearch($F$) do                                        ▷ binary search on V/f level
    $\text{CPI}_m \leftarrow \mathcal{F}_f(\text{CPI}_{\text{AMD}}, f_0 \rightarrow f_m)$                          ▷ impact of frequency
    $P \leftarrow \mathcal{F}_P(f_m, \text{CPI}_m)$                                            ▷ estimate power
    $\text{sat} \leftarrow P \leq P_{b,m}$                                                ▷ power budget satisfied?
    if sat then
        $\text{IPS}_m \leftarrow f_m / \sum_i \text{CPI}_i$
    UpdateBinarySearchLimits($f_m$, sat)
return $\text{IPS}_m$
```

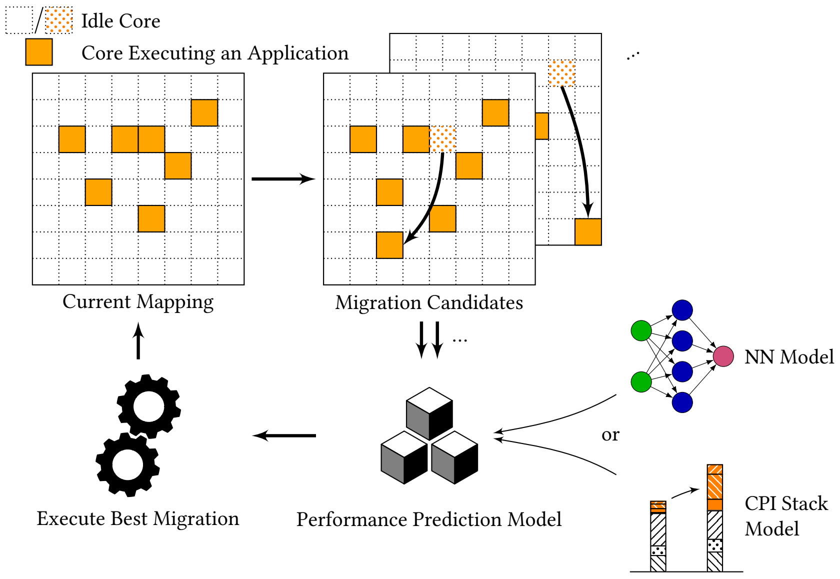


Figure 5.7: *PCMig* performs run-time application migration using a performance prediction model.

consumption close to the power budget. However, it is not clear which V/f level will be selected. Therefore, the frequency f_m is determined using a binary search. Iteratively, the CPI stack is calculated using the frequency model $\mathcal{F}_{freq}$ and the power is estimated using the power model $\mathcal{F}_P$ to check for a power budget violation. If the V/f level is sustainable, the IPS_m after migration can be estimated as $\text{IPS}_m = f / \text{CPI}_{total}$.

5.6 Run-Time Application Migration Algorithm

This section describes our proposed algorithm *PCMig* that performs application migration based on a performance prediction model. Fig. 5.7 gives an overview. Application migration is invoked periodically and comprises three steps: creating migration candidates, rating them, and executing the best migration.

Create Migration Candidates There are $O(n^{|T|})$ possible migrations in a many-core processor with n cores and a set of threads T. This number is too large to explore in its entirety. Therefore, we limit *PCMig* to only migrate one thread at a time or swap two threads. This reduces the number of migration candidates to $O(n \cdot |T|)$.

Rating of the Migration Candidates using Performance Prediction Each of these migration candidates needs to be rated to select the best one. A migration of a single thread affects the performance of this thread. Additionally, changing the mapping also changes the power budgets of all other threads, potentially affecting their performance, as well. Therefore, the performance of all threads needs to be predicted when rating a migration candidate.

Let M_0 denote the mapping before migration. A migration candidate m is defined by the mapping after the migration M_m. AMD and power budget values of all threads can both be calculated directly from the mapping. Cycle stacks can be obtained from hardware performance counters [70]. From these features, the following equation calculates the relative performance improvement over all threads T based on the performance prediction model $\mathrm{IPS}(t, M)$ (either NN-based or analytical):

$$\Delta_{IPS}(m) = \sum_{t \in T} \left(\frac{\mathrm{IPS}(t, M_m)}{\mathrm{IPS}(t, M_0)} - 1 \right) \tag{5.14}$$

No actual migration is performed to calculate this metric.

Executed Best Migration Finally, the migration candidate with the highest predicted performance improvement $\Delta_{IPS}(m)$ is selected. This migration is only executed if $\Delta_{IPS}(m) > \delta$. Migrations with $\Delta_{IPS}(m) < 0$ are predicted to reduce the overall performance. Migrations with $\Delta_{IPS}(m) \geq 0$ but $\Delta_{IPS}(m) < \delta$ are also not executed because the small expected performance improvement is likely outweighed by the performance penalty of the migration itself. Also, it prevents oscillations due to small prediction errors. To further reduce the overhead, we skip performance prediction if the AMD is unchanged and the power budget changes by less than 0.1 W, and assume that the performance does not change.

System Integration If a new application arrives at the system and needs to be mapped, information about its characteristics (e.g., CPI stacks) is not yet available. Therefore, the performance prediction models cannot be used. We adopt the application-agnostic algorithm from Chapter 4 to decide the initial mapping for new applications.

L1 caches are private per core. Migrating an application from one core to another core, therefore, results in cold L1 caches, and it takes some time until the thread's working data is present in the cache, which is known as cache warming. The performance of the thread is reduced during this time. Fig. 5.8 shows that the performance of different *PARSEC* threads saturates within 0.2 ms after a migration We set the migration epoch to 10 ms which is small enough to react fast to changes but large enough to still maintain a low overhead. With smaller migration epochs, the observed benefits in the performance of the

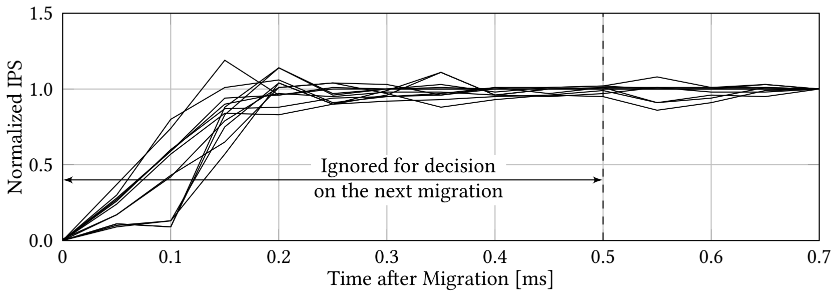


Figure 5.8: IPS of different threads after a migration. IPS values are normalized to the IPS at 0.8 ms after migration.

workload are smaller than the increase in the overhead. The duration during which the performance is reduced is less than 2 % of the migration epoch. Therefore, the threshold δ should be set to a value > 2 %. We set $\delta = 3\,\%$. During the period of reduced performance after a migration, performance counters do not reflect the true characteristics of the thread. For instance, they report too many cache misses. We ignore the performance counters during the first 0.5 ms after a migration.

5.7 Experimental Evaluation

This section presents an empirical evaluation of the performance prediction models and the application migration algorithm *PCMig*, which is based on such prediction models. We first compare the different prediction models to select the best-suited one. Then, the overall performance improvement of the system is evaluated with *PCMig* employing the selected performance prediction model.

The experimental evaluation uses the setup described in Section 3.1.2. Ambient and maximum temperature are set at 45 °C and 70 °C, respectively. TDP is set at 100 W. We execute the same applications from the *PARSEC* benchmark suite as in Chapter 4 with *simsmall* inputs: *blacksholes*, *bodytrack*, *canneal*, *dedup*, *fluidanimate*, *streamcluster*, *swaptions*, and *x264*.

5.7.1 Comparison of the ML-based and Analytical Prediction Models

We start by comparing the trade-off between prediction accuracy and run-time overhead of the different performance prediction models. We then evaluate costs related to creating and storing the models. Based on this comparison, we select one model to employ in our application migration algorithm *PCMig*.

A performance prediction model should have a low prediction error but at the same time have a low run-time overhead. These two metrics form a trade-off. We first evaluate the prediction accuracy of all models. The creation of the NN training data for different applications is described in Section 5.4.2. In total, we obtained 1,026,000 training and

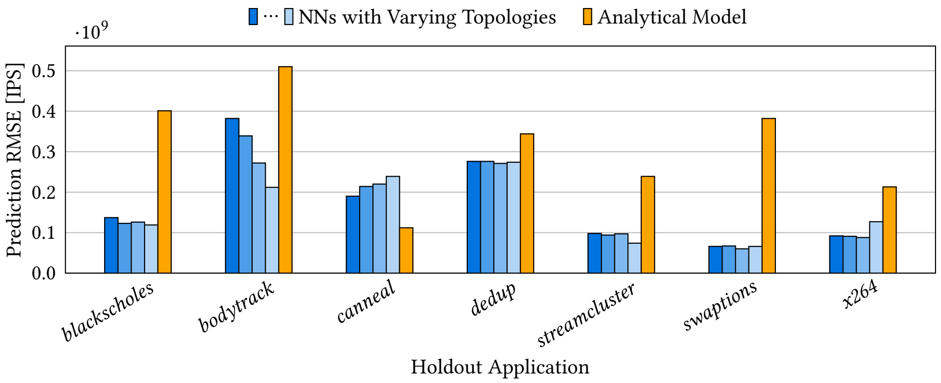


Figure 5.9: Prediction accuracy with the different models. Evaluation is done using k-fold cross-validation, in which the model is trained with all benchmarks except one and then tested on the holdout benchmark.

test examples. To achieve independent training and test sets, training examples obtained from one application are either used only for training or only for testing but never mixed between training and testing. We then perform a *k-fold cross-validation*, in which a model is trained on all applications except one *holdout* application. The model is then tested on the holdout application. This is repeated until every application was used as the holdout application.

Fig. 5.9 shows the achieved prediction accuracy for the NN-based models with different topologies (Section 5.4) and for the analytical model based on CPI stacks (Section 5.5). We report the root-mean-square error (RMSE) of the predictions. The analytical model almost always has a higher prediction error than the NN models. This error stems from accumulating inaccuracies of the individual models (AMD, V/f level, power). The only exception is *canneal*, for which the analytical model yields a lower error. The NN models differ in the employed topologies, which increase in complexity from left to right in the figure. The topology of the most complex NN is described in Section 5.4.3 and uses two hidden layers with *sigmoid* activation. The other three NNs only employ a single hidden layer with 12, 18, or 24 neurons, respectively, and use *hard sigmoid* activation (see Fig. 5.5). For some holdout applications, a rather simple model already achieves the best results. For example, for *x264*, the most complex model shows the highest error. However, for most holdout applications, a more complex topology decreases the prediction error.

The second metric to compare the models is the run-time overhead. The overhead is measured by the average single-threaded execution time of performance prediction. We execute the prediction on an intermediate core (AMD = 5.5 hops) of the simulated many-core processor at 4.0 GHz. The inference is computed in software and we do not assume any special hardware or accelerator. Fig. 5.10 shows the resulting trade-off between the run-time overhead and the prediction error. The analytical model has an average overhead of 0.16 µs per prediction. The largest contributor is the repeated evaluation of

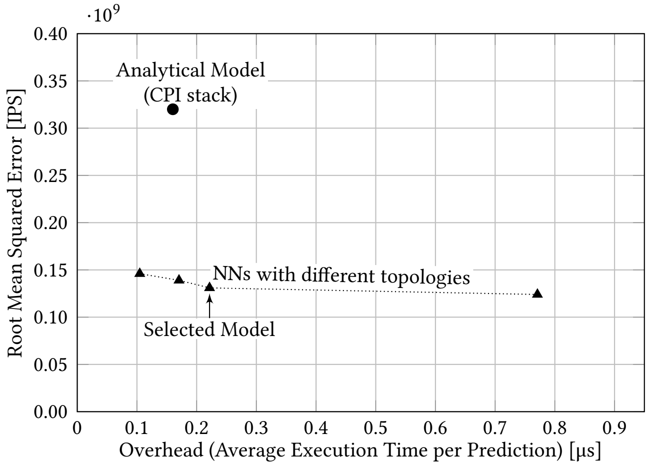


Figure 5.10: Comparison of the different models. NNs find a better trade-off between overhead and prediction error than an analytical model based on CPI stacks.

the power model (up to five times per prediction). However, reducing the complexity of the power model would increase the prediction error and, therefore, is not expedient as the analytical model already has a much larger prediction error than the NN model. The run-time overhead of the NN models strongly depends on the employed topology. The most accurate model also has by far the highest overhead (0.77 μs). Especially the *sigmoid* function is computationally expensive. The other NN models instead use the *hard sigmoid* activation function, and use only one hidden layer. Thereby, their overhead can be reduced to 0.10 μs, 0.17 μs, and 0.22 μs, respectively, at the cost of a slightly higher prediction error. This demonstrates the advantage of NN-based models that allow for easy exploration of different topologies, resulting in a different trade-off between overhead and prediction error. Overall, NN-based modeling outperforms analytical modeling and finds a better trade-off.

While the prediction accuracy and run-time overhead are the most important metrics in our case, we also evaluate additional metrics. Fig. 5.11a shows the training overhead of all models. We perform training of the NN-based models using the *TensorFlow* [135] library on an Intel Core i5-3470. We train the NN models for 50, 000 steps with mini-batches of size 32. The reported time does not include creating the training data (obtaining profiling information as described in Section 5.4.2), as this step is also required for fitting the parameters of the analytical model. We did not observe any overfitting because the employed models are small and we use regularization as described in Section 5.4.3. The training time only slightly increases with increasing model complexity. All NN models can be trained in less than 1 min. The analytical model has a much lower design-time overhead of around 1 s. However, since this is a one-time overhead, even 1 min is very low and can be neglected. Fig. 5.11b shows that the number of parameters is also low for

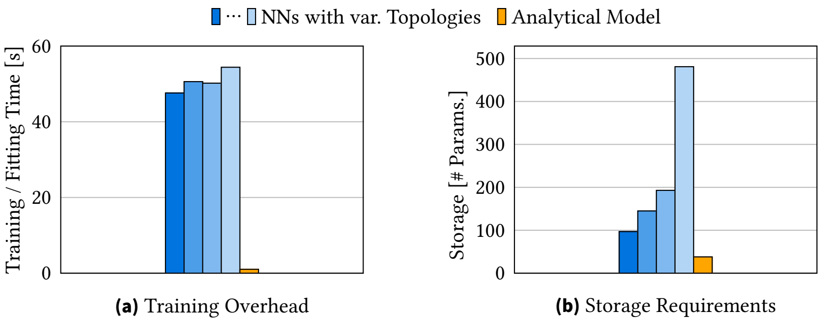


Figure 5.11: Comparison of training overhead and storage requirements of the models. The analytical model has the lowest design-time overhead. However, since this is a one-time overhead, 1 min for training the NN-based models is still very low. All models have low storage requirements because the number of parameters is low.

all studied models. The most accurate NN model has 481 parameters, which are stored as 32-bit floating-point numbers, which results in a memory requirement of less than 2 kB. All other models can be stored in less than 1 kB. Overall, all models have very low storage requirements, which do not incur a significant overhead.

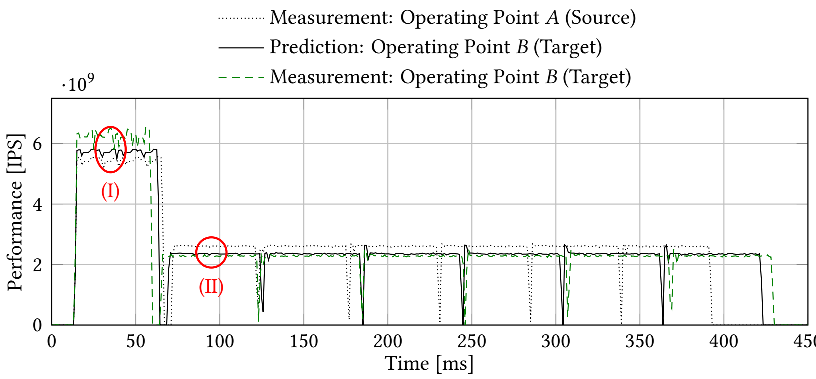
We select the NN model with 24 neurons in a single hidden layer to employ in *PCMig* because it combines a low prediction error ($0.13 \cdot 10^9$ IPS) with a low overhead (0.22 µs). Reducing the prediction error further comes at a high cost in terms of run-time overhead. Further optimizations like pruning or weight quantization potentially may reduce the run-time overhead and memory requirements because they allow avoiding floating-point operations and operate on reduced bit widths [136] but are beyond the scope of this work.

In summary, this analysis has shown that ML-based modeling achieves a better trade-off between accuracy and overhead than analytical modeling. Moreover, it achieves a flexible trade-off by adjusting the NN topology.

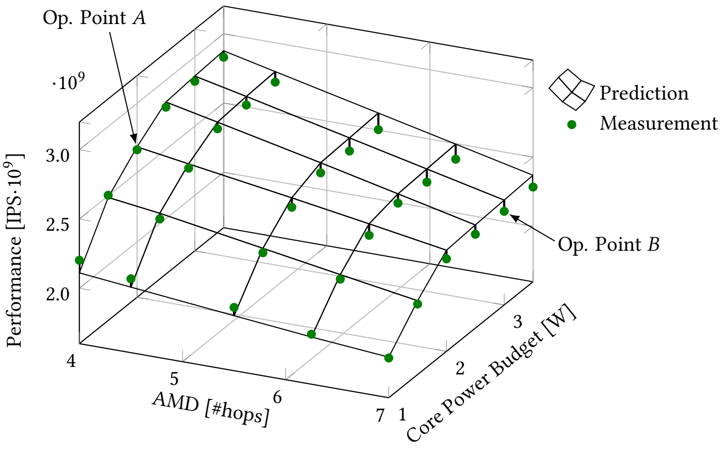
5.7.2 Illustrative Example for Performance Prediction

This section presents an illustrative example of the prediction capabilities (generalization to an unseen application) of the selected NN-based performance prediction model. Fig. 5.12a presents the transformation of a full trace of a slave thread of *PARSEC bodytrack* across two different operating points (each a combination of AMD and power budget). The trace is captured at Operating Point *A*, which executes the thread on a center core (AMD = 4 hops) with a power budget of 2.0 W. The performance prediction model has not been trained on *bodytrack* and is used to transform this trace to Operating Point *B*, which execute the thread on a corner core (AMD = 7 hops) at a higher power budget of 3.0 W.

During Phase I, the higher power budget of Operating Point *B* outweighs the increased LLC latency, and the performance of the thread is higher. The model correctly captures



(a) Prediction of a full trace. Measurements from Operating Point A are used to predict trace at Operating Point B.



(b) Prediction across many possible operation points at Phase II. Traces from Operating Point A are used to predict all other IPS.

Figure 5.12: Demonstration of the prediction accuracy of the IPS model for the unseen *bodytrack* application.

this behavior but slightly underestimates the performance. During Phase II, the thread becomes more memory-intensive and the increased LLC latency has a stronger impact on the performance. The model correctly predicts a reduced performance compared to Operating Point A in Phase II. *In both cases, the performance prediction model correctly takes the individual impacts of AMD and power budget into account.* Fig. 5.12b plots how the thread's performance during Phase II depends on AMD and power budget. The predictions of the performance model closely match the measurements. This further demonstrates that the ML model generalizes well to the unseen *PARSEC bodytrack* across various operating points.

5.7.3 Comparison to Classical Heuristics and to the State of the Art

This section compares the proposed application migration algorithm *PCMig* to classical heuristic mapping and to the state of the art. This is the main experiment in this chapter. We first investigate the achievable gains with application migration by studying the *PARSEC* benchmark individually and then employ mixed workloads. We compare our ML-based *PCMig* to two other application mapping/migration algorithms that are closest to our technique. Firstly, *Defrag* [62] employs application migration to improve the performance of applications running on a NoC-based message-passing architecture. Secondly, the heuristic *PCGov*, which has been introduced in Chapter 4, is designed for performance maximization of thermally-constrained many-core processors with shared LLC, and, therefore, targets the same problem as this chapter. However, it is based on simple heuristics that cannot take the application characteristics into account, forcing it to use an application-independent static mapping of threads to cores which makes a static trade-off between AMD and power budget.

Uniform Workloads We create workloads of several instances of the same *PARSEC* benchmark application that result in full system utilization, i.e., 64 threads. This forces the application mapping or migration algorithm to use all cores of the many-core processor including the corner cores that have high AMD. All applications are started at the same time and we report the time until the last application has finished (makespan).

Fig. 5.13a presents the results. Our ML-based *PCMig* improves the performance for almost all benchmarks. *Defrag* is designed for message-passing and maps threads of the same application close to each other. However, spatial proximity is not beneficial in the case of a distributed shared LLC. It even increases the makespan because some applications are squeezed into the corners of the many-core processor, which has two drawbacks. Firstly, cores in the corner have a high AMD and, therefore, threads experience a high average LLC latency. Secondly, mapping threads close to each other creates a thermal hotspot. This reduces the power budgets, which requires to operate the cores at lower V/f levels. The simple heuristic *PCGov* employs a static application-agnostic mapping and, therefore, neither considers the actual application characteristics nor reacts to execution phases. This is most visible with *canneal*. Its performance mainly depends on the performance of its master thread. Furthermore, the master thread is very memory-intensive and its performance is highly sensitive to the AMD but barely depends on the power budget. Therefore, the application-agnostic trade-off between these two factors that *PCGov* makes is suboptimal. *PCMig* starts with the same initial mapping but quickly migrates all master threads to the center, which better considers the application characteristics and greatly improves the performance, achieving a 20 % lower makespan.

Streamcluster is a short-running application which has a short initial phase in which the master is active, and then all slave threads are active. *PCMig* reacts to the initial phase by migrating the master to the center of the many-core processor. This brings almost no performance benefit because the master terminates shortly after. When the slave threads are active, the center core with low AMD is occupied by the now idle master. It takes

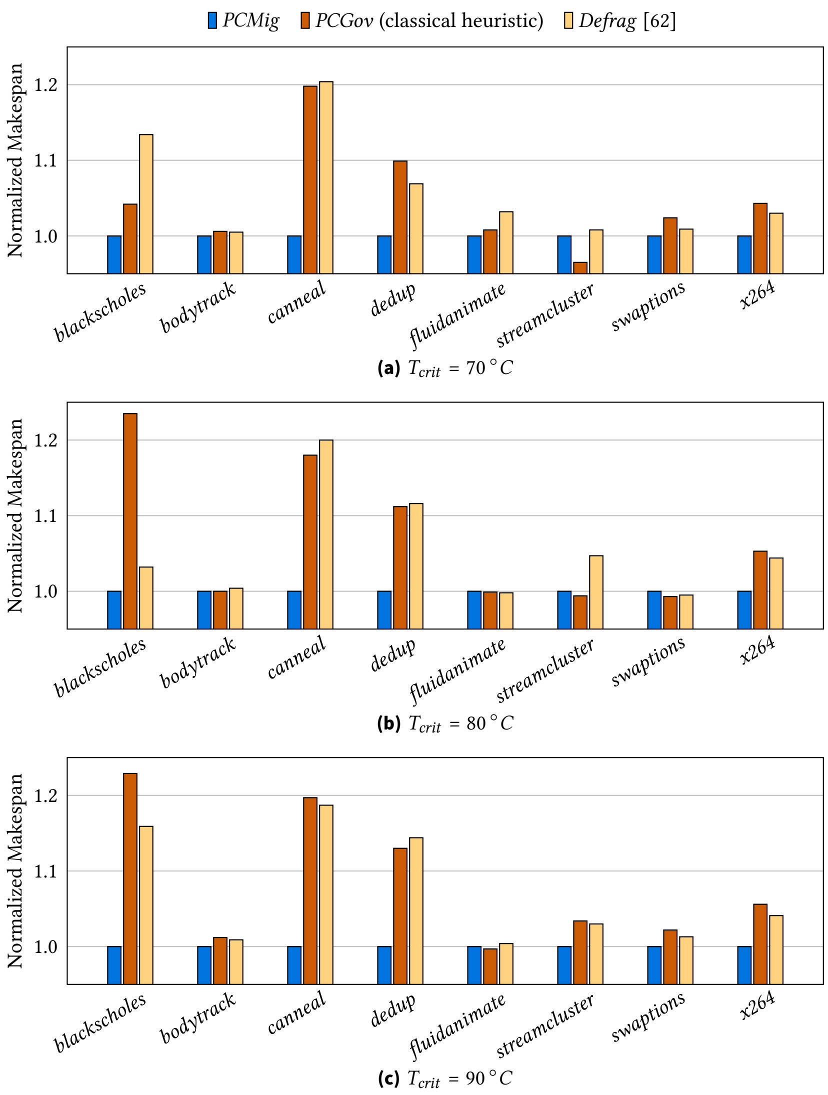


Figure 5.13: Performance with *PCMig* for isolated workloads. ML-based *PCMig* improves the performance by up to 25 % over classical heuristic management and the state-of-the-art. Observed benefits depend on the maximum temperature. A different maximum temperature changes the power budgets and, thereby, shifts the trade-off between power budget and LLC latency. Only adapting the mapping to the application characteristics maximizes the performance.

some time until the master is migrated away to make room for a slave thread. Overall, this temporary suboptimal mapping slightly reduces the performance over a static mapping, which keeps the master in the corner. We verified this in an additional experiment with a very low migration epoch, i.e., the mapping is adapted very fast. Faster migration would reduce the makespan to 50.3 ms, which is slightly lower than *PCGov* (50.6 ms) but at the cost of an increased run-time overhead.

The trade-off between power budget and average LLC latency lies at the heart of this chapter. The power budget is determined by the thermal constraint T_{crit}. A higher maximum temperature increases the power budget with a certain mapping and, thereby, changes the trade-off between power budget and average LLC latency. We perform additional experiments with different thermal constraints. Fig. 5.13b and 5.13c show the results for a thermal constraint of 80 °C and 90 °C, respectively. The advantage of ML-based *PCMig* over classical heuristic *PCGov* increases with increasing temperature. The reason is that *PCMig* can adapt the mapping to the changed trade-off by employing a performance prediction, while *PCGov* uses an application-agnostic static mapping. Similarly, the advantage over *Defrag* increases. Most importantly, this is achieved *without retraining the model.*

These experiments show that application-agnostic heuristic mapping (like *PCGov*) results in a performance loss of up to 20 %. It furthermore demonstrates that the optimal mappings depend on the application characteristics. Because application characteristics can only be observed when the application is running and are not known at mapping time, application migration is required to fine-tune the mapping. Our ML-based *PCMig* achieves a high performance for all applications because it can adapt the mapping to the application characteristics. We also demonstrate that *PCMig* generalizes well to different thermal configurations without the need to retrain the model.

Mixed Workload To evaluate the average performance gains, we conduct an additional experiment in which we execute a random workload consisting of 20 randomly sampled *PARSEC* applications with random parallelism level. To model an open system, we sample the times between application arrivals from a Poisson distribution with varying arrival rates. Thereby, the average (peak) system utilization varies between 9 % (33 %) and 36 % (86 %). Finally, we report the average response time for each resource management algorithm. We set the thermal constraint back to 70 °C.

Fig. 5.14 present the results. Across all application arrival rates, ML-based *PCMig* yields the highest performance, i.e., lowest average response time. It outperforms *Defrag* by up to 18 %. This is because *Defrag* has been designed for a message passing architecture, as has been discussed in the previous experiment. The classical heuristic *PCGov* has been designed for many-core processor with distributed shared LLC but does not consider application characteristics. It has to find mappings that result in good average performance for many applications, and cannot react to individual application characteristics. Furthermore, it does not react to changing application execution phases or changing system load. Therefore, *PCMig* outperforms *PCGov* by up to 7.3 %, where some applications experience

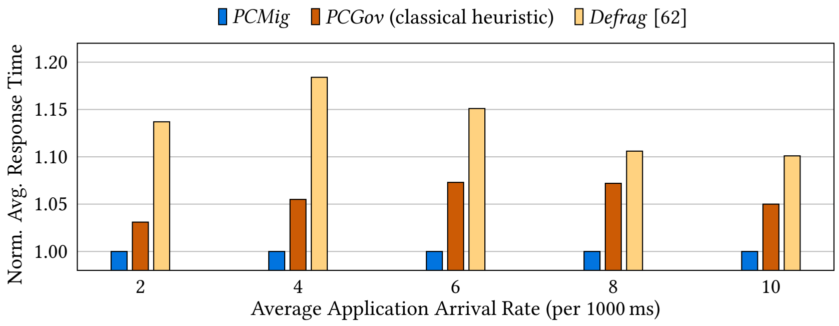


Figure 5.14: ML-based *PCMig* improves the average performance with mixed random workloads by up to 7.3 % over classical heuristic *PCGov*, and by up to 18.4 % over state-of-the-art *Defrag*.

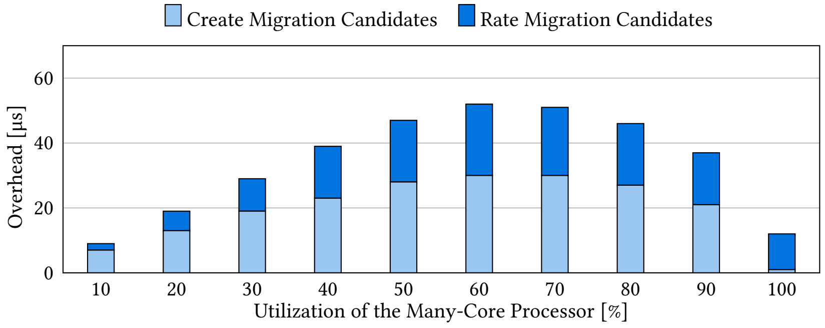


Figure 5.15: Parallelized run-time overhead of *PCMig*.

a reduction in the response time of up to 19.4 %, while other applications experience almost no speedup.

5.7.4 Run-Time Overhead

This section evaluates the run-time overhead of *PCMig* to decide the next migration. Deciding the next migration comprises creating migration candidates and rating each one. The overhead mainly depends on the number of migration candidates. Therefore, we study different utilization values (number of active cores) of the many-core processor.

Fig. 5.15 presents the overhead per invocation of *PCMig* if parallelized to run on all cores. There are two types of migration candidates. Firstly, migrations that migrate a thread to an idle core, and secondly, migrations that swap two threads. The number of migrations depends on the number of threads $|T|$ and the number of cores n in the many-core processor. There are $|T| \cdot (n - |T|)$ migrations to an idle core, and $\frac{1}{2} \cdot |T| \cdot (|T| - 1)$ migrations that swap two threads.

Creating migration candidates consists mainly of calculating the new power budgets after the migration. Migrations that swap two threads do not change the location of active cores and, therefore, do not change the power budgets. The maximum number of migrations of a thread to an idle core is observed at $|T| = \frac{n}{2}$, i.e., at a 50 % system utilization. Calculating the power budgets requires solving linear equation systems. The size of the involved matrices increases with $|T|$. Therefore, the maximum overhead for creating migration candidates is observed at a 60 % system utilization.

After creating migration candidates, each of them needs to be rated. Swapping two threads only affects the AMD and power budget of these two threads. Therefore, only two invocations of the performance prediction model are required per migration candidate. Migrating a thread to an idle cores affects the power budgets of all threads, potentially requiring performance prediction for each thread (if the power budget changes by more than 0.1 W). This results in $\mathcal{O}(|T|)$ invocations per migration candidate. In total, $|T| \cdot (|T| - 1) + \mathcal{O}(|T|^2 \cdot (n - |T|))$ predictions are performed. The overhead of performance prediction also peaks at 60 % system utilization. The total overhead of the proposed application migration algorithm *PCMig* is less than 52 µs. Migrations are invoked every 10 ms, which results in a negligible maximum overhead of 0.5 %.

5.8 Summary

This chapter presented *PCMig*, which performs application migration with the help of an ML model to predict the impact of a migration before executing it. This pattern allows to build proactive resource management where no action is executed without predicting its impact first. Two fundamentally different methods have been studied to create the model: analytical modeling and ML-based modeling. The ML based model with an NN achieves a better trade-off between error and overhead, without requiring detailed knowledge about the internals of the platform. *PCMig*, which employs the NN model, significantly increases the performance of mixed workloads with unseen applications. In addition, we showed that *PCMig* can cope with different thermal constraints without requiring retraining of the model.

The main challenge observed in this chapter is that it requiring separate inference calls per each candidate action potentially incurs a high run-time overhead. This could be solved in *PCMig* by employing a small NN model, and by restricting the set of candidate actions. Additionally, *PCMig* performs DVFS with the help of per-core power budgets, which greatly reduces the complexity of the resource management, but prevents further optimization. When studying more complex optimizations, such as per-core DVFS, as will be studied in the next chapter, or joint application migration and DVFS, as will be studied in Chapter 7, employing the pattern of predicting the impact of potential resource management actions would likely result in an unaffordable overhead.

6 Smart Boosting by Estimating Hidden Application Properties

DVFS has a major impact on the performance and power of applications, which is why it is implemented in virtually all modern processors. In contrast to the previous chapters, which indirectly controlled the V/f levels via a power budget, the technique presented in this chapter directly controls the V/f levels to achieve a more fine-grained optimization. This chapter targets the problem of *boosting*, i.e., maximizing the performance under a thermal constraint using DVFS. The developed technique implements the pattern to use an ML model to estimate hidden properties of applications, which has been introduced in Section 1.4.2. In particular, predictions about the sensitivity of performance and power to V/f changes are required to perform boosting optimization. Considering the sensitivity of the power and temperature within the boosting optimization also enables to proactively estimate the thermal safety of boosting decisions. The developed ML-based management is compared to both simple heuristic management and management based on analytical modeling.

Processor manufacturers developed boosting techniques like Intel TurboBoost [56] or AMD Turbo Core, which employ simple heuristics that upscale the V/f levels of all active cores simultaneously if thermal margin exists. These techniques perform boosting without specific knowledge of the applications running on the cores. However, diverse applications benefit differently from boosting. For instance, compute-intensive applications benefits strongly, while memory-intensive applications benefits to a lesser degree. This observation has been exploited in prior works that perform the boosting decision based on the IPS of the running applications [67]. Particularly, high-IPS applications are boosted while low-IPS applications are throttled, assuming that compute-intensive applications have high IPS, while the memory-intensive ones have low IPS. However, this is not always the case as will be demonstrated in the following examples.

6.1 Motivational Examples

This section presents two motivational examples that demonstrate the difficulties and important aspects in boosting.

This chapter is mainly based on [5].

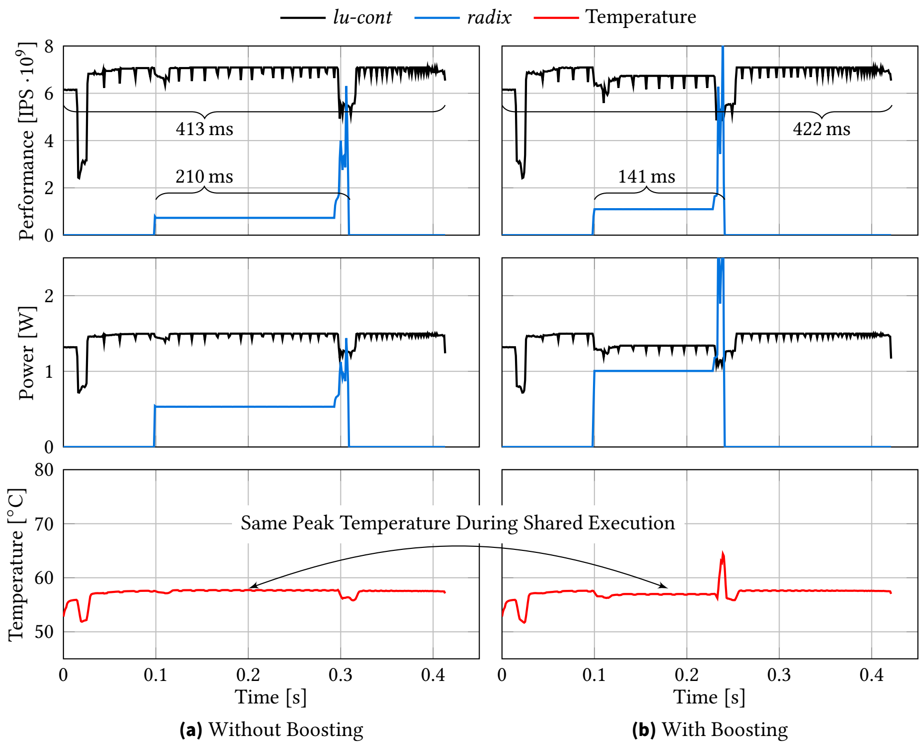


Figure 6.1: Motivational example for boosting. Boosting the application with low IPS significantly improves the overall performance. This appears counter-intuitive at first sight but is a key observation contributing to the efficiency of our smart boosting.

Boosting In Fig. 6.1, a high-IPS high-power application (*lu.cont*) and a low-IPS low-power application (*radix*) from the *SPLASH-2* benchmark suite are running in parallel. In Fig. 6.1a, both applications operate at the same V/f level of 2.0 GHz. The peak temperature is 58 °C. Fig. 6.1b illustrates that boosting *radix* (low IPS) and throttling *lu.cont* (high IPS) leads to significant improvements in the overall system performance, which is defined as the average response time of all running applications in the system. Particularly, the execution time of *radix* is reduced by 33 % and the execution time of *lu.cont* increases by only 2 %, without affecting the peak temperature.

This is unlike what is expected. There are two reasons for it. Firstly, the power consumption of the applications significantly impacts the performance benefits of boosting. Specifically, throttling a high-power application (*lu.cont*) by a little change in the V/f level, i.e., by 100 MHz, enables to boost a low-power application (*radix*) by a big change in the V/f level, i.e., 1.0 GHz, while the peak temperature remains almost the same. This leads to a significant increase in the overall system performance. This observation has not been exploited before. Secondly, *radix*, despite having low IPS, is compute-intensive, and thus

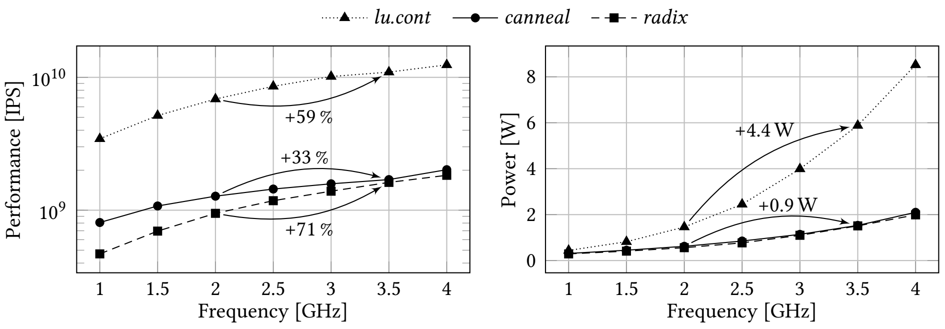


Figure 6.2: V/f sensitivities of performance and power consumption vary widely from one application to another, due to their different characteristics.

significantly benefits from an increased V/f level. The reason for the low IPS of *radix* is that it has many operations that depend on the result of previous operation, which leads to many stall cycles in the CPU pipeline. *This example demonstrates that considering only the performance of the applications during the boosting optimization is suboptimal, power needs to be considered, too.*

V/f Sensitivites of Performance, Power, and Temperature Involving the absolute values of performance and power may be beneficial for boosting, as discussed above, but considering their sensitivities to V/f changes is more relevant. Fig. 6.2 shows an example. The three applications *SPLASH-2 lu.cont*, *PARSEC canneal*, and *SPLASH-2 radix* have widely varying absolute IPS. Their V/f sensitivities of the performance, i.e., how the performance changes with a V/f change, are also very different, where the memory-intensive *canneal* benefits the least, and the compute-intensive *radix* benefits the most. Importantly, these two metrics are not interchangeable. *Radix* has the lowest absolute IPS but the highest sensitivity of the performance. *Canneal* has a similar low absolute IPS but a low sensitivity of the performance. Hence, the sensitivity of the performance is the important metric to be included into the boosting optimization instead of the absolute IPS. Additionally, the V/f sensitivity of the power is illustrated in Fig. 6.2, where large variations between different applications can be seen. The power consumption of *lu.cont* increases by 4.4 W when boosting from 2.0 GHz to 3.5 GHz, while the power consumption of *canneal* and *radix* increases by only 0.9 W. The sensitivity of the power consumption follows different trends then the sensitivity of the performance. *Canneal* and *radix* have very different sensitivities of the performance, but almost indistinguishable power. While the V/f sensitivity of the performance has been adopted in some of the state-of-the-art boosting techniques, e.g., [72], none of them includes also the V/f sensitivity of the power into the boosting optimization.

In addition to the performance and power, temperature also needs to be included into the boosting optimization. The reason is that also temperature and power are not interchangeable. The same power consumption on different cores can result in very different temperatures, e.g., because of the location of the cores on the chip, or because of the power consumption of other cores. Therefore, the cores on the chip have different sensitivity of the temperature. *In summary, three metrics need to be included into the boosting optimization: the application-dependent V/f sensitivities of the performance and power, and the core-dependent sensitivity of the temperature.*

6.2 Challenges and Novel Contributions

Developing a boosting technique that involves all the aforementioned three metrics requires tackling the following key challenges:

1. V/f sensitivities of performance and power depend on the vary between applications but cannot be directly measured. Hence, they need to be estimated at run time. The estimation method needs to work for unknown applications that arrive on an open system and with the consideration of their varying and diverse characteristics (memory accesses, data dependencies, etc.).
2. All three metrics need to be integrated within a comprehensive boosting optimization. However, these metrics may be contradicting. For instance, one application may have a higher sensitivity of the performance, but also a higher sensitivity of the power compared to another application.

The first challenge is tackled by employing an NN model, which is capable of learning complex problems and—if designed and trained well—generalize to unknown applications, as discussed in Section 1.3. The second challenge could be addressed by considering all three metrics consecutively, i.e., first consider one metric, then another, to finally reach a boosting decision. However, such an approach would lead to suboptimal decisions. Instead, we *jointly* involve all three metrics within the optimization, by deriving one boosting metric that integrates them all.

Therefore, this chapter introduces a smart, yet lightweight, boosting technique that, for the first time, jointly involves the V/f sensitivities of performance, power, and temperature within the optimization. This technique not only throttles applications when required to prevent thermal violations but also throttles applications to enable boosting another application if this increases the overall performance. This smart boosting is enabled by the following novel contributions:

- We derive a novel boosting metric that integrates the application-dependent V/f sensitivities of performance and power, and the core-dependent sensitivity of the temperature. It indicates the efficacy to boost an application.
- For that, a single multi-task NN model is designed and employed to predict the varying V/f sensitivities of performance and power of unknown applications.
- Based on the new boosting metric and the developed NN model, we build a smart, yet lightweight, run-time boosting technique for multi-threaded applications.

6.3 Problem Definition

The technique developed in this chapter targets a homogeneous (bus-based) many-core processor with n cores. It targets an open system, where unknown multi-threaded applications arrive at a priori unknown times, and the one-thread-per-core model common in many-core processors [40]. The many-core processor supports per-core DVFS. The objective is to maximize the system performance under a thermal constraint T_{crit}. The overall system performance is measured by the average response times of the applications, which captures their end-to-end performance. This goal is tackled using DVFS-based boosting because it is an always-available low-overhead means with a large impact on performance, power and temperature [137].

We divide the problem in two parts. First, jointly considering power, performance, and temperature within the boosting optimization is achieved by introducing a novel *boostability* metric. Second, the required inputs for this metric, i.e., sensitivities of performance and power of applications, are estimated with an ML model because they cannot be measured at run time.

6.4 Boosting Metric: Boostability

As argued earlier, three parameters need to be integrated in the boosting optimization: the V/f sensitivities of performance, power, and temperature. The sensitivities of performance and power depend on the application characteristics. Applications comprise several threads with potentially different characteristics. In the following, we first model the sensitivities at the granularity of threads and then extend it to multi-threaded applications.

6.4.1 Boostability of Single Threads

Because our performance metric is the applications response time, the sensitivity of the performance is not related to the absolute IPS change upon a V/f change, but to the relative change w.r.t. the current IPS of the thread:

$$s^{perf} = \frac{\partial \mathrm{IPS}}{\partial f} \cdot \frac{1}{\mathrm{IPS}_c}, \qquad \text{unit: \%/GHz.} \tag{6.1}$$

IPS_c is the thread's IPS in its current execution phase and V/f level. Its unit is %/GHz, indicating by how much the performance changes per frequency change.

We consider an absolute temperature limit. Temperature is affected by absolute power values, so we need to consider the absolute power changes in the V/f sensitivity of a thread's power consumption P:

$$s^{pw} = \frac{\partial P}{\partial f}, \qquad \text{unit: W/GHz.} \tag{6.2}$$

Its unit is W/GHz, indicating by how much the power of this thread changes per frequency change.

The sensitivities of performance and power depend on the V/f level but also on the thread characteristics that vary between threads and over time with the execution phases. These sensitivities cannot be measured directly at run time via the available performance counters, and, therefore, an ML model is employed to predict them as will be elaborated in Section 6.6.

Finally, the sensitivity of the temperature needs to be considered. We employ the well-established RC-thermal model [44] that has been introduced in Section 3.1. Changing the V/f level f_i of a single thread running on core i changes the power consumption according to its current sensitivity of the power s_i^{pw}, which changes one element P_i of the power vector P, and, therefore, changes every steady-state temperature $T_{steady,j}$, as defined in Eq. (3.2), of every thermal node j:

$$\frac{\partial T_j}{\partial f_i} = \frac{\partial \left(\sum_{k=1}^{N} B_{jk}^{-1} \cdot P_k + T_{amb} \right)}{\partial f_i} = B_{ij}^{-1} \cdot s_i^{pw} \tag{6.3}$$

Considering the steady-state temperature allows to reduce the complexity of the optimization. We demonstrate in the evaluation in Section 6.7.1 that our technique is also effective in preventing thermal violations in the transient temperatures.

To maximize the system performance under a thermal constraint, the thread whose performance increases the most for the available thermal margin should be boosted. Therefore, we build a single metric that we simply call *boostability b*. It integrates the sensitivities of performance, power, and temperature into a single metric to indicate the change in the performance per peak temperature change. The peak temperature is determined by the core with the highest temperature, which forms the thermal hotspot. We denote its index $h = \arg\max_j T_j$. The boostability b of a thread running on core i is described by:

$$b = \frac{s^{perf}}{B_{ih}^{-1} \cdot s^{pw}}, \qquad \text{unit: \%/°C.} \tag{6.4}$$

The best boosting benefit is obtained by boosting the thread with the highest value of the boostability. When throttling is required to avoid potential thermal violations, the thread with the lowest boostability value should be throttled first because that will minimize the performance penalty to restore thermal safety. Note that the boostability needs to be recomputed at every iteration of the boosting algorithm, because the thermal hotspot or the application characteristics might change.

6.4.2 Boostability of Multi-Threaded Applications

Optimizing the performance of individual threads may not increase the application performance because that might lead to running the individual threads of an application at different V/f levels, potentially leading to synchronization stalls between the threads [131].

Therefore, boosting should be performed at the level of applications. The sensitivities of power, performance, and temperature are properties of the threads and the cores they are running on. We need to combine them into a single metric per each application.

The combined sensitivities of power and temperature simply need to be added because the individual impacts of the threads accumulate on the thermal hotspot. For the sensitivity of the performance, it matters whether the application should be boosted or throttled. The thread with the smallest sensitivity of the performance is affected the least by boosting and determines the application's overall sensitivity of the performance. In contrast, the thread with the highest sensitivity of the performance is affected the most by throttling, and, hence, needs to be considered for throttling. In summary, Eq. (6.4) needs to be extended to define the boostability of an application for boosting ($b^{\uparrow}$) and throttling ($b^{\downarrow}$) based on the sensitivities of all its threads A:

$$b^{\uparrow} = \frac{\min_{i \in A} s_i^{perf}}{\sum_{i \in A}(s_i^{pw} \cdot B_{ih}^{-1})}, \qquad b^{\downarrow} = \frac{\max_{i \in A} s_i^{perf}}{\sum_{i \in A}(s_i^{pw} \cdot B_{ih}^{-1})} \tag{6.5}$$

6.5 Neural Network Model for Sensitivity Prediction

Our algorithm *SmartBoost* is based on the *boostability* metric derived in the previous section. Calculating the *boostability* per application requires knowledge about their V/f sensitivities of performance and power of each of their threads. However, these metrics cannot be measured directly at run time. Measuring the V/f sensitivities at run time would require changing the V/f level during operation of the thread and observing how its power and performance change. This would need to be done for all threads because different threads have different characteristics. Additionally, this would need to be done periodically because thread characteristics may change over time with the application's execution phases. Overall, the induced overhead would be unaffordable. Since we target an open system executing unknown applications, design-time profiling information is also not available. The only metrics observable at run time are hardware performance counters, which provide information about the thread characteristics but require interpretation. Therefore, we develop an NN-based model to estimate the sensitivities of performance and power from the performance counters. This model generalizes to unseen applications because it is lightweight, i.e., low number of parameters, which prevents overfitting. A simpler model could be employed for the prediction, but with a lower accuracy, as will be shown in Section 6.7.3.

6.5.1 Feature Selection

Many potential features are available per thread: the current V/f level and the performance counters. The performance counters are represented as CPI stacks that map the cycles to execute a certain set of instructions to the microarchitectural components (caches, DRAM, branch predictor, etc.) that cause them, as explained in more details in Section 3.1. They form a very concise representation of the application characteristics. We select the

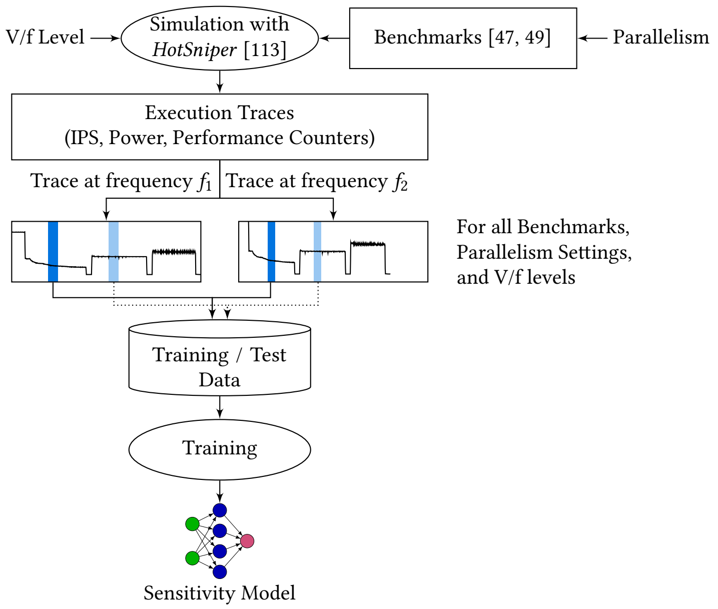
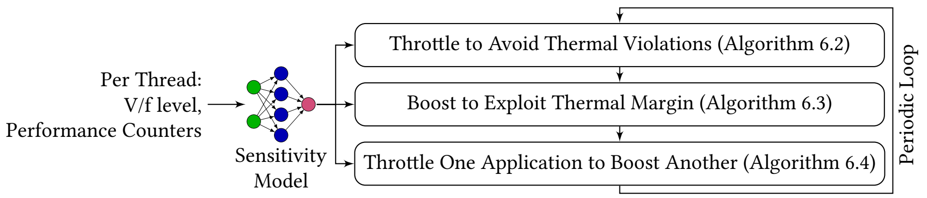

V/f Level
Simulation with *HotSniper* [113]
Benchmarks [47, 49]
Parallelism
Execution Traces (IPS, Power, Performance Counters)
Trace at frequency f_1
Trace at frequency f_2
For all Benchmarks, Parallelism Settings, and V/f levels
Training / Test Data
Training
Sensitivity Model

Figure 6.3: Training the NN model for *SmartBoost* at design time comprises profiling benchmarks, dividing traces into slices, creating training data, and performing the actual training.

current V/f level as the first feature because it has the highest impact on the sensitivities of performance and power. For instance, a small V/f level increase has a higher impact on the performance if the application is operated at a low V/f level, and a higher impact on the power if the application is already operated at a high V/f level. A crucial characteristic of a thread is its memory-intensiveness. The more memory-intensive a thread is, the lower are its sensitivities of performance and power. We select the ratio of all memory-related CPI stack parts over the total CPI as a second feature. This feature describes the fraction of cycles spent waiting on memory.

6.5.2 Training Data Generation

Fig. 6.3 shows the flow to create training and test data for the NN model. We first record execution traces, which comprise IPS, power, and performance counters, of benchmark applications from the *PARSEC* and *SPLASH-2* benchmark suites at different V/f levels. Each application is executed at different levels of parallelism to increase the diversity of application characteristics. To capture execution phases, we follow a similar approach as in Section 5.4.2 and divide the traces of every thread of every application into 20 slices with the same number of executed instructions. We generate training/test examples by comparing slices that only differ in the V/f level but execute the same application, same parallelism, and same instructions. We restrict these comparisons to pairs where the difference in the V/f levels is small, i.e., $\leq$ 500 MHz. Every pair contains information about



Per Thread:
V/f level,
Performance Counters
Sensitivity
Model
Throttle to Avoid Thermal Violations (Algorithm 6.2)
Boost to Exploit Thermal Margin (Algorithm 6.3)
Throttle One Application to Boost Another (Algorithm 6.4)
Periodic Loop

Figure 6.4: Overview of *SmartBoost* management at run time.

a potential V/f change that is used to calculate the sensitivities of performance and power to use as labels in the training data.

$$s^{perf} = \frac{\mathrm{IPS}(f_2) - \mathrm{IPS}(f_1)}{(f_2 - f_1) \cdot \mathrm{IPS}(f_1)}, \qquad s^{pw} = \frac{P(f_2) - P(f_1)}{f_2 - f_1} \tag{6.6}$$

The features for each training/test example are taken from the first trace at f_1.

6.5.3 Neural Network Topology

An intuitive approach to predict the V/f sensitivities of performance and power would be to employ two separate models. However, despite being not interchangeable, these two metrics are still to some degree related. For instance, a thread with a low sensitivity of the performance is more likely to also have a lower sensitivity of the power. Building two separate models for the two metrics would result in similar internal features being learned by the models. Therefore, to reduce the inherent replication and decrease the run-time overhead, we build a single model with two outputs to predict both metrics simultaneously. Such an approach is known as *multi-task learning* [138]. A small fully-connected feedforward NN with two hidden layers of 16 and 8 neurons, respectively, achieves a high accuracy, while keeping a low overhead. The NN is trained once at design time and used only for inference at run time.

6.6 Smart Boosting Algorithm

Our *SmartBoost* algorithm (Fig. 6.4) integrates the NN sensitivity model and the *boostability* metric to perform boosting optimization. Its inputs are 1) the current power consumption per core, 2) the current per-core V/f levels, and 3) the performance counter measurements per thread required for the sensitivity model. Its outputs are the per-application V/f levels for the next epoch.

Algorithm 6.1 shows the overall algorithm. It first estimates the performance and power sensitivities of each thread i based on its V/f level f_i and performance counter measurements C_i using the NN sensitivity model M. In the next step, it sets the V/f level bounds for each application j for the optimization in the current epoch. We only allow for a change of up to k V/f levels in each epoch because the sensitivities of performance and

Algorithm 6.1 *SmartBoost*

```
for each thread $i$ do
    $(s_i^{perf}, s_i^{pw}) \leftarrow M(C_i, f_i)$        ▹ sensitivity predictions of threads
for each application $j$ do
    $(f_j^{min}, f_j^{max}) \leftarrow (\text{clip}(f_j - k \cdot \Delta f),\ \text{clip}(f_j + k \cdot \Delta f))$
$T \leftarrow T_{amb} + B^{-1}P$        ▹ initialize steady-state temperature
ThrottleToAvoidThermalViolations()        ▹ Algorithm 6.2
BoostToExploitThermalMargin()        ▹ Algorithm 6.3
ThrottleAppToBoostAnother()        ▹ Algorithm 6.4
```

Algorithm 6.2 Throttle to Avoid Thermal Violations

```
while $\max T > T_{crit}$ do
    $h \leftarrow \arg\max_i T_i$        ▹ position of the hotspot
    for each application $j$ do
        $c_j \leftarrow \Delta f \cdot \sum_{i \in A_j} \left( s_i^{pw} \cdot B_{ih}^{-1} \right)$        ▹ reduction of $T_h$ per V/f level
        $b_j^{\downarrow} \leftarrow \frac{\max_{i \in A_j} s_i^{perf}}{c_j}$        ▹ boostability of application $j$
    $t \leftarrow \arg\min_{j: f_j > f_j^{min}} b_j^{\downarrow}$        ▹ application to throttle
    $\delta \leftarrow \min \left( f_t - f_t^{min}, \Delta f \cdot \left\lceil \frac{\max T - T_{crit}}{c_t} \right\rceil \right)$        ▹ V/f level change
    $f_t \leftarrow f_t - \delta$        ▹ set new V/f level
    $T \leftarrow T - \delta \cdot \sum_{i \in A_t} \left( B_i^{-1} \cdot s_i^{pw} \right)$        ▹ update steady-state temperature
```

power depend on the V/f level. Therefore, the estimations which are obtained at the current V/f level, are only valid for small V/f changes. The estimate of the steady-state temperature is initialized based on the current power consumption. Three cases may occur at run time that require action: 1) a thermal violation is about to happen, or 2) there is a thermal margin that can be exploited, or 3) V/f levels are suboptimal, where throttling an application to boost another increases the system performance. Each of these phases is implemented in a separate algorithm. The following three sections describe each of them.

Throttle To Avoid Thermal Violations Algorithm 6.2 throttles applications until the steady-state temperatures of all cores are safe, i.e., below T_{crit}. As explained in Section 6.4, the boostability depends on the hotspot location. Therefore, we calculate the boostability of each application given the current hotspot location. We throttle the application with the lowest boostability $b^{\downarrow}$ that is not already at its minimum V/f level, which could either be overall minimum V/f level or the lower V/f level bound determined for the current epoch. The V/f level is reduced such that the thermal violation is resolved, or the minimum V/f level is reached. Then, the steady-state temperatures are updated based

Algorithm 6.3 Boost to Exploit Thermal Margin

```
while True do
    $h \leftarrow \arg\max_i T_i$                                          ▷ position of the hotspot
    for each application $j$ do
        $c_j \leftarrow \Delta f \cdot \sum_{i \in A_j} \left( s_i^{pw} \cdot B_{ih}^{-1} \right)$          ▷ reduction of $T_h$ per V/f level
        $b_j^{\uparrow} \leftarrow \frac{\max_{i \in A_j} s_i^{perf}}{c_j}$                      ▷ boostability of application $j$
    $t \leftarrow \arg\max_{j : f_j < f_j^{max}} b_j^{\uparrow}$                          ▷ application to boost
    if no target application $t$ found then
        return
    $f_t \leftarrow \max \left\{ f \leq f_t^{max} : T_{crit} > T + (f - f_t) \cdot \sum_{i \in A_t} \left( B_i^{-1} \cdot s_i^{pw} \right) \right\}$
    if $f_t$ not changed then
        return
    $T \leftarrow T + \delta \cdot \sum_{i \in A_t} \left( B_i^{-1} \cdot s_i^{pw} \right)$          ▷ update steady-state temperature
```

on the predicted power changes using the sensitivities of the power of all threads of the application. The loop iterates until the thermal violation in the steady-state temperature is resolved. Since the hotspot position may change, the boostability needs to be re-calculated in each iteration.

Boost to Exploit Thermal Margin Algorithm 6.3 exploits potential thermal margin. It selects the application with the highest boostability $b^{\uparrow}$ and then determines its maximum V/f level that does not result in a thermal violation *on any core*. It updates the steady-state temperatures according to the predicted power change from the sensitivity of the power. The loop terminates when the application with the highest boostability cannot be boosted further without violating the thermal constraint.

Throttle one Application to Boost Another Algorithm 6.4 addresses the case when there is no thermal margin but V/f levels are selected in a suboptimal way. In such a case, throttling an application with a low boostability $b^{\downarrow}$ may allow to boost another application with a high boostability $b^{\uparrow}$. If the differences between the applications are large, this potentially improves the overall system performance. The algorithm first selects the two applications with minimum and maximum boostability. Then, it iteratively searches for the highest V/f level for the boosted application that can be made thermally safe by throttling the other application and results in an improvement of the overall performance. Since this is the last step of the boosting optimization, the estimates of the steady-state temperatures do not need to be updated.

Computational Complexity The computational complexity of the initialization in Algorithm 6.1 is $O(n)$, where n is the number of cores, because at most n threads are executed on the system. The loop in Algorithm 6.2 can be executed at most once per application,

Algorithm 6.4 Throttle an Application to Boost Another Application

```
$h \leftarrow \arg\max_i T_i$                                ▷ position of the hotspot
**for each** application $j$ **do**
    $c_j \leftarrow \Delta f \cdot \sum_{i \in A_j} \left( s_i^{pw} \cdot B_{ih}^{-1} \right)$          ▷ reduction of $T_h$ per V/f level
    $b_j^{\uparrow} \leftarrow \frac{\min_{i \in A_j} s_i^{perf}}{c_j}$                  ▷ boostability (up) of application $j$
    $b_j^{\downarrow} \leftarrow \frac{\max_{i \in A_j} s_i^{perf}}{c_j}$                ▷ boostability (down) of application $j$
$t \leftarrow \arg\max_{j: f_j < f_j^{max}} b_j^{\uparrow}$                      ▷ application to boost (target)
$v \leftarrow \arg\min_{j: f_j > f_j^{min}} b_j^{\downarrow}$                     ▷ application to throttle (victim)
**if** no $v$ found **or** no $t$ found **or** $v = t$ **then**
    **return**                                               ▷ no transfer possible
**for** $f_t' \in \{f_t^{max}, f_t^{max} - \Delta f, \ldots, f_t + \Delta f\}$ **do**          ▷ try maximizing $f_t$
    $\Delta T_t \leftarrow (f_t' - f_t) \sum_{i \in A_t} \left( B_i^{-1} \cdot s_i^{pw} \right)$         ▷ $T$ change if boosting to $f_t'$
                    ▷ throttle victim to compensate target boosting (avoid thermal violation)
    $f_v' \leftarrow \max \left\{ f : \max \left( T + \Delta T_t - (f_v - f) \cdot \sum_{i \in A_v} \left( B_i^{-1} \cdot s_i^{pw} \right) \right) \leq T_{crit} \right\}$
                    ▷ check whether throttling to restore thermal safety is feasible
    **if** $f_v'$ exists **and** $f_v' \geq f_v^{min}$ **then**
        $\Delta\text{Perf} \leftarrow \left( \min_{i \in A_t} s_i^{perf} \right) \cdot (f_t' - f_t) + \left( \max_{i \in A_v} s_i^{perf} \right) \cdot (f_v' - f_v)$
        **if** $\Delta\text{Perf} > 0$ **then**
            $(f_t, f_v) \leftarrow (f_t', f_v')$               ▷ found good boosting solution, apply it
            **return**                                       ▷ stop search
```

and each iteration needs to update all elements of the steady-state temperature, which is each in $O(n)$, resulting in a complexity of $O(n^2)$ for Algorithm 6.2. Similarly, since the application to boost changes in every iteration, the loop in Algorithm 6.3 is also traversed at most once per application. Every iteration needs to calculate a steady-state for up to k V/f levels. In total, the complexity of Algorithm 6.3 is $O(kn^2)$. Finally, the complexity of the search in Algorithm 6.4 is $O(k^2 \cdot n^2)$ because the outer loop is executed $O(k)$ times and finding a single f_v' is in $O(k \cdot n^2)$. The number of V/f levels per epoch k is constant and does not depend on the number of cores. The overall complexity of *SmartBoost* is $O(n^2)$.

6.7 Experimental Evaluation

This section evaluates the accuracy of the sensitivity model, the overall performance gains with *SmartBoost* and its run-time overhead, and provides a comparison to the state-of-the-practice and state-of-the-art boosting techniques. We use the setup described in Section 3.1.1 for the experimental evaluation. It simulates a 64-core homogeneous many-core processor. The ambient and maximum temperature are set at 45 °C and 80 °C,

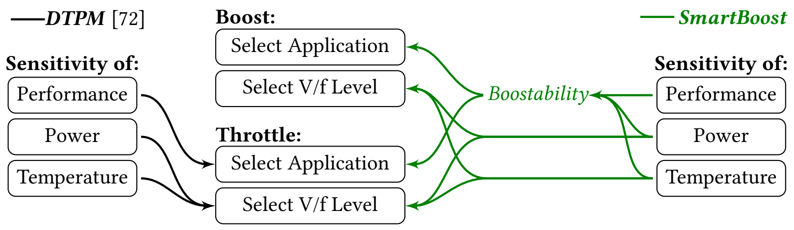


Figure 6.5: *SmartBoost* (right) *comprehensively* considers all the sensitivities, while the state of the art, *DTPM* [72], considers them only *partially*.

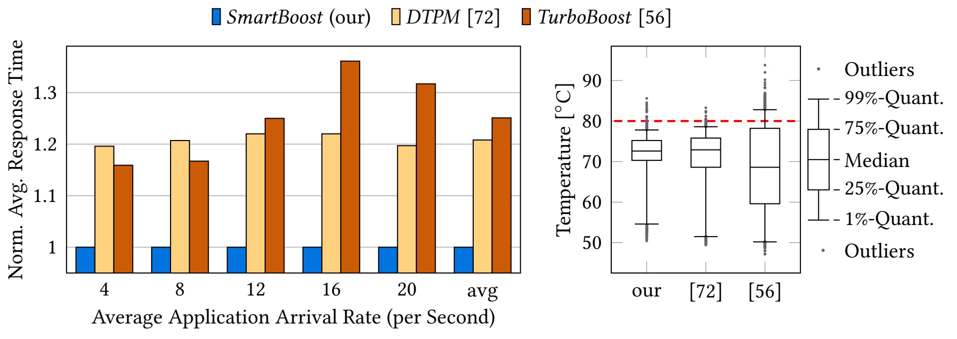
respectively. TDP is set at 100 W. Boosting is invoked every 1 ms [56]. We set the maximum V/f change per epoch at $k = 5$ levels, i.e., 500 MHz, to maintain a low prediction error. This value has been inferred from the model evaluation.

We execute the following applications from the *PARSEC* benchmark suite with *simmedium* input sizes, each at different levels of parallelism: *blackscholes*, *bodytrack*, *canneal*, *dedup*, *fluidanimate*, *streamcluster*, *swaptions* and *x264*. We also execute the following applications from the *SPLASH-2* suite with *large* inputs at different levels of parallelism: *barnes*, *cholesky*, *fft*, *fmm*, *lu.cont*, *lu.ncont*, *ocean.cont*, *ocean.ncont*, *radiosity*, *radix*, *raytrace*, *volrend*, *water.nsq*, and *water.sp*. These benchmark suites have realistic multi-threaded applications, designed to be representative by covering different domains such as computer vision, video encoding, or financial analyses.

SmartBoost is compared to two boosting techniques. The first technique is based on the state-of-the-practice Intel *TurboBoost* [56]. *TurboBoost* uses a simple heuristic that does not consider any application characteristics and boosts/throttles *all applications at the same time* based on whether there is a thermal violation in the transient temperatures on any core. The second technique is *DTPM* [72], which is the state-of-the-art performance-, thermal-, and power-aware boosting technique. It considers performance, power, and temperature by using analytical models of performance and power. However, the difference to our work is how these parameters are involved in the optimization, as shown in Fig. 6.5. When there is a thermal margin that allows boosting, *DTPM* does not consider any application characteristics and increases the V/f levels of all applications step by step until the peak V/f levels are reached or a thermal violation occurs. When throttling is required to mitigate a thermal violation, *DTPM* selects the application to throttle only based on the sensitivities of the performance. In contrast, *SmartBoost* uses our boostability metric, which integrates the sensitivities of performance, power, and temperature, for selecting the application in both boosting and throttling.

6.7.1 Comparison to the State of the Art

This section evaluates the overall performance gains with *SmartBoost*. This is the main experiment in this chapter. We evaluate all techniques on an open system with a random workload of 20 applications that are randomly selected from *PARSEC* and *SPLASH-2* with

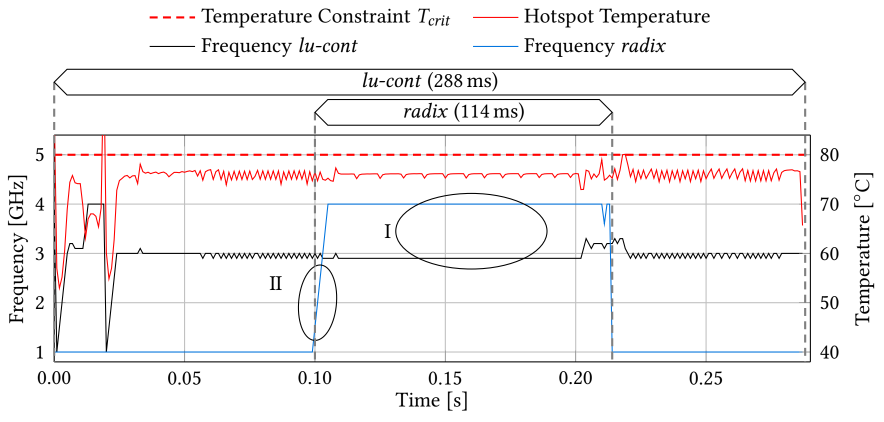


(a) Performance **(b)** Temperature

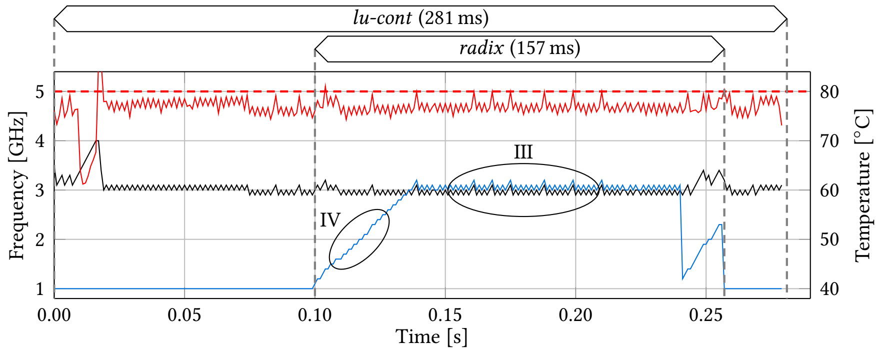
Figure 6.6: Comparing the overall system performance/temperature for a random workload of 20 applications at various arrival rates (thereby varying system utilization). *SmartBoost*'s average performance gains are 21 % and 25 % over the state-of-the-art *DTPM* [72] and the state-of-the-practice *TurboBoost* [56]. *SmartBoost* and *DTPM* only have <0.2 % thermal violations, while *TurboBoost* violates the temperature during 10 % of the execution.

random levels of parallelism. The arrival times of the applications are sampled from a Poisson distribution with varying arrival rate to reach different system utilization values. Thereby, the average system utilization throughout the execution time varies between 16 % and 44 %, while the resulting peak system utilization varies between 78 % and 100 %. The higher the utilization, the harder it gets for boosting techniques to improve the performance, as the thermal margin decreases. To enable a fair comparison and to be independent of a specific mapping technique, we employ the same random mapping for all techniques.

Fig. 6.6a and Fig. 6.6b show the average response time and hotspot temperature distribution with the three techniques, respectively. We calculate the geometric mean of response times to give the same weight to each application despite their widely-varying individual response times, which range from 56 ms to 1.6 s. *SmartBoost* outperforms the state-of-the-art *DTPM* and the state-of-the-practice *TurboBoost* by 21 % and 25 %, on average. *TurboBoost* has the lowest performance because it employs a simple heuristics that ignores the application characteristics. It additionally violates the temperature constraint during 10 % of the execution because it reacts on the transient temperature, which may be too slow to prevent thermal violations. *SmartBoost* and *DTPM* both make decisions based on the steady-state temperature. *SmartBoost* additionally proactively determines the thermal impact of a V/f change before executing it. Consequently, it violates the temperature during less than 0.2 % of the execution. *SmartBoost* achieves the highest performance by considering all relevant parameters in all phases of the boosting optimization. In addition, *SmartBoost* throttles applications to boost another if it increases the overall performance. These smart decisions let *SmartBoost* outperform the state of the art.



(a) *SmartBoost*



(b) *DTPM* [72]

Figure 6.7: An illustrative example showing the benefits of *SmartBoost*. It selects better V/f levels (Windows I and III), and enables faster increase of the V/f levels (Windows II and IV) than the state-of-the-art *DTPM* [72].

6.7.2 Illustrative Example

To provide deeper insights into the reasons for the benefits of *SmartBoost* compared to the state-of-the-art *DTPM*, Fig. 6.7 visualizes the impact of their decisions on both performance and temperature based on an illustrative example, in which two applications with different characteristics run in parallel. The two applications are *lu.cont* and *radix*, as in the motivational example in Section 6.1. Both applications have a high sensitivity of the performance but *lu.cont* has much higher absolute power and sensitivity of the power than *radix*. Therefore, its core forms the thermal hotspot, which also leads to a higher sensitivity of the temperature. Before *radix* arrives at the system, both techniques employ similar V/f levels. Two main differences can be observed during the execution of *radix*, which arrives after 100 ms. Firstly, *DTPM*, which only considers the sensitivity of the performance to select applications, employs a similar V/f level for both applications

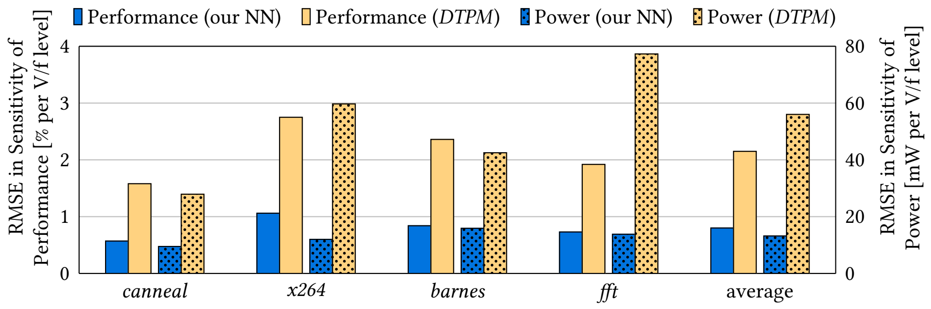


Figure 6.8: Prediction RMSE in the sensitivity of performance and power for both our NN model and for the models of *DTPM* [72].

(Window III). In contrast, *SmartBoost* exploits that *radix* has lower sensitivities of both power and temperature, and slightly throttles *lu.cont* to be able to boost *radix* to the maximum V/f level (Window I). Secondly, *DTPM* does not use information about the power during boosting and increases the V/f level step by step, which is slow (Window VI). *SmartBoost* can increase the V/f level much faster (Window II) because it employs the sensitivity of power and temperature to proactively determine the thermal safety of a V/f level increase. These differences result in a significantly higher overall performance. *Radix* executes 38 % faster, while *lu.cont* experiences only a 2 % slower execution.

6.7.3 Prediction Accuracy of our Sensitivity Model

We evaluate the accuracy of our sensitivity model only for the unseen applications, i.e., applications that are not used in the training phase. These are *x264* and *canneal* from *PARSEC*, and *barnes* and *fft* from *SPLASH-2*. These applications have been selected as they are representative for various characteristics. They comprise applications that are memory-intensive (e.g., *canneal*) or compute-intensive (e.g., *barnes*), have high IPS (e.g., *fft*) or low IPS (e.g., *barnes*), have few long-running phases (e.g., *canneal*) or fast-changing phases (e.g., *x264*). We compare our model to the analytical performance and power models used in *DTPM*. *DTPM* employs a linear performance model that divides the execution of a workload into a V/f level-dependent and a V/f level-independent part. The CPU utilization determines the V/f level-dependent part. The power model of *DTPM* independently models voltage-dependent leakage power and V/f-dependent dynamic power. While leakage power is modeled in detail, it uses V^2f scaling of the dynamic power assuming that the switching activity does not change with the V/f level.

The comparison of the model errors is shown in Fig. 6.8, in terms of % per V/f level and mW per V/f level for predictions of the V/f sensitivities of performance and power, respectively. For the sensitivity of performance, the RMSE of our NN ranges from 0.57 %/level to 1.06 %/level. The average is 0.80 %/level. In comparison, the average RMSE of the model of *DTPM* is 2.15 %/level. Similar improvements can be observed for the power, where the

NN model of *SmartBoost* achieves an average error of 13 mW/level, while the average error of the model of *DTPM* is 56 mW/level. These experiments demonstrate the benefits of a more powerful NN-based model over simpler analytical models. At the maximum V/f change of $k = 5$ levels, the prediction error of our NN is 4.0 % and 66 mW for performance and power, respectively. Such low errors allow *SmartBoost* to make boosting decisions that maximize the performance while achieving thermal safety.

6.7.4 Run-time Overhead

Finally, we evaluate the run-time overhead of *SmartBoost*. Since the run-time overhead depends on the number of active applications and cores, we consider the highest application arrival rate of 20 applications/s, which represents the hardest case, i.e., the overhead would be lower in an average scenario. The average run-time overhead is 8.1 µs if *SmartBoost* is only executed on a single core. More than half of the overhead (4.6 µs) originates in the inference of the sensitivity model. The remainder is required for the three boosting phases shown in Algorithms 6.2 to 6.4. Since boosting is invoked every 1 ms, the resulting total overhead is 0.81 %. Note that the policy is using only one core out of 64. The overhead within the overall system is, therefore, negligible. As discussed in Section 6.6, the complexity of *SmartBoost* is $O(n^2)$, i.e., the overhead grows slowly with a larger number of cores than the studied 8×8 cores.

6.8 Summary

This chapter presented *SmartBoost*, which is a boosting technique that optimizes the performance of thermally-constrained many-core processors using DVFS. This is a complex problem that requires to jointly consider the sensitivities of performance, power, and temperature within the boosting optimization. Moreover, the sensitivities of performance and power depend on the compute-intensiveness and instruction mix of applications, and can not be directly measured at run time. We tackle the first challenge algorithmically with a novel *boostability* metric that integrates all three sensitivity values into a single metric that can be used for the optimization. The second challenge is tackled with ML by training an NN model that estimates the application-dependent sensitivities of performance and power at run time based on performance counter readings. The estimations of the sensitivity of the power and temperature also enable proactive management, in which the impact of a V/f change on the temperature is estimated before executing the change. *SmartBoost* significantly increases the performance of mixed workloads with unseen applications compared to both simple heuristic management and state-of-the-art analytical management.

7 Learning Optimal Management with Imitation Learning

Elevated on-chip temperature accelerates aging mechanisms in processors, and thereby degrades the system reliability. This has been tackled in the previous sections by enforcing a thermal constraint. However, in mobile devices, elevated on-chip temperature also may adversely affect the user experience by increasing the device's skin temperature [139]. This makes temperature minimization of paramount importance. The two main means to manage the temperature are application migration, to dynamically change the mapping of applications to cores, and DVFS. However, using these means without considering the QoS targets, i.e., their required performance, of applications also degrades the user experience [140].

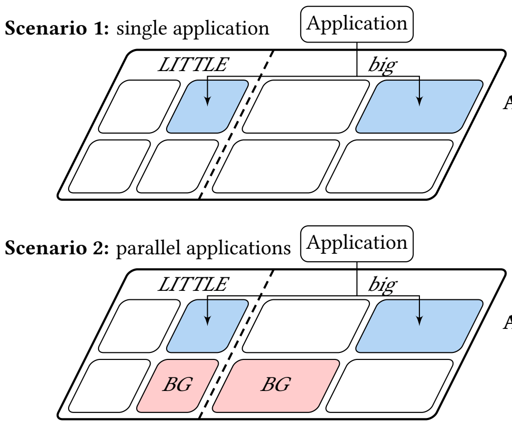
This section develops a lightweight run-time application migration and DVFS technique *TOP-IL* to minimize the temperature under QoS targets. This technique employs the pattern to directly learn resource management decisions, as introduced in Section 1.4.3, because this allows to tackle the involved challenges, as will be discussed in Section 7.2. We target heterogeneous multi-core processors, which are commonly employed in mobile devices. To this end, we use the evaluation platform described in Section 3.2, which features a smartphone SoC with an Arm big.LITTLE CPU. We compare *TOP-IL* to both simple heuristic management and RL-based management.

Thermal optimization without considering the application characteristics of all running applications misses significant optimization opportunities. The reason is that the impact on performance and power when migrating an application between clusters differs from one application to another, as illustrated in Fig. 1.1 for the performance. Similarly, the sensitivities of performance and power to DVFS also vary between applications, which has been exploited already in Chapter 6. The following motivational example demonstrates the role of application characteristics for thermal optimization.

7.1 Motivational Examples

In Scenario 1 in Fig. 7.1, one application, either *adi* or *seidel-2d* from the *Polybench* [141] suite, is executed on an Arm big.LITTLE CPU. Their QoS target is selected as 30 % of the

This chapter is mainly based on [1, 2].


Scenario 1: single application
Application
LITTLE
big
Scenario 2: parallel applications
Application
LITTLE
big
BG
BG

Application	***LITTLE***	***big***
adi	48.1 °C	45.7 °C
seidel-2d	45.1 °C	46.8 °C

Application	***LITTLE***	***big***
adi	58.9 °C	59.6 °C

Figure 7.1: On Arm big.LITTLE, the optimal mapping of applications with QoS targets varies between applications, and with other parallel applications (BG). The clusters are operated at the lowest V/f levels that still satisfy all QoS targets.

performance, measured in IPS, that is reached at the highest V/f level on the *big* cluster. The clusters are operated at the lowest V/f levels that satisfy the QoS target. Intuitively, executing the applications on the *LITTLE* cluster should minimize the temperature, as the in-order CPUs on the *LITTLE* cluster are optimized for energy efficiency. However, this is not always the case. For *adi*, executing it on the *big* cluster instead minimizes the temperature. The reason is that *adi* requires 1.8 GHz when mapped to the *LITTLE* cluster to reach its QoS target, but only 0.7 GHz on the *big* cluster. In contrast, *seidel-2d* reaches its QoS target already at 1.2 GHz on the *LITTLE* cluster, and requires 1.0 GHz on the *big* cluster. This results in a similar temperature on both clusters, with a small advantage of the *LITTLE* cluster. The reason for the different V/f level requirements of the two applications at different clusters is that the applications benefit differently from the out-of-order execution and larger caches on the *big* cluster. Consequently, the different application characteristics render different mappings optimal. *Optimal thermal management needs to consider the characteristics of the running applications and their QoS targets.*

Scenario 2 studies *adi* with the same QoS target as in Scenario 1 but now, additional background applications with high QoS targets are running on both clusters. Intuitively, mapping *adi* to the *big* cluster should still minimize the temperature, as in Scenario 1. However, both clusters need to be operated at the peak V/f levels to satisfy the QoS targets of the background applications. Since this platform has per-cluster DVFS, *adi* is also executed at the peak V/f level. In this case, mapping *adi* to the *LITTLE* or *big* cluster results in almost the same temperature, unlike what has been observed in Scenario 1. Hence, per-cluster DVFS affects the optimal mapping w.r.t. temperature when several applications run in parallel. *Optimal thermal management needs to perform global optimization considering the characteristics and QoS targets of all running applications.*

7.2 Challenges and Novel Contributions

There are several challenges in temperature optimization on heterogeneous multi-core processors under QoS targets. Firstly, as has been discussed in Section 1.2, there is high complexity in all involved aspects of the platform and applications. As shown in Fig. 1.1, the performance of applications depends on their instruction sequence, the CPU microarchitecture and the memory architecture. In addition, it is greatly affected by the V/f levels. The power consumption of applications also depends on all these factors. The temperature depends on the power density, floorplan, and cooling. Secondly, as has also been discussed in Section 1.2, the workload, i.e., the executed applications and their arrival times, is commonly not known at design time. Therefore, the management policy must not be specific to selected applications but achieve good management for any unseen workload. Thirdly, per-cluster DVFS runs all applications on the same cluster at the same V/f level, requiring global optimization. Finally, there is limited access to physical measurements of the platform. For instance, most platforms, such as the one studied in this chapter, feature no power sensors and only few temperature sensors.

Some of these challenges can be solved with models for individual aspects such as power, performance, or temperature. For instance, the techniques presented in Chapters 5 and 6 employed ML models for performance and power. Other techniques based on analytical and ML models are discussed in Sections 2.1.3 and 2.2.3, respectively. However, building such models requires access to measurements of processor-internal properties like power, which may not be available on any platform. This can be solved by performing end-to-end learning of resource management decisions based on the available measurements, as has been introduced in Section 1.4.3. The two main methods to achieve this are RL and IL. In both methods, NN learning can be used to cope with the high platform and application complexity.

As discussed in Section 2.2.1, RL suffers from several problems. Instability in the learning may lead to suboptimal management and online learning may lead to a high run-time overhead. However, run-time thermal minimization under QoS targets requires a lightweight, yet near-optimal optimization to effectively minimize the temperature, and a stable policy to avoid abrupt QoS violations and jumps in the temperature. *IL is the only method that provides these capabilities.* In particular, it enables using the optimality of an oracle policy, yet at low run-time overhead, by design-time training of an NN model from oracle demonstrations. Training at design time until convergence also provides stability. However, since IL does not perform run-time retraining, the model must be trained such that it is capable to cope with the different unseen scenarios that may happen at run time. This includes most notably different workloads, but also different cooling settings.

Motivated by the advantages of IL, researchers have started to apply IL in resource management, as discussed in Section 2.2.2, but they all target power or energy optimization. This significantly differs from temperature optimization, which is subject to spatial (heat transfer) and temporal (heat capacity) effects that do not exist in power/energy. The work presented in this chapter is the first to employ IL for temperature optimization.

Few works have proposed their own specific ML accelerators [77, 142] to accelerate ML-based resource management. However, relying on a specific accelerator incurs additional area overhead to the used platform and limits the applicability of the techniques to platforms that feature this specific accelerator. Recently, generic NN accelerators, e.g., NPUs or digital signal processors (DSPs), became common in end devices such as smartphones [143] to increase the performance and energy-efficiency of user applications that perform NN inference. Despite their increasing spread and benefits, these existing accelerators have never been used to speed up NN-based resource management, and the work presented in this chapter is the first to do that.

This chapter makes the following novel contributions:

- We design, train, and employ NN-based IL for temperature optimization under QoS targets, as it enables near-optimal decisions at a low run-time overhead. Our solution, *TOP-IL*, employs application migration and DVFS on heterogeneous multi-core processors.
- We accelerate *TOP-IL* using an existing generic NN accelerator (an NPU) on a real platform.
- We develop an RL-based thermal optimization technique and show that IL outperforms RL in terms of achieving the target objective and run-time stability.
- We demonstrate that the learned policy with IL generalizes to unseen workloads and different cooling settings than what is used during training.

7.3 Problem Definition

This chapter targets a heterogeneous multi-core processor with per-cluster DVFS, where $\mathcal{F}_x$ is the list of frequencies of cluster x and f_x is its current V/f level. The platform comprises two clusters, *LITTLE* and *big*, i.e., $x \in \{l, b\}$, but our solution is compatible with any number of clusters. The platform executes parallel applications, each with its own QoS target Q_k and current QoS q_k, which are expressed in terms of the IPS. The processor is employed in an open system, where a priori unknown applications arrive at a priori unknown times. Our solution does not rely on run-time power measurements, as they are often not available on real-world platforms [121].

The objective is to minimize the on-chip temperature, while maintaining the QoS of all running applications. This is achieved by dynamically changing the application-to-core mapping via application migration, and by per-cluster DVFS.

We split the problem into two parts: 1) dynamic application-to-core mapping (via application migration), and 2) per-cluster DVFS. Decisions on application migrations are made with NN-based IL. DVFS is implemented using a simple control loop. While it would be intuitive to train a single NN for both application migration and DVFS, performing only migration with the model reduces the complexity of creating training data, the NN topology, and the inference overhead. V/f level information is provided as an input for

Table 7.1: The selected features for IL-based migration cover a) the characteristics of the AoI, b) its QoS target, and c) information about other applications (background).

Feature	Count	Feature	Count
AoI QoS (a)	1	AoI QoS target (b)	1
AoI L2D accesses (a)	1	$\tilde{f}_{x\setminus AoI}/f_x$ (c)	2
AoI current mapping (a)	8	Core utililizations (c)	8

migration decisions to still achieve near-optimal decisions. We accelerate the run-time inference of the NN with an NPU. The design-time training and run-time migration technique are described in Sections 7.4 and 7.5.1, respectively. Section 7.5.2 describes the DVFS control loop.

7.4 Imitation Learning-Based Application Migration

Employing IL requires to select features, create oracle demonstrations, and train the NN model that is used at run time. These steps are described in the following sections.

7.4.1 Feature Selection

The features need to accurately describe the state of the platform and applications to be able to make near-optimal migration decisions, and need to be observable at run time. As shown in the motivational example in Fig. 7.1, the optimal mapping of an application of interest (AoI) depends on a) its characteristics, which affect its power and performance on the different clusters, b) its QoS target, which determines the suitable clusters and the required V/f levels, and c) other (background) applications, which determine the available cores, the required V/f levels per cluster to satisfy the QoS targets of the background applications, and affect the temperature distribution.

Table 7.1 lists the selected features. They cover all three aspects (a-c) discussed above. The characteristics of the AoI (a) comprise its current QoS, measured in IPS, and its number of L2-D cache accesses per second. The cache accesses indicates the memory-intensiveness of the AoI. We use the Linux *perf* API to read hardware performance counters for IPS and L2-D accesses. The current mapping of the AoI provides information about its current core and cluster, and thereby provides context to the performance counter readings. It is represented as one-hot encoding of all cores (in our platform: 8 cores). The QoS target (b) is represented in terms of the required IPS. The background (c) is represented by the current per-core utilization values, as well as by the estimated per-cluster V/f changes if the AoI would not be executed. The latter indicate potential temperature reductions on the cluster if the AoI would be migrated to another cluster. This is calculated by first estimating the minimum V/f level $\tilde{f}_{k,min}$ of each running application k that is required to satisfy its QoS target Q_k. During design-time training data generation, $\tilde{f}_{k,min}$ can be determined from execution traces. At run time, no traces from other V/f levels are available, as we target unknown applications. Linear scaling from the current V/f level $f_{x(k)}$ of its cluster $x(k)$ is performed instead:

$$\tilde{f}_{k,min} = \min\left\{f \in \mathcal{F}_{x(k)} : q_k \cdot f / f_{x(k)} \geq Q_k\right\} \tag{7.1}$$

This estimate is calculated based on the current QoS q_k in the current execution phase of application k, i.e., $\tilde{f}_{k,min}$ does not need to be known at design time and may change over time. Finally, the required V/f level without the AoI is determined per cluster x as the maximum among all other applications running on it:

$$\tilde{f}_{x\backslash AoI} = \max\left\{\tilde{f}_{k,min} : \text{app. } k \text{ mapped to } x \land k \neq AoI\right\} \tag{7.2}$$

All features are normalized to be usable with an NN.

7.4.2 Oracle Demonstrations (Training Data)

The training data need to indicate the optimal migration w.r.t. temperature and QoS for a variety of scenarios. To this end, we collect measurements of temperature and hardware performance counters (traces) of benchmark applications in various scenarios and extract training data from these traces.

Collect Traces The upper part of Fig. 7.2 depicts the process to collect traces. Since this is the most time-consuming part of training, redundant executions must be avoided. The straightforward approach to collect traces would be to select a scenario, i.e., a combination of AoI, its QoS target, and background applications with QoS targets, and execute it once per mapping of the AoI to each free core, which is not occupied by the background. However, this results in redundant executions. The reason is that with per-cluster DVFS, only the application with the highest QoS target, i.e., the highest required V/f level, determines the V/f level of the cluster. As a result, scenarios that differ only in the QoS, may result in the same selected V/f levels, and, therefore, the same execution.

We avoid redundancy by first obtaining traces for different combinations of per-cluster V/f levels and only afterwards select different QoS targets that match the V/f levels to create training data. This approach requires a constant QoS of the benchmarks that are used to create the training data, i.e., no execution phases. At run time, TOP-IL supports also applications with execution phases, and we use such applications in the experimental evaluation. To further accelerate collecting traces, we stop traces when the AoI has executed 10^{10} instructions, which is large enough to observe significant differences in the temperature between traces but still reduces the time to collect each trace. At run time, *TOP-IL* supports applications with any number of executed instructions. Finally, we obtain traces for a reduced set of V/f levels, but still use all available V/f levels at run time. We start the background applications 2 min before starting the AoI to ensure a consistent initial temperature. We randomize the order of executions to avoid any remaining systematic error. We use active cooling with a fan while collecting these traces because it prevents triggering dynamic thermal management (DTM), which would throttle

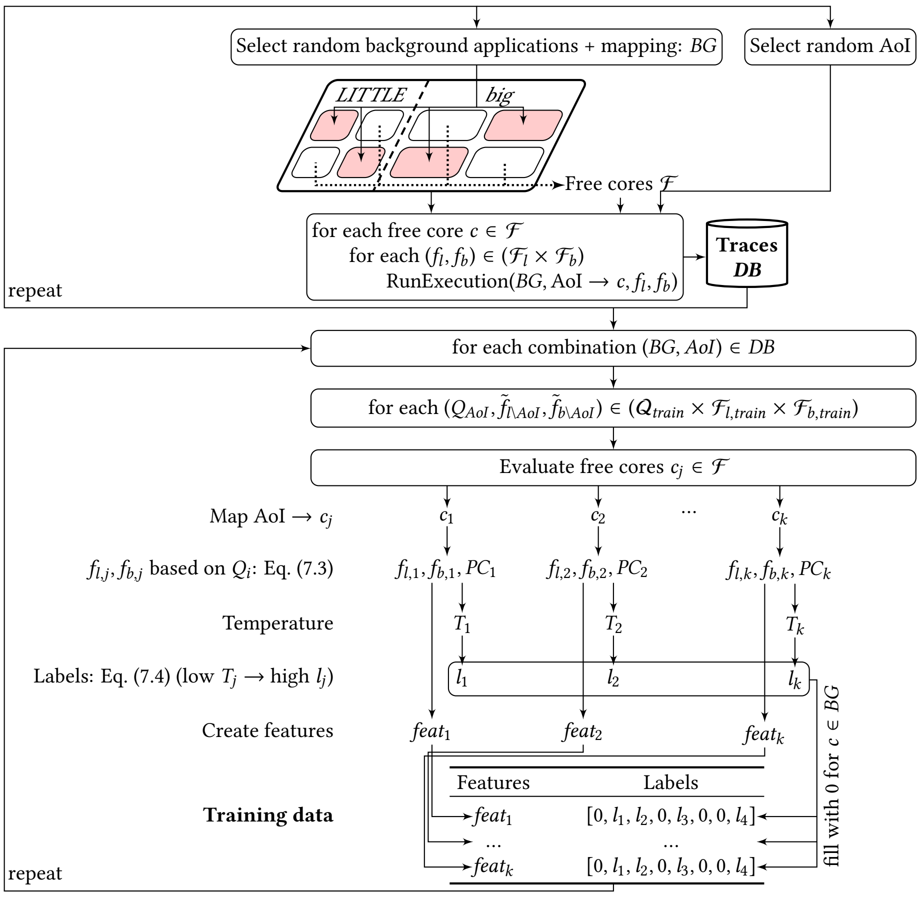

Select random background applications + mapping: BG
Select random AoI
LITTLE
big
Free cores $\mathcal{F}$
for each free core $c \in \mathcal{F}$
for each $(f_l, f_b) \in (\mathcal{F}_l \times \mathcal{F}_b)$
RunExecution(BG, AoI $\rightarrow c, f_l, f_b$)
Traces DB
repeat
for each combination (BG, AoI) $\in DB$
for each $(Q_{AoI}, \tilde{f}_{l\setminus AoI}, \tilde{f}_{b\setminus AoI}) \in (Q_{train} \times \mathcal{F}_{l,train} \times \mathcal{F}_{b,train})$
Evaluate free cores $c_j \in \mathcal{F}$
Map AoI $\rightarrow c_j$
$f_{l,j}, f_{b,j}$ based on Q_i: Eq. (7.3)
Temperature
Labels: Eq. (7.4) (low $T_j \rightarrow$ high l_j)
Create features
Training data
c_1 c_2 ... c_k
$f_{l,1}, f_{b,1}, PC_1$ $f_{l,2}, f_{b,2}, PC_2$ $f_{l,k}, f_{b,k}, PC_k$
T_1 T_2 T_k
l_1 l_2 l_k
$feat_1$ $feat_2$ $feat_k$
Features Labels
$feat_1$ $[0, l_1, l_2, 0, l_3, 0, 0, l_4]$
... ...
$feat_k$ $[0, l_1, l_2, 0, l_3, 0, 0, l_4]$
fill with 0 for $c \in BG$
repeat

Figure 7.2: Design-time training data generation for IL-based application migration.

the V/f levels unpredictably, polluting the training data. The evaluation shows that the trained NN can be used for different cooling, i.e., without a fan, without retraining.

Figs. 7.3a and 7.3b present an illustrative excerpt of the collected traces including the AoI performance and the peak temperature for a single selection of background applications and AoI (*seidel-2d*). In this example, only the two cores 3 and 6 are free, while the other cores are running background applications.

Extract Training Data The lower part of Fig. 7.2 shows the steps to extract training data from the collected traces. We first select a combination of background and AoI from the traces. Then, we sweep the values of the QoS targets of AoI and background. The QoS target Q_{AoI} of the AoI is represented explicitly in terms of the IPS, while the QoS targets of the background is represented by the required V/f levels per cluster $\tilde{f}_{l\setminus AoI}, \tilde{f}_{b\setminus AoI}$. Next,

Performance (MIPS)		f_b 0.7 GHz	1.2 GHz	1.5 GHz	…
f_l	0.5 GHz	137	140	139	…
	1.4 GHz	366	363	373	…
	1.8 GHz	471 I	478	479	…

Peak Temp. (°C)		f_b 0.7 GHz	1.2 GHz	1.5 GHz	…
f_l	0.5 GHz	35.8	42.3	50.7	…
	1.4 GHz	40.5	46.2	53.7	…
	1.8 GHz	42.5 I	49.6	56.1	…

(a) Trace results (running AoI on core 3)

Performance (MIPS)		f_b 0.7 GHz	1.2 GHz	1.5 GHz	…
f_l	0.5 GHz	256	455	563 II	…
	1.4 GHz	255	455 I	563	…
	1.8 GHz	256	454	562	…

Peak Temp. (°C)		f_b 0.7 GHz	1.2 GHz	1.5 GHz	…
f_l	0.5 GHz	38.0	46.2	52.2 II	…
	1.4 GHz	38.4	46.6 I	56.5	…
	1.8 GHz	39.5	48.8	57.0	…

(b) Trace results (running AoI on core 6)

			AoI on core 3			AoI on core 6			Labels								
Q_{AoI} (MIPS)	$\tilde{f}_{l\setminus AoI}$ (GHz)	$\tilde{f}_{b\setminus AoI}$ (GHz)	$f_{l,3}$ (GHz)	$f_{b,3}$ (GHz)	T_3 (°C)	$f_{l,6}$ (GHz)	$f_{b,6}$ (GHz)	T_6 (°C)	l_0	…						l_7	
400	1.4	0.7	1.8	0.7	42.5	1.4	1.2	46.6	0	0	0	1.00	0	0	0.02	0	I
200	1.4	1.2	1.4	1.2	46.2	1.4	1.2	46.6	0	0	0	1.00	0	0	0.65	0	
400	0.5	1.5	1.8	1.5	56.1	0.5	1.5	52.2	0	0	0	0.02	0	0	1.00	0	
500	0.5	0.7	–	0.7	–	0.5	1.5	52.2	0	0	0	−1	0	0	1.00	0	II
…	…	…	…	…	…	…	…	…	…								

(c) Examples for calculating the labels

		Features						Labels		
f_l (GHz)	f_b (GHz)	q_{AoI} (MIPS)	Q_{AoI} (MIPS)	AoI curr. map.	Core utils.	$\tilde{f}_{l\setminus AoI}/f_l$	$\tilde{f}_{b\setminus AoI}/f_b$	l_3	l_6	
1.8	0.7	471	400	0 0 0 1 0 0 0 0	1 1 1 0 1 1 0 1	0.76	1.00	1.00	0.02	I
1.4	1.2	455	400	0 0 0 0 0 0 1 0	1 1 1 0 1 1 0 1	1.00	0.56	1.00	0.02	I
1.8	0.7	471	500	0 0 0 1 0 0 0 0	1 1 1 0 1 1 0 1	0.28	1.00	−1	1.00	II
0.5	1.5	563	500	0 0 0 0 0 0 1 0	1 1 1 0 1 1 0 1	1.00	0.46	−1	1.00	II
…	…	…	…	…	…	…	…	…		

(d) Training data examples

Figure 7.3: Illustrative example of training data generation. Only cores 3 and 6 are free for the AoI. (a) and (b) show the trace results (AoI performance and peak temperature) for the two free cores and selected combinations of frequencies f_l and f_b. (c) demonstrates the calculation of the labels for a given AoI QoS target Q_{AoI} and minimum required frequency to satisfy the QoS of the background ($\tilde{f}_{l\setminus AoI}$ and $\tilde{f}_{b\setminus AoI}$). For each mapping, first the minimum frequencies that satisfy all QoS targets are selected to determine the corresponding temperature for each potential mapping. Labels are calculated with Eq. (7.4). (d) lists selected training examples.

we need to find the corresponding trace when mapping the AoI on core j with the selected QoS targets. The V/f levels f_l, f_b of this trace are the lowest V/f levels to satisfy all QoS targets Q_{AoI}, $\tilde{f}_{l\setminus AoI}$, and $\tilde{f}_{b\setminus AoI}$:

$$f_l, f_b = \underset{f_l', f_b'}{\arg\min} \left(f_l' \geq \tilde{f}_{l\setminus AoI} \ \wedge \ f_b' \geq \tilde{f}_{b\setminus AoI} \ \wedge \ q_{AoI}(f_l', f_b') \geq Q_{AoI} \right) \tag{7.3}$$

Next, the peak temperature for each mapping of the AoI to each free core j is determined from these traces. We observe that in many cases, several mappings result in a similar temperature. An example are mappings to different *LITTLE* cores. In our experiments, there is on average one additional mapping that is within 1 °C of the temperature obtained with the optimal mapping. Therefore, we create a soft label $l_j \in [0,1]$, indicating the quality of mapping the AoI to core j:

$$l_j = \begin{cases} 0 & \text{core } j \text{ occupied by a background application} \\ -1 & \text{core } j \text{ cannot meet } Q_{AoI} \\ e^{-\alpha \cdot (T_j - \min_{j'} T_{j'})} & \text{otherwise} \end{cases} \tag{7.4}$$

Cores that run a background application get $l_j = 0$. Mappings that violate the QoS target of the AoI at the highest V/f level get $l_j = -1$. The mapping with the lowest temperature gets $l_j = 1$. For all other mappings, the higher the temperature is compared to the optimum, the closer l_j gets to 0. The parameter α determines a trade-off between tolerating higher temperatures and susceptibility to temperature measurement noise. We empirically set $\alpha = 1$. Fig. 7.3c lists selected illustrative examples. For instance, when selecting $Q_{AoI} = 400 \cdot 10^6$ IPS, $\tilde{f}_{l \backslash AoI} = 1.4$ GHz, and $\tilde{f}_{b \backslash AoI} = 0.7$ GHz (Line I), the minimum frequencies of the *LITTLE*/*big* cluster to satisfy all QoS targets are 1.8 GHz/0.7 GHz and 1.4 GHz/1.2 GHz for a mapping of the AoI to cores 3 and 6, respectively. The respective temperatures are 42.5 °C and 46.6 °C, i.e., a mapping to core 3 is cooler. Therefore, the labels for cores 3 and 6 are 1 and 0.02, respectively. Fig. 7.3c also lists examples where the two cores result in a similar temperature, where core 6 is beneficial, and where core 3 cannot reach the QoS target of the AoI at the highest V/f levels (Line II).

After creating the labels, the features that describe the execution of the AoI with the selected QoS and background are determined from the traces as described in Section 7.4.1. One training example is created for each free core, on which the AoI could be executed at run time, i.e., each source of a migration. Fig. 7.3d illustrates this with a few examples for the examples I and II discussed earlier. By creating one training example for *every* free core for each selection of Q_{AoI}, $\tilde{f}_{l \backslash AoI}$, and $\tilde{f}_{b \backslash AoI}$, the training data is already exhaustive because the policy is trained to recover from each potential mapping of the AoI at run time. This is the reason why the *DAgger* [89] algorithm, which initially only trains the policy on the optimal sequence of management decisions, and only gradually adds training data to recover from suboptimal decisions to increase the robustness of the model, is not required. 19,831 training examples are created from 100 combinations of AoI and background.

7.4.3 IL Model Creation and Training

We employ a fully-connected NN model and find the optimal topology (number of layers and neurons) by neural architecture search (NAS). Fig. 7.4 shows the result of the grid search to determine the depth (number of layers) and width (number of neurons per layer) of the NN. The optimal topology uses 4 hidden layers with 64 neurons, each. The hidden layers use ReLU activation, the output layer with 8 neurons does not use any activation

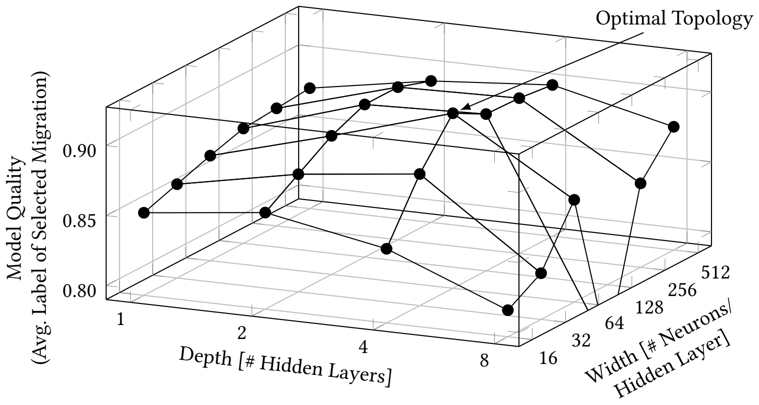


Figure 7.4: Visualization of the grid search results for the NN topology. The optimal topology uses 4 hidden layers with 64 neurons, each.

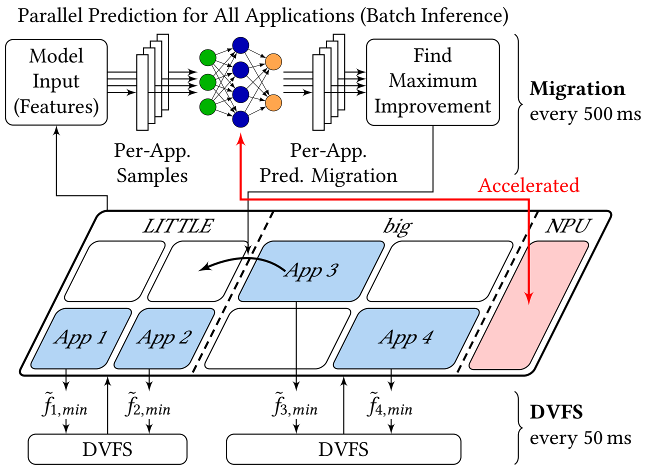


Figure 7.5: Illustration of *TOP-IL* at run time. IL-based application migration uses the NPU to accelerate the inference for predicting the best migration per each application.

function. We use *Adam* optimizer with momentum, where the exponentially decaying learning rate is set at $0.01 \cdot 0.95^{(epoch)}$. We use mean squared error (MSE) loss and stop the training when the model has not improved for 20 epochs (early stopping). Three models are trained that are initialized with different random seeds. This allows to demonstrate that the training is robust to the weight initialization, as will be shown in the evaluation.

7.5 Run-Time Temperature and QoS Management

The run-time part of *TOP-IL* integrates the IL-based application migration with a per-cluster DVFS control loop. It is illustrated in Fig. 7.5.

7.5.1 Application Migration with NPU-Accelerated IL

If K applications run in parallel, each should be migrated to its optimal core w.r.t. temperature and QoS considering all other applications. However, migrating several applications at once results in a high number of potential combinations to chose from and the impact of several migrations at once would be difficult to predict. We solve this by migrating only one application at a time, but we find in each control epoch the optimal migration among all possible migrations of all applications. The NN migration model has been trained for one AoI, which is migrated, and several other background applications, i.e., it requires selecting an AoI. Therefore, we perform parallel inference, where each application is used as the AoI once. The inference output is a matrix, where each entry $l_{k,c}$ contains the rating of mapping application k to core c. The optimal migration maximizes the improvement in the rating compared to the current mapping $c(k)$:

$$\hat{k}, \hat{c} = \arg\max_{k',c'} \left(l_{k',c'} - l_{k',c(k')} \right) \tag{7.5}$$

The result of this optimization is to migrate application $\hat{k}$ to core $\hat{c}$ if it is not already mapped to it. The migration policy is executed each 500 ms. This is fast enough to adapt to potential workload phases of the applications, which run for several minutes, but still allows to maintain a low overhead.

To further reduce the overhead of the NN inference, we employ the existing NPU of the *Kirin 970* SoC. The available parallelism in the NPU allows performing the parallel inference for all applications simultaneously in a single batch. The NPU is accessible via the *HiAI DDK*, which originally is designed for Java applications, i.e., to speed up user apps. We develop a C++ binary that runs in user space, uses the Linux *perf* API to read the hardware performance counters, and uses the `/proc` filesystem to read information about running applications and the system utilization. It employs the NPU for inference via the *HiAI DDK* with a non-blocking call, and uses the Linux *affinity* feature to migrate applications.

Since, we perform migration each 500 ms, the migration overhead, e.g., for transferring the context to the target core or due to cold caches, is negligible. We perform experiments to quantify the worst-case overhead of application migration, i.e., periodically migrating an application between the *big* and *LITTLE* cluster in each migration epoch. The migration overhead m is calculated by:

$$m = \frac{\frac{1}{2} \cdot \left(\frac{1}{t_{\text{big}}} + \frac{1}{t_{\text{LITTLE}}} \right)}{\frac{1}{t_{\text{migrate}}}} - 1 \tag{7.6}$$

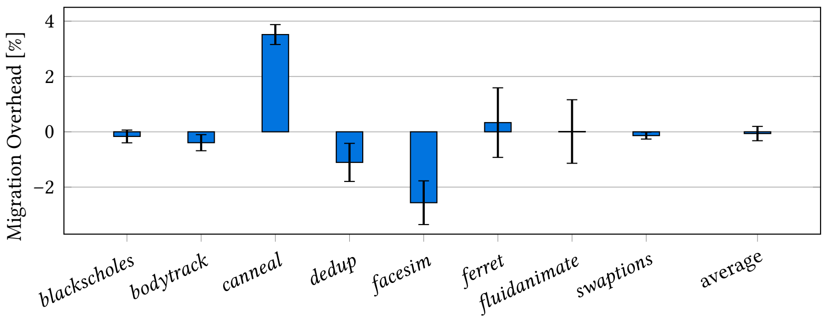


Figure 7.6: The impact of periodic migration of an application on its performance is negligible.

The numerator calculates the average performance of the *big* and *LITTLE* clusters, while the denominator represents the measured performance with periodic migration. We repeat this experiment three times with several *PARSEC* applications. Fig. 7.6 plots the average and standard deviation of the migration overhead per application. The overhead differs between applications because of their different cache intensity. Some applications, i.e., *dedup*, *facesim*, experience a negative overhead, which we interpret as follows. If an application has different execution phases that benefit differently from the features of the *big* cluster, potential correlation between the migration epoch and the execution phases improves the performance, and thereby results in a negative overhead. The maximum worst-case migration overhead is less than 4 % and the average worst-case migration overhead is 0.1 %, which is negligible.

7.5.2 Control Loop for Per-Cluster DVFS

The IL-based migration is integrated with a DVFS control loop that selects the per-cluster V/f levels. The control loop utilizes the estimated $\tilde{f}_{k,min}$ per application k, as defined in Eq. (7.1), which is an estimate of the minimum required V/f level to reach the QoS target of this application. It then determines the minimum required V/f level per each cluster x to satisfy the QoS target of all applications running on it:

$$\tilde{f}_x = \max\left\{\tilde{f}_{k,min} : \text{application } k \text{ mapped to cluster } x\right\} \tag{7.7}$$

Since the run-time estimates of $\tilde{f}_{k,min}$ are based on linear scaling, they are only accurate for small V/f changes. Therefore, we adjust the current V/f level f_x of each cluster by only one step towards $\tilde{f}_x$ and call this control loop more frequently than migration, i.e., every 50 ms. This control loop is skipped in two iterations, one when application migration is executed and one directly after a migration, to account for transient effects of cold caches that result in spurious QoS violations. Idle clusters are throttled to the lowest V/f level. The control loop is embedded into the same C++ application that also performs migration, and uses the Linux *userspace* governor to set per-cluster V/f levels.

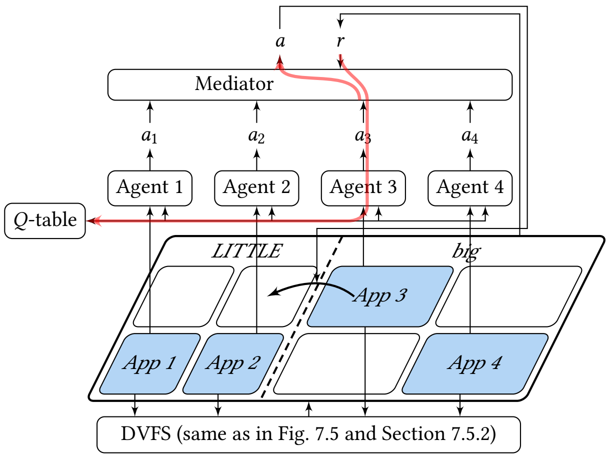


Figure 7.7: Overview of RL-based application migration. One agent is instantiated per application. All agents share the same Q-table. A mediator selects the executed action a from all per-agent actions a_i (in this example from Agent 3). Only the agent whose action has been selected updates the Q-table based on the reward.

The combination of IL-based application migration and a simple DVFS control loop enables us to achieve temperature optimization under QoS targets, as will be evaluated in Section 7.7.

7.6 Reinforcement Learning-Based Application Migration

As discussed in Section 1.4.3, RL is another method for end-to-end learning and directly making management decisions, like IL. However, IL outperforms RL in terms of quality and stability of the learned policy. To demonstrate this in a quantitative comparison, there is a need to implement an RL-based technique *Therm-RL* with the same goal as our IL-based *TOP-IL*. However, there is no state-of-the-art technique that employs RL for application mapping/migration or DVFS for temperature minimization and considers heterogeneous cores with per-cluster DVFS running parallel applications. Therefore, this section develops an RL-based application migration policy, motivated by the state of the art, to serve as a baseline for the IL-based policy described in Section 7.4. To enable a fair comparison between RL and IL, we also perform only application migration with RL and employ the same DVFS control loop described in the previous section.

TOP-IL copes with a varying number of running applications by performing independent inference per each running application, denoted the AoI, to find the optimal migration. RL additionally requires to perform training at run time, which requires maintaining information about the previous state. Therefore, one agent is instantiated per application. This has the additional benefit of maintaining state and action spaces at a reasonable size. Fig. 7.7 depicts the overall structure of *Therm-RL*.

State The state space used for the RL agents comprises the same features as also used for the IL model. In particular, these are the QoS, number of L2-D accesses, and the current mapping of the AoI, as well as the V/f levels and utilization values of the *big* and *LITTLE* clusters. All these features are quantized to maintain a reasonable size of the Q-table. For instance, the information about the QoS of the AoI is represented by a binary signal indicating whether or not the QoS target is met.

Action The action space is selected the same as with the IL technique, which is also the same as in [79]. There is one action per core, indicating a migration to this core, i.e., 8 actions in total. The Q-table contains 2,304 entries, which is similar in size to what is reported in the related work, e.g., in [77].

Reward The reward function needs to combine the objective, i.e., temperature minimization and constraint, i.e., QoS targets, into a single scalar value. The objective the same as in [79], which only rewards a low temperature T: $r = 80°\mathrm{C} - T$. We extend it to penalize QoS violations:

$$r = \begin{cases} 80°\mathrm{C} - T & \text{if } \forall i : q_i \geq Q_i \\ -200 & \text{otherwise (QoS violation)} \end{cases} \tag{7.8}$$

We have empirically tuned the negative reward of −200 for a QoS violations in order to achieve a good trade-off between low temperature and low QoS violations.

Multi-Agent Learning for Parallel Applications As discussed earlier, one RL agent is instantiated per application. Therefore, mediation between the agents is required to avoid 1) contradicting decisions by different agents, and 2) further instability in the learning. Contradicting migration decisions could happen if two agents decide to perform a migration at the same time to the same core. Such decisions should be not executed, because applications sharing a core would likely violate the QoS targets. Moreover, even two migrations to different cores should not be executed at the same time, as simultaneous migrations might nullify the benefits of each other. Additionally, a change in temperature when performing two migrations at once, which is represented in the reward signal, can not be traced back to either of the two, causing further instability in the learning.

We solve this by implementing a mediator between the agents, similar to [144]. The mediator selects the best action among the individual actions selected by each agent based on the highest Q-value. After having executed this action, the reward obtained in the next control step should only be used to perform learning about this action, not about actions from other agents that have not been selected and executed. Therefore, the mediator forwards the reward signal only to the agent selected in the previous step to perform learning. Fig. 7.7 illustrates this mediation process. All agents share a single common Q-table to improve generalization to different applications, and to immediately start with a trained policy when a new application arrives to the system.

Training We select the training parameters similar to [79]. We employ an ϵ-greedy policy with $\epsilon = 0.1$, a discount factor $\gamma = 0.8$, and a learning rate $\alpha = 0.05$. As the Q-table is initialized with constant values, the initial performance of an RL policy is not representative. A high-quality RL policy is only obtained after significant training. Therefore, we irst train a policy until convergence (~3 h) on a different random workload from what is used later in the evaluation. The resulting Q-table is stored and loaded at the beginning of each evaluation run. To reduce the impact of randomness on the evaluated policy performance, three policies are trained from scratch with different random seeds, like with the IL model.

7.7 Experimental Evaluation

This section presents an experimental evaluation of *TOP-IL*. The evaluation uses the setup described in Section 3.2, which employs a *HiSilicon Kirin 970* smartphone SoC that implements the common Arm big.LITTLE architecture, and comes with an NPU to accelerate NN inference.

TOP-IL is compared with the RL-based technique *Therm-RL* presented in Section 7.6, as well as with state-of-the-practice solutions, Linux GTS, paired with either *ondemand* or *powersave* governors. GTS assigns applications to a cluster depending on their computational requirements, i.e., mostly-idle and performance-hungry applications are migrated to the *LITTLE* and *big* cluster, respectively. *Ondemand* aims at providing a high performance but saving power when applications are mostly idle. It scales the V/f levels based on the CPU utilization, where V/f levels are upscaled if the utilization exceeds a fixed threshold, and downscaled if it falls below a second threshold. *Powersave* minimizes the power consumption by always operating the clusters at the lowest V/f levels, irrespective of the associated performance losses. These Linux policies are not aware of detailed application characteristics like memory-/compute-intensity or QoS targets. *GTS/ondemand* is the default resource management that is shipped with Android 8.0 on *HiKey 970*.

The following experiments demonstrate that *TOP-IL* and the employed NN model can cope with:

1. Unseen applications that have not been used for training.
2. Different cooling: We perform experiments also with passive cooling (without a fan) instead of the active cooling used for training data generation.
3. Randomness in the training and at run time: We train three models with different random seeds to demonstrate the robustness to weight initialization. We then repeat the experiments three times, where each repetition uses a different model, and report average and standard deviation of results. This demonstrates robustness to run-time variability due to workload fluctuations.

In addition, we demonstrate the stability of the learned policy.

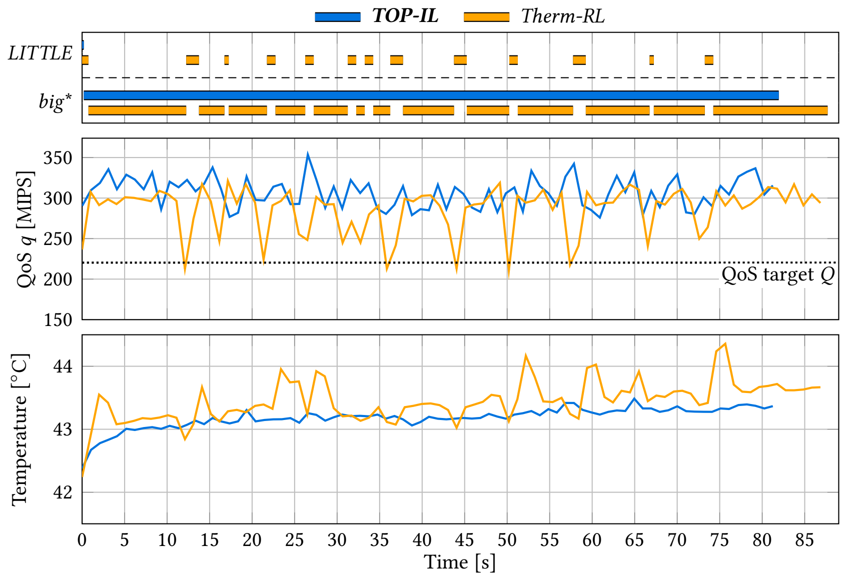
7.7.1 Illustrative Example

This section presents an illustrative example comparing the migration decisions of IL and RL. We study the same case as presented in the motivational example in Fig. 7.1, i.e., we run the two applications *adi* and *seidel-2d*. Fig. 7.8a plots the selected cluster (mapping) of *adi*. As has been shown in Fig. 7.1, a mapping to the *big* cluster is optimal for *adi*. *TOP-IL* always selects the optimal mapping. *Therm-RL* also mostly maps *adi* to the *big* cluster but infrequently migrates *adi* to the *LITTLE* cluster. With both techniques, *adi* reaches its QoS target. Their temperature is also similar, as they select the same mapping most of the time. Fig. 7.8b shows the mappings with *seidel-2d*, for which the *LITTLE* cluster is optimal. *TOP-IL* again consistently selects the optimal mapping. In contrast, *Therm-RL* is more unstable and migrates *seidel-2d* irregularly between the clusters. This results in an unnecessarily high QoS during the time on the *big* cluster, which also increases the temperature during these periods. These experiments illustrate that the policy learned with IL is stable and consistently selects the optimal mapping, in contrast to RL, which is more unstable. This ultimately results in a lower temperature with *TOP-IL*. The instability of RL leads to even worse results, such as frequent QoS violations, with more realistic workloads with multiple parallel applications, as will be shown in the next section.

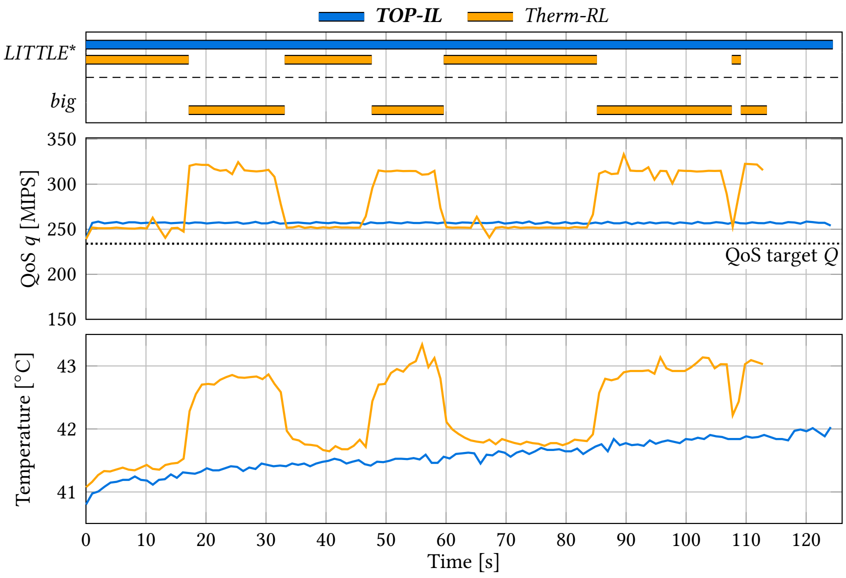
7.7.2 Parallel Mixed Workload

We now evaluate the capabilities of all techniques to minimize the temperature under QoS targets. This is the main experiment in this chapter. We create a mixed workload of 20 randomly selected applications from *blackscholes*, *bodytrack*, *canneal*, *dedup*, *facesim*, *ferret*, *fluidanimate*, and *swaptions* from *PARSEC*, and *adi*, *fdtd-2d*, *floyd-warshall*, *gramschmidt*, *heat-3d*, *jacobi-2d*, *seidel-2d*, and *syr2k* from *Polybench*. Only the *Polybench* applications (except *jacobi-2d*) have been used for training in *TOP-IL* and *Therm-RL*. All other applications are unseen. We create a random QoS target for each application. The arrival times are distributed by a *Poisson* distribution with varying arrival rate to test different system load values. With *TOP-IL*, the average/peak system utilization varies from 13 %/38 % to 37 %/75 %, for the minimum and maximum arrival rate, respectively. We let the board cool down for 10 min between subsequent experiments. All experiments are performed three times with different models for *TOP-IL* and *Therm-RL*, as explained earlier.

Figs. 7.9a and 7.9b show the results (mean and standard deviation for three repetitions) for the cooling with a fan, i.e., same as during training data generation, *and without a fan*, i.e., different from the training data, respectively. *TOP-IL* reduces the average temperature during executing the workload by up to 17 °C compared to *GTS/ondemand* at only slightly more QoS violations. *GTS/powersave* achieves the lowest average temperature but the majority of applications violate their QoS target. Finally, the temperatures with *Therm-RL* and *TOP-IL* are similar. However, *TOP-IL* achieves 63 % to 89 % fewer QoS violations than *Therm-RL*. In summary, *TOP-IL* is the only technique to achieve temperature minimization at few QoS violations. This result is independent of the cooling setting.

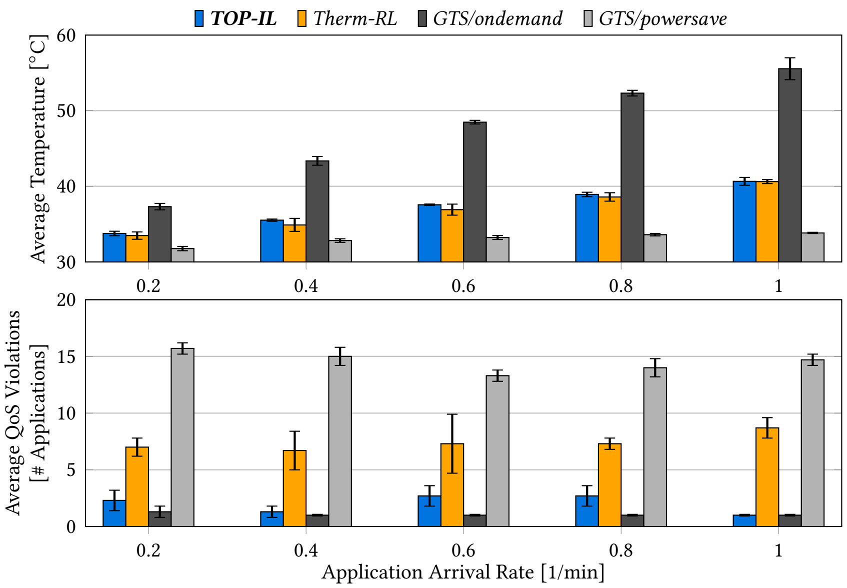


(a) *adi* (*optimal mapping: *big*)

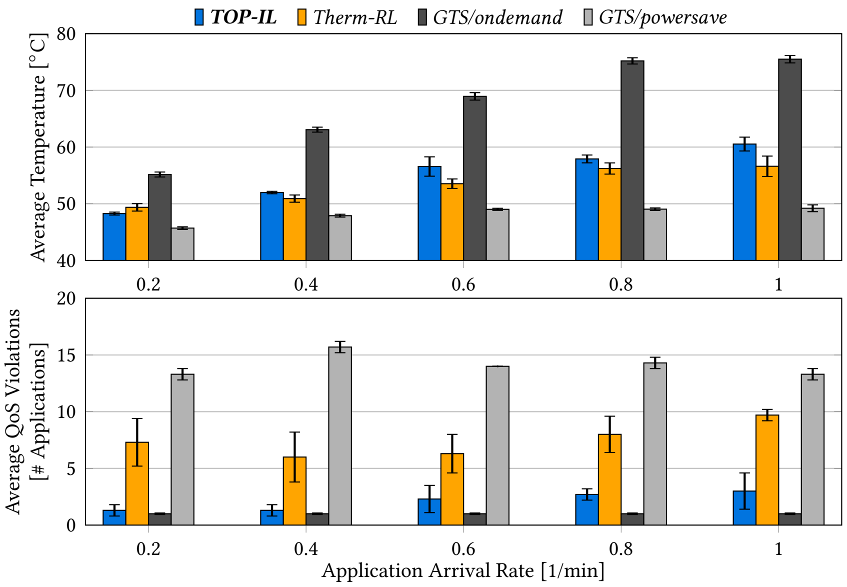


(b) *seidel-2d* (*optimal mapping: *LITTLE*)

Figure 7.8: Illustrative example demonstrating the mappings chosen by *TOP-IL* and *Therm-RL* with the two applications *adi* and *seidel* studied already in the motivational example (Fig. 7.1). *TOP-IL* consistently selects the optimal mapping for both applications. *Therm-RL* in general shows a similar trend but is unstable, selecting also suboptimal mappings. The QoS targets are reached in all cases. However, the unstable mappings of *Therm-RL* cause a higher temperature with *seidel-2d*.

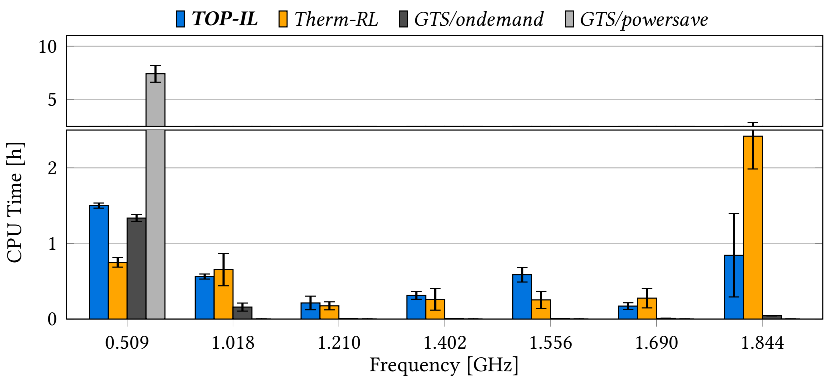


(a) With a fan (same as for oracle demonstrations)

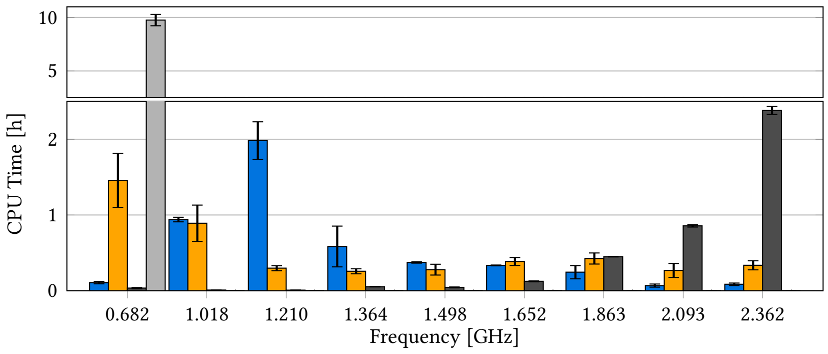


(b) Without a fan

Figure 7.9: *TOP-IL* significantly reduces the temperature, while achieving few QoS violations. This is the case both when running with a fan, i.e., same as when recording the traces for the oracle demonstrations, but also without the fan, demonstrating the generalization of the NN model. The bars show mean and standard deviation over three experiments. *TOP-IL* and *Therm-RL* use models trained with different random seeds.



(a) *LITTLE* cluster



(b) *big* cluster

Figure 7.10: Distribution of the total CPU time (among all arrival rates and repetitions) per technique to the clusters and V/f levels in the experiments with a fan (Fig. 7.9a).

To explain these results we analyze the selected mappings and V/f levels by the techniques. Fig. 7.10 plots the distribution (mean and standard deviation for the three repetitions) of the total CPU time for executing the workload at all arrival rates according to the cluster and selected V/f level for the experiment without a fan. The CPU time is counted only during executing an application. GTS favors the *big* cluster and *ondemand* selects high V/f levels when applications are executed. As a result, *GTS/ondemand* uses most CPU time at the highest V/f level on the *big* cluster, explaining it low QoS violations. However, this also leads to high temperature and ultimately even causes thermal throttling by DTM, forcing *GTS/ondemand* to occasionally reduce the V/f levels. In contrast, *powersave* always selects the lowest V/f level. This reduces the performance, and, hence, increases the number of simultaneously running applications. This forces GTS to also use the *LITTLE* cluster. As a result, *GTS/powersave* uses most CPU time on both clusters at the lowest V/f level, leading to the lowest temperature but many QoS violations. *Therm-RL* uses most CPU

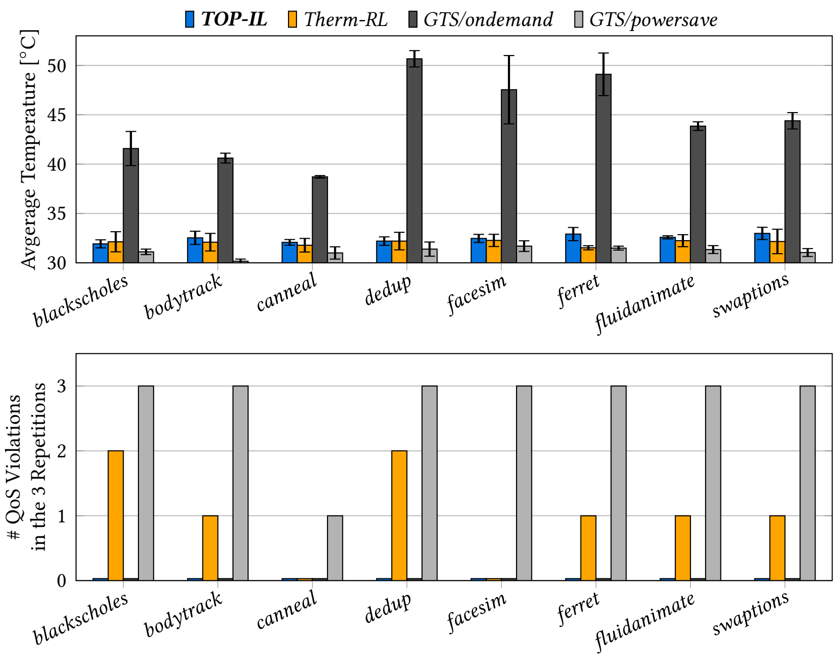


Figure 7.11: *TOP-IL* is the only technique to achieve no performance violations, yet low temperature for all single application workloads. All applications are unseen, i.e., not used for training.

time on the *LITTLE* cluster at the highest V/f level and on the *big* cluster at the lowest V/f level. When executing on the *big* cluster at the lowest V/f level, likely a migration to the *LITTLE* cluster would be beneficial to reduce the temperature. More severely, when the DVFS control loop selects the highest V/f levels on the *LITTLE* cluster, it is likely the case that the QoS target of an application is missed, explaining the high number of QoS violations. In both cases, a migration to the other cluster would likely have been beneficial. The reason for the suboptimal mapping decisions of *Therm-RL* are policy instability due to continual exploration in online learning and combining objectives and constraints into a single scalar reward, as discussed earlier. In contrast, *TOP-IL* uses more time on the *big* cluster at rather low V/f levels, which allows it to meet the QoS target at a low temperature, as reported in Fig. 7.9. We also did this analysis for the experiment with a fan and found similar results. The only difference is that a fan prevents thermal throttling even with *GTS/ondemand*. *In summary, TOP-IL is the only technique to achieve temperature minimization and low QoS violations. This is achieved for mixed workloads with unseen applications, for different cooling setting than used during training, and is reproducible for models trained with different random initialization.*

7.7.3 Single-Application Workloads

The mixed workloads used in the previous section contain both seen and unseen applications. To further demonstrate the generalization of *TOP-IL*, we perform experiments with *only unseen* applications from *PARSEC*. The QoS targets of the applications are set such that they can be met at the highest V/f level on the *LITTLE* cluster. As in the previous section, we repeat each experiment three times with different models. Fig. 7.11 visualizes the average temperature and QoS violations for different applications. As in the previous experiments, *GTS/ondemand* reaches the highest average temperature. The other three techniques all achieve in a similar low temperature. There is only one application per workload, which can either reach or violate its QoS target. Therefore, Fig. 7.11 reports the number of executions with a QoS violation instead of the average number of applications that violate their QoS. As expected, *GTS/powersave* violates almost all QoS targets, as it selects the lowest V/f levels. The only exception is *canneal*, which is memory-intensive and its performance depends less on the V/f level. *Therm-RL* also violates the QoS target in 33 % of the executions. The reason is that the policy learned with RL suffers from instabilities, which causes frequent migrations, as observed in Fig. 7.8. After each migration, the DVFS control loop requires a few iterations to determine the required V/f level, as idle clusters are throttled to the minimum V/f level. During this period, the QoS may be temporarily violated, potentially resulting in a global QoS violation among the whole execution. *TOP-IL* is the only technique that achieves both a low temperature and no QoS violations. These experiments demonstrate again the capabilities of *TOP-IL* to effectively minimize the temperature under a QoS target, but most importantly *demonstrate the generalization capabilities of TOP-IL to unseen applications.*

7.7.4 Model Evaluation

This section evaluates the NN model of *TOP-IL*. We split the training/test data into training and test based on the AoI, where seven out of nine (78 %) benchmarks are only used for training (same as in the previous sections), and others (22 %) only for testing. As discussed earlier, our goal is to select any near-optimal mapping in case several mappings result in a similar low temperature. The following numbers report the mean and standard deviation across three models trained with different random seeds. Our model selects a mapping within 1 °C of the optimum in 82±5 % of the test examples. The selected mapping is, on average, only 0.5±0.2 °C hotter than the optimal mapping. *This demonstrates that the training process of TOP-IL is robust and consistently creates models that make near-optimal decisions.*

7.7.5 Run-Time Overhead

The results in Figs. 7.8 to 7.11 already inherently contain the run-time overhead in terms of additional CPU load and induced temperature of *TOP-IL*, as it is running in parallel to the workload. We perform in this section additional experiments to explicitly report the overhead of our technique. We study different system utilization values, i.e., different numbers

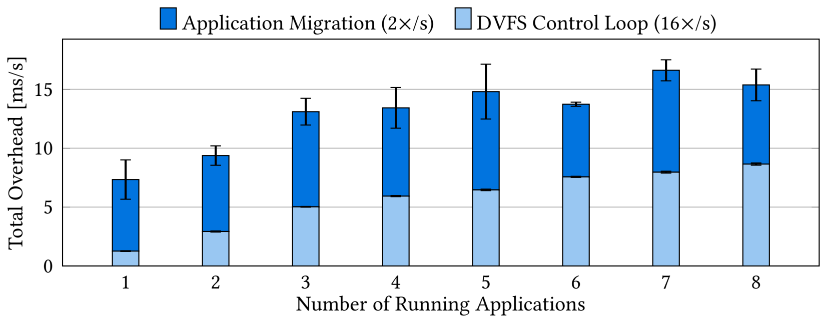


Figure 7.12: The run-time overhead of the DVFS control loop increases with the number of executed applications, whereas batch inference allows to maintain a constantly low overhead for the application migration policy.

of running applications, ranging from 1 to 8 parallel applications. Fig. 7.12 presents the results. The DVFS control loop is invoked 16 times per second. Its overhead increases with the number of applications. The main component is reading the performance counters, which scales linearly with the number of applications. In contrast, the overhead of the migration policy, which is executed twice per second, barely changes with more parallel applications. This is as its main component is the NN inference, which uses parallel inference of the NN using the NPU, and thereby maintains a constant low latency. In the worst case, the DVFS control loop and migration policy have an overhead of 8.7 ms/s and 8.6 ms/s (0.54 ms and 4.3 ms per invocation), respectively. The total run-time overhead of *TOP-IL* is less than 1.7 %, and, therefore, negligible. It is important to notice that *TOP-IL* uses a single-threaded implementation, i.e., the overhead only affects a single core.

7.8 Summary

This chapter presented *TOP-IL*, which performs temperature minimization under QoS targets on heterogeneous clustered multi-core processors using application migration and DVFS. We showed that optimization can only be achieved by jointly considering the diverse characteristics and QoS targets of all running applications, and, hence, is a complex problem. We tackle this complexity with NN-based IL, which enables us to combine the optimality of the oracle policy with a low run-time overhead. In addition, this is the first work to employ an existing NPU of a smartphone SoC to accelerate the run-time inference of NN-based resource management. We also implemented RL-based management, which is a popular method to achieve end-to-end learning of resource management actions. The evaluation showed that RL suffers from instability in the learning due to continuous exploration and online learning at run time, and due to requiring to combine objective and constraints into a single scalar reward. In contrast, IL-based management offers stable near-optimal management. *TOP-IL* generalizes to different workloads and cooling settings than what have been used for training.

8 Resource-Adaptive On-Device Training

The techniques presented in the previous chapters all perform training at design time. While this is for instance beneficial to maintain a low run-time overhead, gathering representative training data for all possible scenarios that may happen at run time is challenging. This could be solved by performing continuous training at run time based on training data that is collected at run time. However, as discussed earlier, run-time training may greatly increase the run-time overhead. This chapter targets run-time training of an NN with limited available computational resources.

Training an NN model may require more resources in terms of computation and data than are available on a single device. Therefore, distributed training is commonly performed, which leverages the computational resources and data of many devices, such as smartphones or IoT nodes. A prime use case is FL, which has already been introduced in Section 3.3. FL has proven effective in large-scale real-world systems [37, 145, 146]. However, training of a deep NN model is resource-hungry in terms of computation, energy, time, etc. [147], and it is rather unrealistic to assume that all devices that participate in an FL system can perform all types of training computations all the time, especially if the training is distributed on edge devices. The reason is that the computational capabilities of devices participating in an FL system may be *heterogeneous*, e.g., different hardware, different generations, availability of an NN accelerator [145]. *More importantly, the resources available on a device for training may change over time.* This may for instance be due to shared resource contention [148], where CPU time, cache memory, energy, etc. are shared between the learning task and parallel tasks. This is illustrated with the following two examples.

8.1 Motivational Examples

Edge Computing for Real-Time Video Analytics Edge computing is already employed in ML-based real-time video analytics, where each edge device processes images from several camera modules [149]. Currently, these edge devices mostly perform inference, but there is a clear trend towards additionally performing distributed learning via FL [150]. This results in the learning task and the inference task sharing computational resources. The workload induced by the inference task depends on the activity in the video streams and,

This chapter is mainly based on [3].

therefore, changes over time, as processing may be skipped for subsequent similar images to save resources [149]. These changes happen fast, i.e., within seconds [151], while FL round times may be minutes [145]. In addition, the moments when changes happen are unpredictable and uncorrelated to the FL round times.

Mobile Computing for Next-Word Prediction Google *GBoard* [152] trains a next-word-prediction model using FL on end users' mobile phones. This model is used by the keyboard application to provide suggestions to the user for faster typing. To avoid slowing down other user applications, and thereby degrading the user experience, training is currently performed only when the device is charging and idle, and aborted when these conditions change. This introduces a bias towards certain devices and users, degrading the achievable model accuracy [152]. This could be resolved by allowing training also when the device is in use, but only using free resources. Smartphone workloads change within seconds [153], which is faster than the *GBoard* round time of several minutes, changing the free resources for training fast. In addition, workload changes originate in user activity, which is independent of the FL round times.

In both examples, the learning task is subject to changing resource availability. In particular, resources may change *faster than the FL round times* and *at any time, unpredictable to the FL system.*

8.2 Challenges and Novel Contributions

While several works study the problem of static resource heterogeneity across devices [154, 155], time-varying resource availability has so far been neglected. This chapter introduces a distributed, resource-aware, adaptive, on-device learning technique, *DISTREAL*, which enables local training on each device to fully exploit and efficiently utilize the available resources, dealing with all these types of heterogeneity. Our objective is to maximize the accuracy that is reached after limited training time, i.e., the convergence speed. To fulfill this goal, we need to C1) *fully* exploit the available, limited, resources on a device. This requires fine-grained adjustability of the training on a device, and a method to instantly react to changes; and C2) *efficiently* use the available resources on a device, to maximize the accuracy improvement and, hence, the overall convergence speed. Specifically, the work presented in this chapter provides the following novel contributions:

- We introduce and formulate the problem of heterogeneous time-varying computational resource availability in FL.
- We propose a dropout technique to dynamically adjust the resource requirements of training the model. Thereby, each device locally decides the dropout setting which fits its available resources, without requiring any assistance from the server, addressing C1.
- We show that using different per-layer dropout rates achieves a better trade-off between the resource requirements and the convergence speed, compared to using the same rate at all layers as the state of the art, addressing C2. We present a DSE

technique to automatically find the Pareto-optimal per-layer dropout rates for a given NN at design time.

We implement *DISTREAL* in an FL system, in which the availability of computational resources varies both between devices and over time. We show through extensive evaluation that *DISTREAL* significantly increases the convergence speed over the state-of-the-art techniques, and is robust even to rapid changes in resource availability at the devices, without compromising on the final accuracy.

8.3 Problem Definition

We target synchronous FL, as introduced in Section 3.3. Our objective is to maximize the convergence speed, i.e., the reached accuracy after a certain number of rounds, under heterogeneous resource availability between devices and over time.

Our technique to perform resource-aware training of NNs comprises two parts. At run time, *DISTREAL* dynamically drops parts of the NN using an adapted version of dropout [51]. By changing the dropout rates per layer, a fine-grained trade-off between resource requirements and convergence speed can be achieved. The Pareto-optimal vectors of dropout rates are obtained at design time using a DSE.

8.4 Dropout to Reduce Computations In Training

Dropout has been originally designed as a regularization method to mitigate overfitting [51]. It randomly drops individual neurons during training with a certain probability called the *dropout rate*. This results in an irregular fine-grained pattern of dropped neurons. The major deep learning libraries perform dropout by calculating the output of all neurons and multiplying the dropped ones with 0 [135, 156]. This wastes computational resources. It would be more efficient to instead not calculate values that are going to be dropped. Convolutional and fully-connected layers are implemented as matrix-vector or matrix-matrix operations that are heavily optimized with the help of vectorization [135, 156]. Skipping the calculation of individual values would result in sparse matrix operations, which breaks vectorization, increasing the required resources instead of decreasing them [157].

8.4.1 Filter-based Structured Dropout

To reduce the number of computations, the dropout pattern needs to show some regularity that still allows performing vectorization of dense matrix operations. This is achieved by dropping contiguous parts of the computation [111]. Modern NNs consist of many different layer types such as convolutional, pooling, fully-connected, activation, and normalization layers. Many of these layers are computationally lightweight (e.g., pooling, activation), while some contain the majority of computations (convolutional and fully-connected layers). Fig. 8.1 shows how the number of MACs for the forward pass is distributed between fully-connected and convolutional layer types for different state-of-the-art convolutional

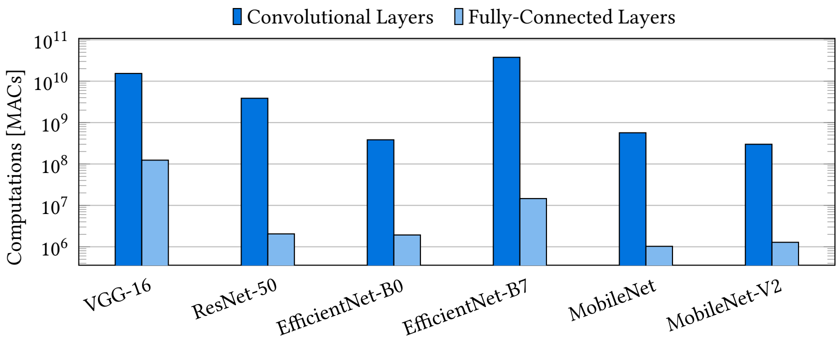


Figure 8.1: Convolutional layers account for the majority of MAC operations in CNNs.

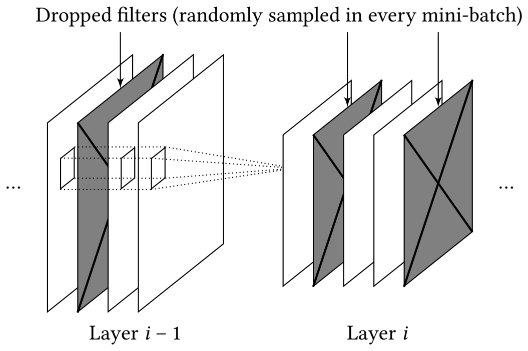
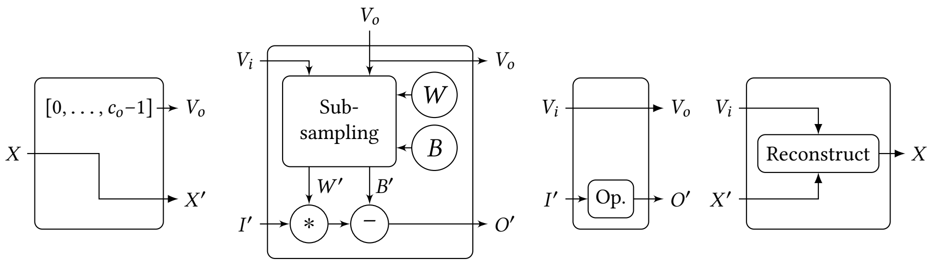


Figure 8.2: Filter-based structured dropout in a convolutional layer significantly reduces the required computations while still maintaining regularity in the calculations.

neural networks (CNNs). We observe that for all architectures, the convolutional part requires orders of magnitude more MACs in total than the fully-connected part. Therefore, a technique to save computations needs to target the convolutional layers. Fig. 8.2 depicts filter-based structured dropout in a convolutional layer, as we apply in this chapter: instead of dropping individual pixels in the output, whole filters are dropped stochastically. This technique reduces the number of computations while allowing to keep existing vectorization methods.

However, dropout also changes the convergence speed. A higher dropout rate means that in each training update, a smaller fraction of each layer's weights is updated, thereby slowing down the training. Consequently, the dropout rate determines a trade-off between the resource requirements and the convergence speed. The dropout rate should always be selected as low as is affordable with the available resources. To adjust to changing resource availability, we propose to *dynamically change the dropout rate at run time.* The inference uses always the whole model.



(a) Begin sparse part **(b)** Sparse convolution **(c)** ReLU, pooling, etc. **(d)** End sparse part

Figure 8.3: The building blocks for sparse NN that implement filter-based structured dropout.

8.4.2 Efficient Implementation of Structured Dropout

This section summarizes our implementation of structured dropout in PyTorch [156], which is publicly available[1].

Convolutional layers A convolutional layer performs the following computation:

$$O = I * W + B \tag{8.1}$$

where $O \in \mathbb{R}^{b \times c_o \times w_o \times h_o}$ is the output activation (b samples per mini-batch, c_o output feature maps with width w_o and height h_o), $I \in \mathbb{R}^{b \times c_i \times w_i \times h_i}$ is the input activation (c_i input feature maps with width w_i and height h_i), $W \in \mathbb{R}^{c_o \times c_i \times k_w \times k_h}$ (width k_w and height k_h of the convolutional kernel) are the weights and $B \in \mathbb{R}^{c_o}$ are the bias.

Structured dropout drops some of the filters from the computation, i.e., some of the output feature maps in O contain only 0. Dropout in a preceding layer results in some of the input feature maps in I to contain only 0. To benefit from dropout in terms of computational requirements, unnecessary computations need to be avoided. This is achieved by excluding dropped parts from the computations by performing structured sparse convolution.

Let V_i denote the set of the indices of valid (not dropped) feature maps in the input I of the convolutional layer. V_o denotes the set of the indices of valid feature maps in the output O and is obtained from a Bernoulli process with the dropout rate d. $O' \in \mathbb{R}^{b \times |V_o| \times w_o \times h_o}$ represents the valid output feature maps in O and $I' \in \mathbb{R}^{b \times |V_i| \times w_i \times h_i}$ represents the valid output feature maps in I. It follows that

$$O' = I' * W[V_o, V_i, :, :] + B[V_o] \tag{8.2}$$

where $W[V_o, V_i, :, :]$ represents the parts of W that are obtained by keeping only valid columns and rows according to V_o and V_i (same for $B[V_o]$). *Thereby, a structured sparse convolution is replaced by a dense convolution on the subsampled inputs, weight, and bias.*

[1] https://git.scc.kit.edu/CES/DISTREAL

Subsampling of W and B needs to be done at run time in every mini-batch because V_i and V_o may change in every mini-batch. However, if the subsequent layer is aware of V_o, the full output O does not need to be reconstructed from O', and O' can simply be forwarded to the next layer, along with V_o containing the indices of valid filters. By operating on sparse representations of O and I in the form of tuples (O', V_o) and (I', V_i), our implementation avoids copying input and output activations. Fig. 8.3b depicts a structured sparse convolutional layer for structured dropout with rate d.

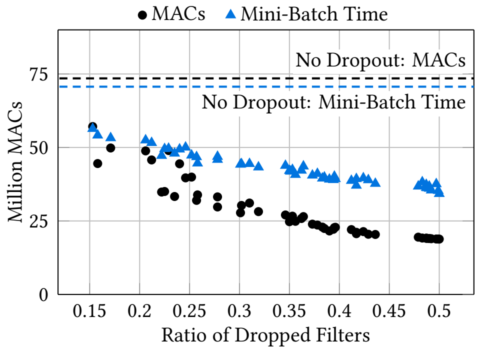
PyTorch stores weights and activations in tensors, i.e., multi-dimensional matrices. Tensors in PyTorch are stored in memory with fixed strides. Subsampling the weights W and B according to V_i and V_o would results in an irregular stride pattern, and, therefore, our implementation requires copying the underlying data in memory. These copies could be avoided by extending the backend to be aware of V_i and V_o, and still providing the full W and B to the backend. However, PyTorch supports several backends, some of which (e.g., CUDA) comprise proprietary code, and, hence, cannot be modified easily. It is also important to notice that the weight and bias tensors in convolutional layers are much smaller than the input and output activation, reducing the copy overheads.

The backward pass needs to propagate the gradient to the full weight tensors W and B and cannot operate directly on the subsampled copies because the optimizer may have an internal state (e.g., momentum). This is supported by PyTorch's *autograd* feature, which is fully-compatible with the employed subsampling of weights and bias.

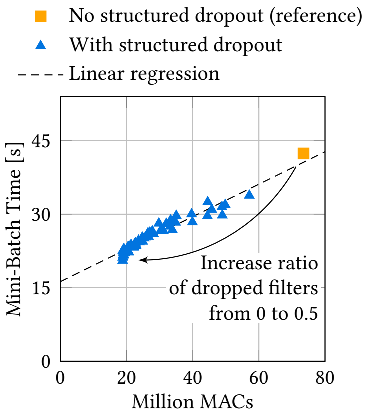
Dropout Traditional dropout performs two operations during training: it replaces dropped values by 0 and scales up non-dropped values by a factor $\frac{1}{1-d}$. Our implementation of structured dropout separates these two operations. The first operation is replaced by passing the indices of feature maps to compute V_o to the preceding convolutional layer (potentially with other layers like batch normalization in between), which consequently skips computations of filters that would be dropped. Scaling needs to be done by the dropout layer, and is implemented as with traditional dropout.

Other layers Many layers, such as activation, pooling, etc., process the individual feature maps independently. As a result, these operations can directly be applied to the subsampled valid feature maps I'. The indices of the valid feature maps of the output V_o are the same as of the input V_i, as depicted in Fig. 8.3c. We implement two additional layers that begin and end the sparse part of the NN, respectively. The beginning layer (Fig. 8.3a) annotates the activation with the information that all feature maps are valid ($V_o = [1, \dots, c_o - 1]$), which does not require any copy. The end layer (Fig. 8.3d) fills in zeros at the positions of invalid feature maps, which requires copying tensors from the valid feature maps. However, this layer is only required once near the end of the NN. We did not implement sparse fully-connected layers but similar concepts as for the convolutional layers could be applied.

Performance Fig. 8.4a depicts how the number of MACs of the forward pass evolves when applying different vectors of per-layer dropout rates for *DenseNet-40* [158]. These



(a) Number of MACs and training time over the ratio of dropped filters.



(b) Training time over number of MACs.

Figure 8.4: (a) The number of MACs and mini-batch time of training *DenseNet-40* on a Raspberry Pi 4 decrease almost quadratically with the ratio of dropped filters. (b) The mini-batch training time of *DenseNet-40* for different dropout vectors on a Raspberry Pi 4 shows a linear relationship with the number of MACs in the forward pass.

vectors are determined by applying the DSE technique introduced in this chapter. The x-axis shows the resulting ratio of dropped filters. Multiple vectors may results in the same ratio of dropped filters, while providing different convergence / resource requirement trade-offs, explaining why for some ratios there are multiple MACs. We observe that the number of MACs decreases almost quadratically with the ratio of dropped filters. We also report the training time of a single mini-batch (size 128) on a Raspberry Pi 4, which serves as an example for an IoT device, using our implementation of structured dropout in 32-bit PyTorch 1.7.1. A training step including forward pass, backpropagation, and weight updates requires about 2× more MACs than the forward pass alone [159]. The training time shows a similar trend as the number of MACs but with an offset. This is because we do not modify the backend of PyTorch to be aware of this dropout, which results in copy operations of weight tensors to repack them, as discussed earlier. As a consequence, the achieved benefits are smaller than what is theoretically achievable. In summary, this experiment demonstrates that structured dropout significantly reduces the required computational resources.

Suitability of MACs/s as Resource Metric The work presented in this chapter is based on the framework presented in Section 3.3, which expresses the resource availability as MACs/s. This paragraph describes why this abstract metric 1) is still accurate, i.e., is strongly correlates with the underlying limited resource, and 2) is a practical abstraction, i.e., it is possible to determine the available MACs/s at run time. We follow the use cases presented in Section 8.1, where the CPU time is the limited resource.

Fig. 8.4b plots the mini-batch training time (forward and backward pass) over the number of MACs in the forward pass for the same setup used in Fig. 8.4a. The mini-batch training time shows a clear linear relationship with the number of MACs. Consequently, the number of MACs in the forward pass is a good representation of the actual resource requirements during training. Hence, it can be used in the DSE to evaluate a dropout vector and at run time to represent the available resources. Note that in a general case, estimating the resource requirements from the total number of MAC is inaccurate [160] because different layer types may result in a different relationship between resource requirements and the theoretical number of MACs. This is for example the case if different layers cause different data movement between the DRAM and the CPU, or if different layers benefit differently from vectorization. However, we use the number of MACs only to compare different configurations (dropout vectors) of the *same topology*, i.e., we do not compare different topologies. Therefore, the number of MACs accurately represents the resource requirements.

Due to its strong correlation with the training time, the number of available MACs for training can also be derived at run time. In the example of CPU time as the main limited resource, the FL training is one of several tasks that compete for CPU time. The OS scheduling decides the time during which each application is executed based on many factors, such as fairness or priorities. The amount of CPU time available for training is, therefore, known at the OS level and can be made available to the applications. The time per mini-batch can be calculated from the available CPU time and the required throughput to satisfy the FL round time. Finally, the number of MACs can be derived from the mini-batch time as in Fig. 8.4b. This shows the practicality of the chosen abstraction.

8.5 Design-Time: Find Pareto-Optimal Dropout Vectors

The resource requirements for training (MACs) and convergence speed both depend on the dropout rates of *each layer*. Prior works restrict themselves to applying the same dropout rate to all layers [112]. Relaxing this restriction opens up a larger design space, where each dropout rate of each layer may be adjusted towards a better trade-off between resource requirements and training convergence. However, this design space is too large to be explored manually. For instance, *DenseNet-100* has 99 convolutional layers that each need to be assigned a dropout rate. Some works apply simple parametric functions of the depth of a layer to similar problems [161]. However, this only works in case of a homogeneous NN structure, where properties of layers (e.g., MACs) change monotonically with the depth. This is not generally the case. For instance, layers in *DenseNet* alternate between computationally lightweight and complex, rendering a simple parametric function suboptimal. This section describes the required automated DSE technique to efficiently explore such a large design space. The DSE is performed only once at design time.

Specifically, the design space contains all combinations of dropout values per each layer. We select dropout values from the continuous range $[0, 0.5]$ because higher values reduce the final achievable accuracy, as we observe in our experiments, as well as indicated in

Table 8.1: Expected number of MACs of the forward pass of individual layers with structured dropout.

Layer Type	MACs		
Convolution	$(1-d)\cdot\lvert Y\rvert\cdot\big((1-d_p)\cdot c_i\cdot k_w\cdot k_h + b\big)$		
	where	d	Dropout rate in this layer
		$\lvert Y\rvert$	Number of output pixels
		d_p	Dropout rate in the preceeding conv. layer
		c_i	Input channels
		k_w,k_h	Kernel width and height
		b	1 if bias is used, 0 otherwise
Batch Normalization, Activation, Pooling	$(1-d_p)\cdot x$		
	where	d_p	Dropout rate in the preceeding conv. layer
		x	Number of MACs without dropout

previous works [51]. The design space for an NN with k convolutional layers is $[0, 0.5]^k$. We have two contradicting objectives: low resource requirements and high convergence speed.

8.5.1 Calculate the Resource Requirements

As discussed in Section 3.3, the number of MACs is an implementation-independent representation of the resource requirements. The previous section additionally showed that it is a suitable abstraction. Dropout is a probabilistic process, i.e., the number of MACs varies between different update steps because a different number of filters may be dropped in every mini-batch. Table 8.1 shows for the most important layers in a CNN how the expected number of MACs in the forward pass depends on the dropout rate d. For a convolutional layer, the number of MACs depends not only on the dropout rate of this layer but also on the dropout rate of the preceding convolutional layer because a reduced number of filters (dropout of this layer) is applied to a reduced number of input feature maps (dropout of previous layer). In the specific case that both dropout rates are the same, the expected number of MACs reduces almost quadratically with the dropout rate. Most other layers (batch normalization, activation, pooling) process each feature map independently. Their expected number of MACs reduces linearly with the dropout rate. As convolutional layers account for the majority of MACs, the overall reduction of the expected number of MACs is almost quadratic.

8.5.2 Measure the Convergence Speed

The convergence speed with a certain dropout vector is measured by observing the accuracy change when training with this vector. Exploring the design space would take too long if a full training with every candidate dropout vector would be performed. Instead, we assess the accuracy change after a short training, similar to learning curve extrapolation methods used in NAS [162]. We train for 64 mini-batches with batch size 64, which allows us to explore many candidate dropout vectors in reasonable time. This corresponds to the

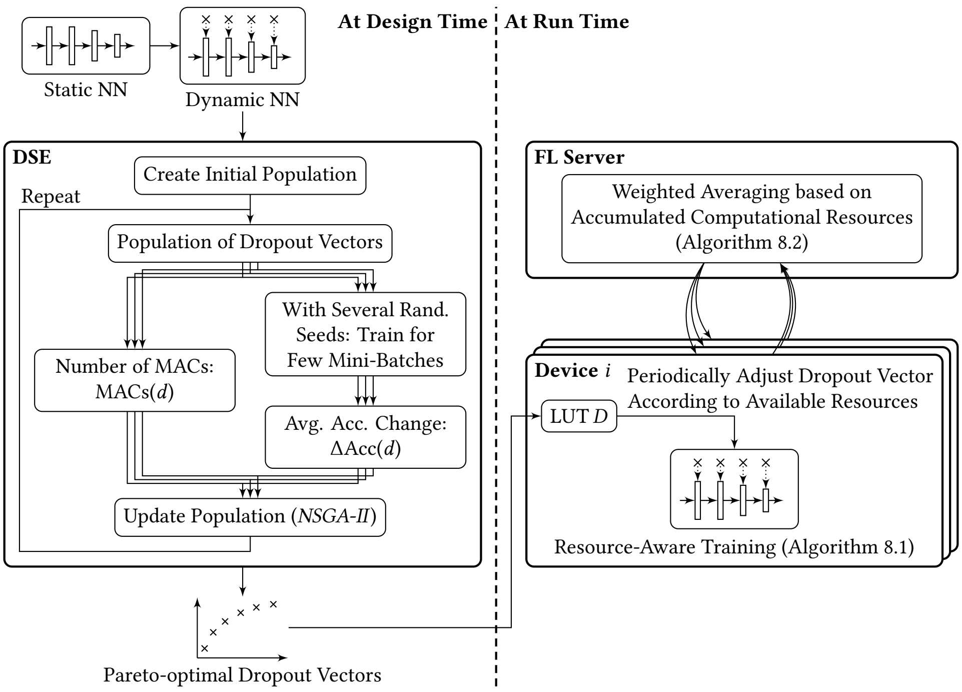
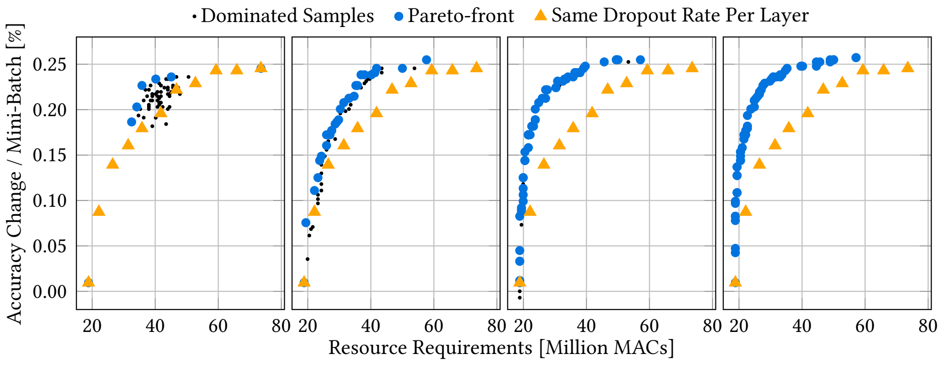

Static NN
Dynamic NN
At Design Time
At Run Time
DSE
Create Initial Population
Repeat
Population of Dropout Vectors
With Several Rand. Seeds: Train for Few Mini-Batches
Number of MACs: MACs(d)
Avg. Acc. Change: ΔAcc(d)
Update Population (*NSGA-II*)
Pareto-optimal Dropout Vectors
FL Server
Weighted Averaging based on Accumulated Computational Resources (Algorithm 8.2)
Device i
Periodically Adjust Dropout Vector According to Available Resources
LUT D
Resource-Aware Training (Algorithm 8.1)

Figure 8.5: Efficient resource-aware training with *DISTREAL* comprises the DSE to find Pareto-optimal vectors of dropout rates per layer w.r.t. the computational resources (number of MACs) and convergence speed (accuracy change after few batches of retraining), resource-aware training on each device at run time, and resource-aware weight aggregation on the server at run time.

amount of data collected by few devices. To reduce the impact of the random initialization, the NN is not trained from scratch but from a snapshot after partially training it on a distorted version of the dataset. For instance, we reduce the brightness, contrast, and saturation to 0.5 of the original value for *CIFAR-10/100*. The DSE, therefore, does not require access to the devices' data, but only access to a small amount of similar data. We repeat this with three different random seeds to further reduce the impact of random variations. The convergence speed is represented by the average accuracy improvement after short training.

8.5.3 Evolutionary Design Space Exploration

Fig. 8.5 shows the DSE flow. The problem of finding the Pareto-optimal dropout vectors is a multi-objective optimization. This is a well-studied class of problems with a plethora of established algorithms. Evolutionary algorithms have successfully been employed for NAS [163], which is related to the problem studied in this chapter. Note that we are not searching for a new NN topology, but tune parameters of a given topology. The output of the DSE is the Pareto-front of per-layer dropout vectors. To have a large variety of



(a) Generation 0 (random) **(b)** Generation 5 **(c)** Generation 20 **(d)** Generation 50 (final)

Figure 8.6: The Pareto-front for DenseNet-40 significantly outperforms setting the same rate for all layers.

options to chose from at run time, but also keep a low number of vectors to be stored, the Pareto-front should be approximately equidistantly represented. We use the *NSGA-II* [164] genetic algorithm from the *pygmo2* library [165]. *NSGA-II* explores the design space by crossover, which creates new dropout vectors by combining two dropout vectors of the current population, and mutation, which applies random changes to dropout vectors, and is designed to obtain an equidistant representation of the Pareto-front. Thereby, an *individual* is one dropout vector of per-layer dropout rates. For our largest studied NN, *DenseNet-100*, each individual contains 99 float values between 0 and 0.5. A *population* is a set of individuals. We use a population size of 64. A *generation* performs one optimization step on the population with the goal to find a new population closer to the true Pareto-front. The optimization minimizes a two-dimensional *fitness* function $f(d)$ for a dropout vector d that normalizes the values of the resource requirements $\mathrm{MACs}(d)$ and convergence speed $\Delta\mathrm{Acc}(d)$ to the range $[0, 1]$:

$$f(d) = \begin{pmatrix} \frac{\mathrm{MACs}(d) - \mathrm{MACs}(\{0.5,\ldots,0.5\})}{\mathrm{MACs}(\{0,\ldots,0\}) - \mathrm{MACs}(\{0.5,\ldots,0.5\})} \\ \\ \frac{\Delta\mathrm{Acc}(\{0,\ldots,0\}) - \Delta\mathrm{Acc}(d)}{\Delta\mathrm{Acc}(\{0,\ldots,0\}) - \Delta\mathrm{Acc}(\{0.5,\ldots,0.5\})} \end{pmatrix} \tag{8.3}$$

Fig. 8.6 plots the evolving population of dropout vectors for *DenseNet-40*. The initial population comprises 62 random dropout vectors and two samples, where all dropout values are 0 or 0.5, respectively, to accelerate the exploration of the Pareto-front by leveraging the crossover operation in *NSGA-II*. After 50 generations of *NSGA-II*, the Pareto-front has fully evolved and finds a continuous trade-off between resource requirements and convergence speed. Importantly, the Pareto-front found by the DSE provides a significantly better trade-off between the resource requirements and the convergence speed than using the same dropout rate for all layers.

Algorithm 8.1 Each Selected Device i (Client) in *DISREAL*

```
Require: $D$: LUT of Pareto-optimal dropout vectors (from DSE)
  receive $\theta_{init}$ from server
  $\theta \leftarrow \theta_{init}, c \leftarrow 0$
  for each $b \in X_i$ do                         ▷ iterate over mini-batches from local data
      $r \leftarrow r_i(t)$                                    ▷ current resource availability
      $d \leftarrow D[r]$                                    ▷ resource-aware dropout vector
      Update dropout values of local NN with $d$
      $\theta \leftarrow \theta - \eta \frac{\partial}{\partial\theta} L(b; \theta)$                                  ▷ update step
      $c \leftarrow c + \text{MACs}(d)$                          ▷ accumulate computations
  send $(\theta - \theta_{init}, c)$ to server              ▷ weight update and computations
```

Algorithm 8.2 Server in *DISREAL*

```
  $\theta_0 \leftarrow$ random initialization
  for each round $t = 1, 2, \dots$ do
      $K \leftarrow$select $n$ devices
      broadcast $\theta_{t-1}$ to selected devices $K$
      receive $(d\theta_i, c_i)$ from devices $i \in K$
      $C \leftarrow \sum_{i \in K} c_i$                          ▷ total number of computations
      $d\Theta \leftarrow \sum_{i \in K} c_i \cdot d\theta_i$                                ▷ weighted sum
      $d\theta \leftarrow d\Theta / C$                                  ▷ weighted average
      $\theta_t \leftarrow \theta_{t-1} + d\theta$
```

8.6 Run Time: Resource-Aware Training of Neural Networks

After finding the Pareto-optimal dropout vectors, they are stored in a LUT D, along with the corresponding number of MACs. The LUT is small in size (e.g., 25 kB for *DenseNet-100* for storing 64 dropout vectors with 99 dropout values and the corresponding number of MACs, each in 32-bit float format) and stays constant for all rounds. At run time, a device selects the dropout vector d that best corresponds to its current resource availability. When the resource availability changes at the device, the used dropout vectors can be adjusted to these changes at almost zero overhead before every mini-batch. No recompilation, repacking of weights, etc. is required for adapting the resource requirements at run time.

In an FL setting running *DISTREAL*, each device selects its dropout vector at run time according to its current resource availability, as shown in Algorithm 8.1. This is done at the granularity of single mini-batches, i.e., devices can quickly react to changes. Additionally, the server does not need to know the resource availability at each device at the beginning of the round or during the round, reducing signaling overhead, and avoiding the requirement to predict resource availability ahead of time. This enables scalability with the number of devices. At the end of each round, the devices send the weight updates and the computational resources they put into training (number of MACs, as stored in the LUT)

Table 8.2: System configuration for FL.

	FEMNIST	*CIFAR-10*	*CIFAR-100*
#Devices	3,550	100	100
#Samples/device	181±70.7	500	500
Devices/round	35	10	10
NN topology	Simple CNN	*DenseNet-40*	*DenseNet-100*
Resources variability	3×	4×	4×

to the server. The server (Algorithm 8.2) performs a weighted averaging of the received updates by the devices' reported computational resources. Thereby, updates from devices that have trained with lower dropout rates (high resources), are weighted stronger. This is an extension of *FedAvg* [122], which performs weighted averaging only based on the amount of data (number of mini-batches) per device. In the case of the same resource availability on all devices, our coordination technique behaves the same as *FedAvg*. As the type of data exchanged between the devices and the server is not changed compared to *FedAvg*, we can still apply and adopt techniques that mitigate communication aspects, such as compression and sketched updates [123].

8.7 Experimental Evaluation

The evaluation uses the setup described in Section 3.3, which implements a synchronous FL system. It tests the classification accuracy of the synchronized model at the end of each round. The main performance metric is the *convergence speed*, i.e., the test accuracy achieved after a certain number of rounds, but we also report the final accuracy after convergence to demonstrate that the gains in convergence speed do not come at the cost of a lower final accuracy.

The experiments use three datasets: *Federated Extended MNIST* (*FEMNIST*) [166] with non-independently and identically distributed (non-iid) split data, similar to *LEAF* [167], and *CIFAR-10/100* [168]. *FEMNIST* contains 641,828 training and 160,129 test examples, each a 28×28 grayscale image of one out of 62 classes (10 digits, 26 upper-case and 26 lower-case letters). The experiments with *FEMNIST* distribute the data among 3,550 devices, demonstrating the scalability of our solution. *CIFAR-10* contains 50,000 training and 10,000 test examples, each a 32×32 RGB image of one out of 10 classes such as airplane or frogs. *CIFAR-100* is similar to *CIFAR-10* but contains images from 100 classes. We use 100 devices for *CIFAR-10/100*. For *FEMNIST*, we use a similar network as used in *Federated Dropout* [112], with a depth of 4 layers. It requires 4.0 million MACs in the forward pass. We use *DenseNet* [158] for *CIFAR-10* and *CIFAR-100* with growth rate $k = 12$ and a depth of 40 and 100, respectively. This results in 74 million MACs for *CIFAR-10* and 291 million MACs for *CIFAR-100* in the forward pass, respectively. Table 8.2 summarizes these configurations.

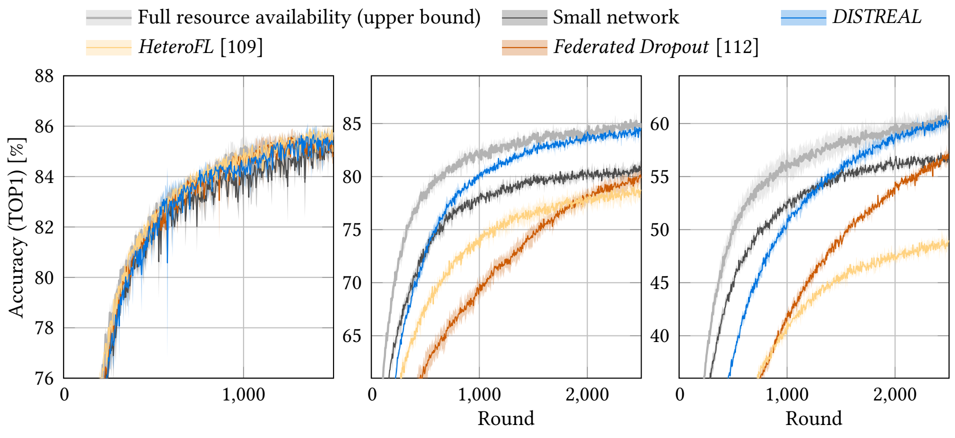
DISTREAL is compared to four baselines:

1. *Full resource availability.* This baseline assumes that all devices have the full resources to train the full NN in each round. This is a theoretical baseline, which serves as an upper bound.
2. *Small network.* The NN complexity is statically reduced to fit the device with the lowest resources. Thereby, each device can train the full (reduced) NN in each round. For *CIFAR-10* and *CIFAR-100*, we reduce the depth of *DenseNet* to 19 and 40, respectively. Because the network of *FEMNIST* already has only a few layers, we reduce the number of filters in the convolutional layers.
3. *Federated Dropout* [112]. Similar to our *DISTREAL*, it uses dropout to reduce the computational complexity. However, the rates are determined by the server at the start of each round. In addition, it uses the same dropout rate for all layers. To have a fair comparison, we extend the technique to allow for different dropout rates for different devices according to their resource availability at the beginning of the round.
4. *HeteroFL* [109]. It uses a shrinking ratio $0<s<1$. The NN is divided into several levels $p = 1, 2, \ldots$, where level p reduces the width of each hidden channel (i.e., number of filters) to a fraction s^{p-1}. Like in *Federated Dropout*, reducing the model is done by the server at the beginning of each round. The work in [109] provides no details on how to set s. We use $s = 0.7$, as it shows the best performance.

8.7.1 Heterogeneity Across Devices

The first experiments study heterogeneity across devices, i.e., devices have different resource availability but for now, there are no changes over time. Each device is assigned a random resource availability that is sampled uniformly from a range with the upper bound being selected such that training the full NN without dropout is possible. The variability in the resource availability, i.e., ratio of minimum to maximum possible resource availability, is reported in Table 8.2. We repeat every experiment three times and report the average and standard deviations of the test accuracy.

Fig. 8.7 shows the accuracy results. *FEMNIST* uses a simple network. We observe that the small network baseline has the lowest convergence speed, while all other techniques show a similar performance. The varying quantity (see Table 8.2) and distribution of local training data on the devices make the training noisy, as a different random set of devices participates in the training in each round. Nevertheless, *DISTREAL* is not more sensitive to such non-iid data than other solutions and reaches the same convergence speed and final accuracy. With *CIFAR-10*, *DISTREAL* achieves a significantly higher convergence speed than *Federated Dropout* or *HeteroFL* and almost reaches the accuracy of the theoretical baseline with full resource availability after 2,000 rounds. The simple baseline that uses a smaller network on all devices initially converges faster but then saturates early. Similar observations can be made with *CIFAR-100*. Table 8.3 reports the final accuracy, where we train with each technique for 7,500 rounds. This ensures that all techniques have fully



(a) *FEMNIST* (non-iid) **(b)** *CIFAR-10* **(c)** *CIFAR-100*

Figure 8.7: Convergence during FL on heterogeneous devices. *DISTREAL* improves the convergence speed.

Table 8.3: Final accuracy (after 7,500 rounds). *DISTREAL* reaches the same or a higher final accuracy than the state of the art.

	FEMNIST	*CIFAR-10*	*CIFAR-100*
Full resource availability (upper bound)	87.4±0.3	87.8±0.1	65.6±0.5
Small NN	86.4±0.1	82.2±0.6	57.8±0.5
DISTREAL	86.7±0.1	<u>85.7±0.3</u>	<u>65.7±0.5</u>
HeteroFL [109]	<u>87.1±0.4</u>	81.2±0.5	52.2±0.6
Federated Dropout [112]	86.9±0.3	84.2±0.3	65.3±0.5

converged. *DISTREAL* and *Federated Dropout* reach the highest final accuracy, similar to full resource availability, up to 13 % higher than *HeteroFL*.

As the resources do not change over time, the main contributions of *DISTREAL* in this scenario are the application of the DSE, which enables devices to efficiently utilize the available resources and the fact that *DISTREAL* applies a probabilistic approach and drops different filters in different mini-batches, allowing to support a large number of fine-grained resource levels. *Federated Dropout* uses the same dropout rates for all layers, preventing it from efficiently utilize the available resources. *HeteroFL* supports only a few resource levels, which prevents it from fully exploiting the resources, and removes the filters in always the same order. *DISTREAL* is the only technique to fully utilize the available resources, and most efficiently use these resources w.r.t. convergence. In addition, we observe higher relative gains with increasing NN model complexity.

8.7.2 Heterogeneity Across Devices and Over Time

This section studies a fully heterogeneous FL system, i.e., the resource availability varies between devices and for each device over time. This is the main experiment in this chapter. As discussed in Section 8.1, this is for instance due to shared resource contention between the learning task and other tasks. As discussed earlier, the available resources for learning may change at any time, i.e., also in the middle of a round, and may be unpredictable to the device. We model them as random, with the time between consecutive changes following an exponential distribution with rate parameter λ. Thereby, the number of the changes in a round follows a Poisson distribution with rate parameter λ. The resource availability levels are sampled from the same range as in the previous section, i.e., also according to Table 8.3. The average resource availability across all devices and over time is the same as in the previous section. We study four values of $\lambda \in \{0.5, 1, 2, 4\}$, to simulate a range of slowly to rapidly changing scenarios.

Fig. 8.8 shows the convergence for *CIFAR-10* and *CIFAR-100*. We also plot the convergence with constant resources (previous results from Fig. 8.7) for reference. We observe that the convergence speed with *DISTREAL* is independent of the rate of resource changes and almost matches the results of the previous section. In contrast, the convergence speeds of *HeteroFL* and *Federated Dropout* significantly degrade with a higher λ. For instance, *DISTREAL* consistently reaches 50 % accuracy on *CIFAR-100* with $\lambda = 4$ after around 1,100 rounds, i.e., around 2.5× faster than *HeteroFL*, which requires around 2,700 rounds. In addition, *DISTREAL* reaches the highest final accuracy (shown in Table 8.4), independently of λ. The baselines with full resource availability or a small model perform the same as in Fig. 8.7, independently of λ, and are not shown again.

HeteroFL and *Federated Dropout* both select the trained model subsets per device at the server at the start of a round. The devices train on the assigned subset for the whole round and, hence, cannot react to potential changes in the resource availability during a round. An increase in the resource availability results in underutilization of the available resources, as the training finishes early and the device is idle until the end of the round. A decrease in resource availability results in the training not finishing in time (i.e., the device becomes a straggler), leading to the device being dropped from the round, completely wasting the available resources on this device in this round. In contrast, *DISTREAL* dynamically adjusts the resource requirements of training at run time locally on each device by selecting a different dropout vector when a change occurs, always finishing the training in time and fully utilizing the available resources.

These results show the importance of tackling the challenges discussed in Section 8.2, i.e., to *fully* and *efficiently* utilize available resources on each device. *DISTREAL* achieves this by performing dropout at the devices, enabling them to react fast to the changes in a fine-grained manner. This enables to fully utilize all available resources, making convergence robust to unpredictable changes in the resource availability. Furthermore, the DSE enables *DISTREAL* to efficiently utilize the available resources by finding Pareto-optimal dropout vectors w.r.t. the resource requirements and the achieved convergence speed. These gains in convergence speed do not come at the cost of a lower final accuracy.

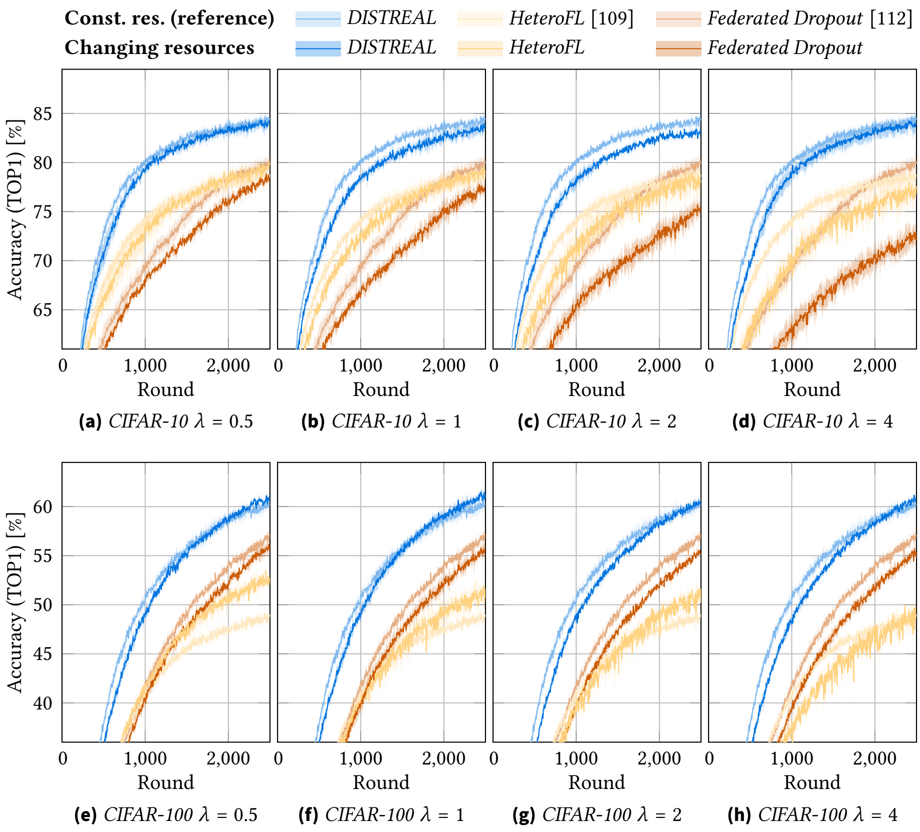


Figure 8.8: Convergence with *CIFAR-10* and *CIFAR-100* on heterogeneous devices where resources availability changes randomly over the time with varying rate parameter λ. *DISTREAL* achieves a higher convergence speed than the state of the art, independently of λ. Experiments with full resource availability or with a small network are not repeated as they perform the same as in Fig. 8.7.

Table 8.4: Final accuracy (after 7,500 rounds) for *CIFAR-10* and *CIFAR-100* with changing resources. *DISTREAL* reaches the highest final accuracy.

	CIFAR-10				*CIFAR-100*			
	λ = 0.5	λ = 1	λ = 2	λ = 4	λ = 0.5	λ = 1	λ = 2	λ = 4
DISTREAL	86.3±0.3	86.0±0.2	85.1±0.3	85.9±0.4	65.4±0.4	65.5±0.9	64.7±0.3	64.7±0.3
HeteroFL [109]	82.6±0.2	81.9±1.3	81.7±0.5	81.3±0.3	57.2±0.4	56.8±1.0	56.5±1.0	56.6±1.0
Fed. Dropout [112]	83.4±0.6	83.2±0.4	82.4±0.4	82.7±0.6	65.0±0.3	64.5±0.2	64.5±0.3	64.5±0.3

8.8 Summary

This chapter addressed the problem of training of NNs under dynamic heterogeneous resource availability. In particular, the training task often needs to share computational resources with other tasks running on the system. This leads to the available computational resources for training *changing over time*. The presented work is the first work to tackle this problem. Maximizing the convergence speed can only be achieved by *fully* and *efficiently* using the available resources. The proposed technique *DISTREAL* achieves this by 1) employing structured dropout to make a fine-grained trade-off between resource requirements of training and the achieved convergence speed, 2) employing different per-layer dropout rates and finding the Pareto-optimal vectors w.r.t. resources requirements and convergence speed with a DSE, and 3) adjusting the dropout rates locally and fast on each device, without requiring any assistance from the server. *DISTREAL* is evaluated in a heterogeneous FL setting, where many devices perform distributed training of an NN. Thereby, resources vary both between devices *and over time*. The results show that *DISTREAL* significantly improves the convergence speed without compromising on the final accuracy.

This chapter focused on computational resources and did not take communication limitations into consideration. However, as we do not change the type of data exchanged between the devices and the server, compared to *FedAvg*, we can still apply and adopt techniques that have been proposed to mitigate communication aspects, such as compression and sketched updates [123]. Besides, *DISTREAL* is currently tailored towards a CNN. However, it would also be applicable with minor changes to other NN types, such as fully-connected networks. Finally, the supported range of resource availability is limited by the maximum dropout rate. We studied dropout rates up to 0.5, i.e., up to 4× variability in the resources.

9 Conclusion

This dissertation tackles the problem of optimizing the management of limited resources in computing systems using machine learning (ML). In particular, the main focus of this dissertation is on system-level resource management, i.e., applying ML to operating system (OS)-level optimization using application mapping, application migration, and dynamic voltage and frequency scaling (DVFS) for performance maximization under a thermal constraint, or temperature minimization under a quality of service (QoS) constraint.

Chapter 4 presents a classical technique (without ML) for application mapping and DVFS using power budgets to gain a deeper understanding of the involved challenges. It tackles the problem of performance maximization on thermally-constrained many-core processors with distributed shared last-level cache (LLC) and presents the first work to exploit the trade-off between power budget maximization and LLC latency minimization defined by the application mapping on such platforms. Even though the presented technique demonstrates improvements over the state of the art, it still has several shortcomings. The developed classical heuristic for application mapping is suboptimal as it ignores application characteristics. In addition, it is not applicable to different cooling or thermal constraints as the underlying trade-off would change. This is as the heuristic ignores parts of the underlying complexity in applications and the platform. The heuristic power budget reallocation algorithm is also suboptimal and a higher performance would be achievable when proactively throttling some applications to boost others. These shortcomings can not easily be tackled, as the complexity of the heuristics would increase impractically.

ML can tackle some of the main challenges in resource management, i.e., i) coping with the high complexity of applications and platforms, ii) coping with unseen (not known at design time) scenarios in the workload and platform configuration, iii) achieving proactive management, and iv) maintaining a low run-time overhead. Three main patterns have been identified to employ ML in resource management: 1) predict the impact of a management action before executing it, 2) estimate hidden properties of applications and platform, and 3) directly learn the resource management policy.

Chapter 5 studies a similar problem as in Chapter 4, addressing its shortcomings. The presented technique employs application migration, taking into account the application characteristics and overall workload. Thereby, it finds per each application the mapping that makes the optimal trade-off between power budget and LLC latency. This is achieved with a neural network (NN) model to predict the performance impact of a potential migration before executing it. Only the migration with the best predicted impact is executed. Proactive management is achieved by executing no migration before predicting its impact. The model is trained to cope with unseen applications, i.e., not used during training, which is achieved by using general performance counters as features, and the technique generalizes to different thermal constraints. In addition, the model achieves a better trade-off between accuracy and overhead than analytical modeling based on cycles per instruction (CPI) stacks, without requiring deep insights into the internals of the platform. This technique outperforms the classical heuristic management presented in Chapter 4. However, it still performs DVFS with power budgets, which simplifies the problem but prevents further optimization.

The technique presented in Chapter 6 further studies the problem of maximizing the performance under a thermal constraint using DVFS (boosting). A key observation in this technique is that boosting needs to consider the sensitivities of performance, power, and temperature on voltage/frequency (V/f) levels, which vary depending on the application and current mapping. This is tackled with a *boostability* metric to integrate all three required factors. The sensitivities of performance and power are not measurable at run time. Therefore, an NN model is designed, trained, and employed to estimate these unobservable (hidden) properties of unseen applications at run time. By providing estimates of hidden properties of applications, employing ML helps tackle the complexity of the problem. ML-based modeling achieves a higher prediction accuracy than analytical models, which enables achieving a higher performance but still thermally-safe operation. Therefore, the presented technique outperforms classical heuristic management and analytical management.

Chapter 7 tackles the problem of joint application migration and DVFS on heterogeneous clustered multi-core processors to minimize the temperature under QoS targets. This is challenging, as a global optimization is required to consider all running applications and their respective QoS targets, and different applications require different optimal migrations. The problem is solved with a two-part algorithm that employs NN-based imitation learning (IL) to decide on application migrations to change their mapping and employs a simple control loop to determine the V/f levels for the given mapping. IL enables combining the optimality of oracle demonstrations with a low run-time overhead. The NN generalizes to unseen applications and even different cooling settings. This technique is also the first to accelerate the run-time inference of NN-based resource management using a generic, already existing NN accelerator, a neural processing unit (NPU), to reduce the run-time overhead. Finally, this chapter shows that IL-based management achieves better and more stable management than reinforcement learning (RL), and also outperforms classical heuristic management.

The previous works all perform training at design time. In contrast, Chapter 8 targets run-time training, which may provide additional adaptability. One of the main observations in this chapter is that the available resources for training at run time vary over time due to, e.g., shared resource contention with other applications running on the system. This requires that the training is dynamically adjustable to varying resources. The presented solution achieves adjustability by dynamically dropping parts of the NN during training using structured dropout to save computations. The Pareto-optimal per-layer dropout rates are determined using a design space exploration (DSE). The proposed technique is evaluated in a distributed training setting using federated learning (FL), where the available computational resources vary between devices and on every device over time.

In summary, this dissertation and the presented techniques showed that ML is a key technology to tackle the main challenges in resource management, thereby improving the overall efficiency, e.g., improving the achievable performance.

9.1 Future Work

This dissertation opens several directions for future research. This section outlines two promising directions: generalize across platforms and increase the run-time adaptability.

Generalization Across Platforms A major focus in all the resource management techniques developed within this dissertation is the generalization to unseen workloads with unknown applications and unknown input data, which is essential in open systems. However, the developed techniques are still specific to a certain hardware platform. While this dissertation has presented the first steps toward generalization to different platforms, e.g., by studying variations in the thermal constraint (Chapter 5) and cooling settings (Chapter 7), more research is needed.

Generalization to different hardware platforms, where, for instance, a model is trained on a processor of one generation and could still be employed for management of a processor of the next generation, would greatly enhance the applicability of the techniques. This is as it would eliminate the requirement to obtain training data on every potential platform. A first step could be to employ transfer learning [169], where a model is trained on one platform (requiring a full set of training data), and then re-trained on another platform using only a small amount of training data. Transfer learning exploits similarities between problems, and, therefore, only works if platforms have some similarity. The ultimate goal would be to train a single model that generalizes to various platforms. The main challenges are finding general features that accurately describe a platform, as well as acquiring a representative set of training data from various platforms. In addition, this may come at the cost of requiring a more complex model, adversely affecting the run-time overhead.

Increase Adaptability through Run-time Learning Generalization to different scenarios (workloads or even platforms) is necessary if the same ML model is deployed to various instances that experience different scenarios. Such generalization comes at the cost of requiring representative training data for various scenarios and potentially increasing the model complexity. However, a specific instance only needs to cope with the specific scenarios that occur on it.

This could be achieved through run-time (online) learning, where the employed model is continuously updated based on observations during run time. However, several challenges need to be tackled. First, it must be possible to create labels at run time. This is straightforward when employing ML to predict the impact of management actions, where the real impact can be observed after executing the action. This is also possible with RL. However, it is difficult for instance for IL, which requires oracle demonstrations for training. Second, Chapter 7 has already demonstrated the importance of run-time stability, i.e., updating the model should not result in temporary bad management. This could not be achieved with RL. In addition, there is a trade-off between adapting fast to changes in the observed scenarios due to concept drift [170], but still achieving good management for scenarios that have not been observed for a long time but still might eventually happen again. Finally, the run-time learning comes at the cost of increased overhead for additional computations and storing not only the model but also training data. One way to solve this is to perform training only with free resources that are not used by other applications, applying the technique presented in Chapter 8.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AMD	Average Manhattan distance
AoI	Application of interest
CNN	Convolutional neural network
CPI	Cycles per instruction
CPU	Central processing unit
DRAM	Dynamic random access memory
DRL	Deep reinforcement learning
DSE	Design space exploration
DSP	Digital signal processor
DTM	Dynamic thermal management
DVFS	Dynamic voltage and frequency scaling
EDA	Embedded design automation
FL	Federated learning
FSM	Finite state machine
GTS	Global Task Scheduling
IL	Imitation learning
ILP	Integer linear program
IoT	Internet of things
IPC	Instructions per cycle
IPS	Instructions per second
LLC	Last-level cache
LUT	Lookup table
MAC	Multiply-accumulate
MD	Manhattan distance
MDP	Markov decision process
ML	Machine learning

MLP	Memory-level parallelism
MSE	Mean squared error
NAS	Neural architecture search
NN	Neural network
NoC	Network-on-chip
non-iid	Non-independently and identically distributed
NPU	Neural processing unit
OS	Operating system
QoS	Quality of service
RAM	Random access memory
RL	Reinforcement learning
RMSE	Root-mean-square error
S-NUCA	Static non-uniform cache access
SoC	System-on-chip
TDP	Thermal Design Power
TSP	Thermal Safe Power
V/f	Voltage/frequency

Bibliography

[1] Martin Rapp, Heba Khdr, Nikita Krohmer, and Jörg Henkel. “NPU-Accelerated Imitation Learning for Thermal Optimization of QoS-Constrained Heterogeneous Multi-Cores”. In: *arXiv preprint arXiv:2206.05459* (2022).

[2] Martin Rapp, Nikita Krohmer, Heba Khdr, and Jörg Henkel. “NPU-Accelerated Imitation Learning for Thermal- and QoS-Aware Optimization of Heterogeneous Multi-Cores”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022. DOI: `10.23919/DATE54114.2022.9774681`.

[3] Martin Rapp, Ramin Khalili, Kilian Pfeiffer, and Jörg Henkel. “DISTREAL: Distributed Resource-Aware Learning in Heterogeneous Systems”. In: *AAAI Conference on Artificial Intelligence (AAAI)*. 2022.

[4] Martin Rapp, Hussam Amrouch, Yibo Lin, Bei Yu, David Pan, Marilyn Wolf, and Jörg Henkel. “MLCAD: A Survey of Research in Machine Learning for CAD (Keynote Paper)”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021). DOI: `10.1109/TCAD.2021.3124762`.

[5] Martin Rapp, Mohammed Bakr Sikal, Heba Khdr, and Jörg Henkel. “SmartBoost: Lightweight ML-Driven Boosting for Thermally-Constrained Many-Core Processors”. In: *Design Automation Conference (DAC)*. 2021. DOI: `10.1109/DAC18074.2021.9586287`.

[6] Martin Rapp, Anuj Pathania, Tulika Mitra, and Jörg Henkel. “Neural Network-based Performance Prediction for Task Migration on S-NUCA Many-Cores”. In: *IEEE Transactions on Computers (TC)* 70.10 (2021). DOI: `10.1109/TC.2020.3023022`.

[7] Martin Rapp, Mark Sagi, Anuj Pathania, Andreas Herkersdorf, and Jörg Henkel. “Power-and Cache-Aware Task Mapping with Dynamic Power Budgeting for Many-Cores”. In: *IEEE Transactions on Computers (TC)* 69.1 (2020). DOI: `10.1109/TC.2019.2935446`.

[8] Martin Rapp, Anuj Pathania, Tulika Mitra, and Jörg Henkel. “Prediction-Based Task Migration on S-NUCA Many-Cores”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019. DOI: `10.23919/DATE.2019.8714974`.

[9] Martin Rapp, Anuj Pathania, and Jörg Henkel. “Pareto-Optimal Power- and Cache-Aware Task Mapping for Many-Cores with Distributed Shared Last-Level Cache”. In: *Int. Symp. on Low Power Electronics and Design (ISLPED)*. ACM/IEEE. 2018. DOI: `10.1145/3218603.3218630`.

[10] Martin Rapp, Omar Elfatairy, Marilyn C. Wolf, Jörg Henkel, and Hussam Amrouch. "Towards NN-based Online Estimation of the Full-Chip Temperature and the Rate of Temperature Change". In: *Workshop on Machine Learning for CAD (MLCAD)*. 2020, pp. 95–100. DOI: `10.1145/3380446.3430648`.

[11] Martin Rapp, Ramin Khalili, and Jörg Henkel. "Distributed Learning on Heterogeneous Resource-Constrained Devices". In: *arXiv preprint arXiv:2006.05403* (2020).

[12] Martin Rapp, Sami Salamin, Hussam Amrouch, Girish Pahwa, Yogesh Chauhan, and Jörg Henkel. "Performance, Power and Cooling Trade-Offs with NCFET-based Many-Cores". In: *Design Automation Conference (DAC)*. 2019. DOI: `10.1145/3316781.3317880`.

[13] Martin Rapp, Hussam Amrouch, Marilyn C. Wolf, and Jörg Henkel. "Machine Learning Techniques to Support Many-Core Resource Management: Challenges and Opportunities". In: *Workshop on Machine Learning for CAD (MLCAD)*. ACM/IEEE. 2019. DOI: `10.1109/MLCAD48534.2019.9142064`.

[14] Lokesh Siddhu, Rajesh Kedia, Shailja Pandey, Martin Rapp, Anuj Pathania, Jörg Henkel, and Preeti Ranjan Panda. "CoMeT: An Integrated Interval Thermal Simulation Toolchain for 2D, 2.5 D, and 3D Processor-Memory Systems". In: *ACM Transactions on Architecture and Code Optimization (TACO)* (2022).

[15] Marcel Mettler, Martin Rapp, Heba Khdr, Daniel Müller-Gritschneder, and Jörg Henkel. "An FPGA-based Approach to Evaluate Thermal and Resource Management Strategies of Many-Core Processors". In: *ACM Transactions on Architecture and Code Optimization (TACO)* (2022).

[16] Mohammed Bakr Sikal, Heba Khdr, Martin Rapp, and Jörg Henkel. "Thermal- and Cache-Aware Resource Management based on ML-Driven Cache Contention Prediction". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022.

[17] Kilian Pfeiffer, Martin Rapp, Ramin Khalili, and Jörg Henkel. "CoCo-FL: Communication- and Computation-Aware Federated Learning via Partial NN Freezing and Quantization". In: *arXiv preprint arXiv:2203.05468* (2022).

[18] Veera Venkata Ram Murali Krishna Rao Muvva, Martin Rapp, Jörg Henkel, Hussam Amrouch, and Marilyn C. Wolf. "On the Effectiveness of Quantization and Pruning on the Performance of FPGAs-based NN Temperature Estimation". In: *Workshop on Machine Learning for CAD (MLCAD)*. 2021.

[19] Mark Sagi, Martin Rapp, Heba Khdr, Yizhe Zhang, Nael Fasfous, Nguyen Anh Vu Doan, Thomas Wild, Jörg Henkel, and Andreas Herkersdorf. "Long Short-Term Memory Neural Network-based Power Forecasting of Multi-Core Processors". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 1685–1690.

[20] Sami Salamin, Victor M Van Santen, Martin Rapp, Jörg Henkel, and Hussam Amrouch. “Minimizing Excess Timing Guard Banding Under Transistor Self-Heating Through Biasing at Zero-Temperature Coefficient”. In: *IEEE Access* 9 (2021), pp. 30687–30697.

[21] Hussam Amrouch, Martin Rapp, Sami Salamin, and Jörg Henkel. “Impact of Negative Capacitance Field-Effect Transistor (NCFET) on Many-Core Systems”. In: *A Journey of Embedded and Cyber-Physical Systems*. Springer, 2021, pp. 107–123.

[22] Mark Sagi, Nguyen Anh Vu Doan, Martin Rapp, Thomas Wild, Jörg Henkel, and Andreas Herkersdorf. “A Lightweight Nonlinear Methodology to Accurately Model Multi-Core Processor Power”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39.11 (2020), pp. 3152–3164.

[23] Sami Salamin, Martin Rapp, Jörg Henkel, Andreas Gerstlauer, and Hussam Amrouch. “Dynamic Power and Energy Management for NCFET-Based Processors”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39.11 (2020), pp. 3361–3372.

[24] Behnaz Pourmohseni, Michael Glaß, Jörg Henkel, Heba Khdr, Martin Rapp, Valentina Richthammer, Tobias Schwarzer, Fedor Smirnov, Jan Spieck, Jürgen Teich, Andreas Weichslgartner, and Stefan Wildermann. “Hybrid Application Mapping for Composable Many-Core Systems: Overview and Future Perspective”. In: *Journal of Low Power Electronics and Applications (JLPEA)* 10.4 (2020).

[25] Sami Salamin, Martin Rapp, Anuj Pathania, Arka Maity, Jörg Henkel, Tulika Mitra, and Hussam Amrouch. “Power-Efficient Heterogeneous Many-Core Design with NCFET Technology”. In: *IEEE Transactions on Computers (TC)* 70.9 (2020), pp. 1484–1497.

[26] Sami Salamin, Martin Rapp, Hussam Amrouch, Andreas Gerstlauer, and Jörg Henkel. “Energy Optimization in NCFET-based Processors”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 630–633.

[27] Marvin Damschen, Martin Rapp, Lars Bauer, and Jörg Henkel. “i-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems”. In: *Embedded, Cyber-Physical, and IoT Systems*. Springer, 2020.

[28] Jörg Henkel, Hussam Amrouch, Martin Rapp, Sami Salamin, Dayane Reis, Di Gao, Xunzhao Yin, Michael Niemier, Cheng Zhuo, X Sharon Hu, Hsiang-Yun Cheng, and Chia-Lin Yang. “The Impact of Emerging Technologies on Architectures and System-level Management”. In: *International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2019.

[29] Sami Salamin, Martin Rapp, Hussam Amrouch, Girish Pahwa, Yogesh Chauhan, and Jörg Henkel. “NCFET-aware Voltage Scaling”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE. 2019.

[30] Jörg Henkel, Heba Khdr, and Martin Rapp. “Smart Thermal Management for Heterogeneous Multicores”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 132–137.

[31] Jörg Henkel, Andreas Herkersdorf, Lars Bauer, Thomas Wild, Michael Hübner, Ravi Kumar Pujari, Artjom Grudnitsky, Jan Heisswolf, Aurang Zaib, Benjamin Vogel, Vahid Lari, and Sebastian Kobbe. “Invasive Manycore Architectures”. In: *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2012, pp. 193–200.

[32] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Springer Nature, 2021.

[33] Jörg Henkel, Heba Khdr, Santiago Pagani, and Muhammad Shafique. “New Trends in Dark Silicon”. In: *Design Automation Conference (DAC)*. IEEE. 2015.

[34] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark Silicon and the End of Multicore Scaling”. In: *International Symposium on Computer Architecture (ISCA)*. IEEE. 2011, pp. 365–376.

[35] Mohssen Mohammed, Muhammad Badruddin Khan, and Eihab Bashier Mohammed Bashier. *Machine Learning: Algorithms and Applications*. CRC Press, 2016.

[36] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. 2018.

[37] Boyi Liu, Lujia Wang, and Ming Liu. “Lifelong Federated Reinforcement Learning: A Learning Architecture for Navigation in Cloud Robotic Systems”. In: *IEEE Robotics and Automation Letters* 4.4 (2019), pp. 4555–4562.

[38] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. “Imagenet: A Large-Scale Hierarchical Image Database”. In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2009, pp. 248–255.

[39] Yibo Lin, Zixuan Jiang, Jiaqi Gu, Wuxi Li, Shounak Dhar, Haoxing Ren, Brucek Khailany, and David Z Pan. “DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 40.4 (2020), pp. 748–761.

[40] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M Frans Kaashoek, Robert Morris, et al. “Corey: An Operating System for Many Cores”. In: *Symp. Operating System Design and Implementation (OSDI)*. 2008.

[41] *Cortex-A73 MPCore Processor Technical Reference Manual. Revision r1p0.* Arm Limited. 2018.

[42] *Cortex-A53 MPCore Processor Technical Reference Manual. Revision r0p4.* Arm Limited. 2018.

[43] Changkyu Kim, Doug Burger, and Stephen W Keckler. “An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches”. In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. 2002, pp. 211–222. DOI: 10.1145/635508.605420.

[44] Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R Stan. “HotSpot: A Compact Thermal Modeling Methodology for Early-Stage VLSI Design”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.5 (2006), pp. 501–513. DOI: 10.1109/TVLSI.2006.876103.

[45] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions”. In: *IEEE Journal of Solid-State Circuits (JSSC)* 9.5 (1974), pp. 256–268.

[46] Michael R Garey and David S. Johnson. “Complexity Results for Multiprocessor Scheduling under Resource Constraints”. In: *SIAM Journal on Computing* (1975).

[47] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. ACM. 2008, pp. 72–81. DOI: 10.1145/1454115.1454128.

[48] Dror G Feitelson and Larry Rudolph. “Metrics and Benchmarking for Parallel Job Scheduling”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1998.

[49] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. “The SPLASH-2 Programs: Characterization and Methodological Considerations”. In: *Int. Symp. Computer Architecture (ISCA)* (1995).

[50] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S Awwal, and Vijayan K Asari. “The History began from AlexNet: A Comprehensive Survey on Deep Learning Approaches”. In: *arXiv preprint arXiv:1803.01164* (2018).

[51] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. “Dropout: a Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research (JMLR)* 15.1 (2014), pp. 1929–1958.

[52] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. “Human-Level Control through Deep Reinforcement Learning”. In: *Nature* 518.7540 (2015), pp. 529–533.

[53] Ayse Kivilcim Coskun, Tajana Simunic Rosing, Keith A Whisnant, and Kenny C Gross. “Temperature-Aware MPSoC Scheduling for Reducing Hot Spots and Gradients”. In: *Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE Computer Society. 2008, pp. 49–54.

[54] Heba Khdr, Santiago Pagani, Muhammad Shafique, and Jörg Henkel. “Thermal Constrained Resource Management for Mixed ILP-TLP Workloads in Dark Silicon Chips”. In: *Design Automation Conference (DAC).* ACM. 2015.

[55] Daniel Olsen and Iraklis Anagnostopoulos. “Performance-Aware Resource Management of Multi-Threaded Applications on Many-Core Systems”. In: *Great Lakes Symposium on VLSI (GLSVLSI).* 2017, pp. 119–124.

[56] *Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors.* Intel Corporation. 2008.

[57] Brian Jeff. “big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling”. In: *ARM white paper* (2013).

[58] Venkatesh Pallipadi and Alexey Starikovskiy. “The ondemand Governor”. In: *Proceedings of the Linux Symposium.* Vol. 2. 00216. 2006, pp. 215–230.

[59] Shi Sha, Wujie Wen, Shaolei Ren, and Gang Quan. “M-Oscillating: Performance Maximization on Temperature-Constrained Multi-Core Processors”. In: *IEEE Trans. Parallel and Distributed Systems (TPDS)* 29.11 (2018), pp. 2528–2539.

[60] Anuj Pathania and Jörg Henkel. “Task Scheduling for Many-Cores with S-NUCA Caches”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2018, pp. 557–562. DOI: 10.23919/DATE.2018.8342069.

[61] Anil Kanduri, Mohammad-Hashem Haghbayan, Amir M Rahmani, Muhammad Shafique, Axel Jantsch, and Pasi Liljeberg. “adBoost: Thermal Aware Performance Boosting through Dark Silicon Patterning”. In: *IEEE Transactions on Computers (TC)* 1 (2018).

[62] Jim Ng, Xiaohang Wang, Amit Kumar Singh, and Terrence Mak. “Defragmentation for Efficient Runtime Resource Management in NoC-Based Many-Core Systems”. In: *Transactions on Very Large Scale Integration (VLSI) Systems* 24.11 (2016), pp. 3359–3372.

[63] Di Zhu, Lizhong Chen, Timothy M Pinkston, and Massoud Pedram. “TAPP: Temperature-Aware Application Mapping for NoC-Based Many-Core Processors”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE).* EDA Consortium. 2015, pp. 1241–1244.

[64] Xiaohang Wang, Amit Kumar Singh, Bing Li, Yang Yang, Hong Li, and Terrence Mak. “Bubble Budgeting: Throughput Optimization for Dynamic Workloads by Exploiting Dark Cores in Many Core Systems”. In: *IEEE Transactions on Computers (TC)* 67.2 (2018), pp. 178–192.

[65] Xiaohang Wang, Baoxin Zhao, Ling Wang, Terrence Mak, Mei Yang, Yingtao Jiang, and Masoud Daneshtalab. “A Pareto-Optimal Runtime Power Budgeting Scheme for Many-Core Systems”. In: *Microprocessors and Microsystems* 46 (2016), pp. 136–148.

[66] Heba Khdr, Hussam Amrouch, and Jörg Henkel. “Aging-Constrained Performance Optimization for Multi Cores”. In: *Design Automation Conference (DAC).* IEEE. 2018.

[67] Guangshuo Liu, Jinpyo Park, and Diana Marculescu. “Procrustes 1: Power Constrained Performance Improvement using Extended Maximize-then-Swap Algorithm”. In: *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. (TCAD)* 34.10 (2015), pp. 1664–1676.

[68] Can Hankendi and Ayse Kivilcim Coskun. “Scale & Cap: Scaling-Aware Resource Management for Consolidated Multi-Threaded Applications”. In: *ACM Trans. Design Automation of Electronic Systems (TODAES)* 22.2 (2017).

[69] Hai Wang, Wei Li, Wenjie Qi, Diya Tang, Letian Huang, and He Tang. “Runtime Performance Optimization of 3-D Microprocessors in Dark Silicon”. In: *IEEE Trans. Computers (TC)* (2020).

[70] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. “Scheduling Heterogeneous Multi-Cores Through Performance Impact Estimation (PIE)”. In: *SIGARCH Computer Architecture News*. Vol. 40. 3. IEEE Computer Society. 2012, pp. 213–224. DOI: 10.1145/2366231.2337184.

[71] Mihai Pricopi, Thannirmalai Somu Muthukaruppan, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. “Power-Performance Modeling on Asymmetric Multi-Cores”. In: *Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE. 2013. DOI: 10.1109/CASES.2013.6662519.

[72] Ganapati Bhat, Gaurav Singla, Ali K Unver, and Umit Y Ogras. “Algorithmic Optimization of Thermal and Power Management for Heterogeneous Mobile Platforms”. In: *IEEE Trans. Very Large Scale Integration (VLSI) Systems* 26.3 (2017), pp. 544–557.

[73] Santiago Pagani, Sai Manoj PD, Axel Jantsch, and Jörg Henkel. “Machine Learning for Power, Energy, and Thermal Management on Multi-Core Processors: A Survey”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2018). DOI: 10.1109/TCAD.2018.2878168.

[74] Anup Das, Rishad A Shafik, Geoff V Merrett, Bashir M Al-Hashimi, Akash Kumar, and Bharadwaj Veeravalli. “Reinforcement Learning-based Inter-and Intra-Application Thermal Optimization for Lifetime Improvement of Multicore Systems”. In: *Design Automation Conference (DAC)*. 2014.

[75] Rishad A Shafik, Sheng Yang, Anup Das, Luis A Maeda-Nunez, Geoff V Merrett, and Bashir M Al-Hashimi. “Learning Transfer-Based Adaptive Energy Minimization in Embedded Systems”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 35.6 (2015), pp. 877–890.

[76] Sai Manoj Pudukotai Dinakarrao, Arun Joseph, Anand Haridass, Muhammad Shafique, Jörg Henkel, and Houman Homayoun. “Application and Thermal-Reliability-Aware Reinforcement Learning based Multi-Core Power Management”. In: *ACM Jrnl. on Emerging Tech. in Computing Systems (JETC)* 15.4 (2019).

[77] Eunji Kwon, Sodam Han, Yoonho Park, Jongho Yoon, and Seokhyeong Kang. "Reinforcement Learning-Based Power Management Policy for Mobile Device Systems". In: *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I)* (2021).

[78] Di Liu, Shi-Gui Yang, Zhenli He, Mingxiong Zhao, and Weichen Liu. "CARTAD: Compiler-Assisted Reinforcement Learning for Thermal-Aware Task Scheduling and DVFS on Multicores". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021).

[79] Shiting Lu, Russell Tessier, and Wayne Burleson. "Reinforcement Learning for Thermal-Aware Many-Core Task Allocation". In: *Great Lakes Symp. on VLSI (GLSVLI)*. 2015, pp. 379–384.

[80] Ujjwal Gupta, Sumit K Mandal, Manqing Mao, Chaitali Chakrabarti, and Umit Y Ogras. "A Deep Q-Learning Approach for Dynamic Management of Heterogeneous Processors". In: *Computer Architecture Letters (CAL)* 18.1 (2019), pp. 14–17.

[81] Shi-Gui Yang, Yuan-Yuan Wang, Di Liu, Xu Jiang, Hui Fang, Yu Yang, and Mingxiong Zhao. "ReLeTa: Reinforcement Learning for Thermal-Aware Task Allocation on Multicore". In: *arXiv preprint arXiv:1912.00189* (2019).

[82] Zhuo Chen, Dimitrios Stamoulis, Student Member, and Diana Marculescu. "Profit: Priority and Power / Performance Optimization for Many-Core Systems". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.10 (2018), pp. 2064–2075.

[83] Bryan Donyanavard, Armin Sadighi, Florian Maurer, Tiago Mück, Amir Rahmani, Andreas Herkersdorf, and Nikil Dutt. "SOSA: Self-optimizing Learning with Self-adaptive Control for Hierarchical SoC Management". In: *Int. Symp. on Microarchitecture (MICRO)* (2019).

[84] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. "Concrete Problems in AI Safety". In: *arXiv preprint arXiv:1606.06565* (2016).

[85] Ujjwal Gupta, Chetan Arvind Patil, Ganapati Bhat, Prabhat Mishra, and Umit Y Ogras. "DyPO: Dynamic Pareto-Optimal Configuration Selection for Heterogeneous MPSoCs". In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (2017).

[86] Ryan Gary Kim, Wonje Choi, Zhuo Chen, Janardhan Rao Doppa, Partha Pratim Pande, Diana Marculescu, and Radu Marculescu. "Imitation Learning for Dynamic VFI Control in Large-Scale Manycore Systems". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.9 (2017), pp. 2458–2471. ISSN: 10638210. DOI: `10.1109/TVLSI.2017.2700726`.

[87] Sumit K. Mandal, Ganapati Bhat, Chetan Arvind Patil, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y. Ogras. “Dynamic Resource Management of Heterogeneous Mobile Platforms via Imitation Learning”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.12 (2019), pp. 2842–2854. DOI: `10.1109/TVLSI.2019.2926106`.

[88] Anderson L Sartor, Anish Krishnakumar, Samet E Arda, Umit Y Ogras, and Radu Marculescu. “Hilite: Hierarchical and Lightweight Imitation Learning for Power Management of Embedded SoCs”. In: *IEEE Computer Architecture Letters (CAL)* 19.1 (2020), pp. 63–67.

[89] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning”. In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2011.

[90] William Lloyd Bircher, Madhavi Valluri, Jason Law, and Lizy K John. “Runtime Identification of Microprocessor Energy Saving Opportunities”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE. 2005, pp. 275–280.

[91] Sheriff Sadiqbatcha, Hengyang Zhao, Hussam Amrouch, Jörg Henkel, and Sheldon X-D Tan. “Hot Spot Identification and System Parameterized Thermal Modeling for Multi-Core Processors Through Infrared Thermal Imaging”. In: *Design, Automation & Test in Europe (DATE)*. IEEE. 2019, pp. 48–53.

[92] Ujjwal Gupta, Manoj Babu, Raid Ayoub, Michael Kishinevsky, Francesco Paterna, and Umit Y Ogras. “STAFF: Online Learning with Stabilized Adaptive Forgetting Factor and Feature Selection Algorithm”. In: *Design Automation Conference (DAC)*. IEEE. 2018.

[93] Matthew J Walker, Stephan Diestelhorst, Andreas Hansson, Anup K Das, Sheng Yang, Bashir M Al-Hashimi, and Geoff V Merrett. “Accurate and Stable Run-Time Power Modeling for Mobile and Embedded CPUs”. In: *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. (TCAD)* 36.1 (2016), pp. 106–119.

[94] Kaicheng Zhang, Akhil Guliani, Seda Ogrenci-Memik, Gokhan Memik, Kazutomo Yoshii, Rajesh Sankaran, and Pete Beckman. “Machine Learning-Based Temperature Prediction for Runtime Thermal Management Across System Components”. In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 29.2 (2017), pp. 405–419.

[95] Yang Ge, Qinru Qiu, and Qing Wu. “A Multi-agent Framework for Thermal Aware Task Migration in Many-Core Systems”. In: *Transactions on Very Large Scale Integration (VLSI) Systems* 20.10 (2011), pp. 1758–1771.

[96] Yeseong Kim, Pietro Mercati, Ankit More, Emily Shriver, and Tajana Rosing. “P^4: Phase-Based Power/Performance Prediction of Heterogeneous Systems via Neural Networks”. In: *International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 683–690.

[97] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Kenny C Gross. “Utilizing Predictors for Efficient Thermal Management in Multiprocessor SoCs”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 28.10 (2009), pp. 1503–1516.

[98] Javad Mohebbi Najm Abad and Ali Soleimani. “Novel Feature Selection Algorithm for Thermal Prediction Model”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.10 (2018), pp. 1831–1844.

[99] Hokchhay Tann, Soheil Hashemi, R Iris Bahar, and Sherief Reda. “Runtime Configurable Deep Neural Networks for Energy-Accuracy Trade-Off”. In: *Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE. 2016.

[100] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. “Slimmable Neural Networks”. In: *International Conference on Learning Representations (ICLR)* (2018).

[101] Maral Amir and Tony Givargis. “Priority Neuron: A Resource-Aware Neural Network for Cyber-Physical Systems”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37.11 (2018), pp. 2732–2742.

[102] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. “Federated Optimization in Heterogeneous Networks”. In: *Machine Learning and Systems (MLSys)*. Vol. 2. 2020.

[103] Junjie Shi, Jiang Bian, Jakob Richter, Kuan-Hsun Chen, Jörg Rahnenführer, Haoyi Xiong, and Jian-Jia Chen. “MODES: Model-Based Optimization on Distributed Embedded Systems”. In: *Machine Learning* 110.6 (2021), pp. 1527–1547.

[104] Daliang Li and Junpu Wang. “FedMD: Heterogenous Federated Learning via Model Distillation”. In: *Conference on Neural Information Processing Systems (NeurIPS) Workshop on Federated Learning for Data Privacy and Confidentiality*. 2019.

[105] Hongyan Chang, Virat Shejwalkar, Reza Shokri, and Amir Houmansadr. “Cronus: Robust and Heterogeneous Collaborative Learning with Black-Box Knowledge Transfer”. In: *arXiv preprint arXiv:1912.11279* (2019).

[106] Tao Lin, Lingjing Kong, Sebastian U Stich, and Martin Jaggi. “Ensemble Distillation for Robust Model Fusion in Federated Learning”. In: *Conference on Neural Information Processing Systems (NeurIPS)*. 2020.

[107] Samuel Horváth, Stefanos Laskaridis, Mario Almeida, Ilias Leontiadis, Stylianos I Venieris, and Nicholas D Lane. “FjORD: Fair and Accurate Federated Learning under heterogeneous targets with Ordered Dropout”. In: *Conference on Neural Information Processing Systems (NeurIPS)*. 2021.

[108] Rong Yu and Peichun Li. “Toward Resource-Efficient Federated Learning in Mobile Edge Computing”. In: *IEEE Network* 35.1 (2021), pp. 148–155.

[109] Enmao Diao, Jie Ding, and Vahid Tarokh. “HeteroFL: Computation and Communication Efficient Federated Learning for Heterogeneous Clients”. In: *International Conference on Learning Representations (ICLR)*. IEEE. 2021.

[110] Zirui Xu, Fuxun Yu, Jinjun Xiong, and Xiang Chen. “HELIOS: Heterogeneity-Aware Federated Learning with Dynamically Balanced Collaboration”. In: *Design Automation Conference (DAC)*. IEEE. 2021, pp. 997–1002.

[111] Ben Graham, Jeremy Reizenstein, and Leigh Robinson. “Efficient Batchwise Dropout Training using Submatrices”. In: *arXiv preprint arXiv:1502.02478* (2015).

[112] Sebastian Caldas, Jakub Konečny, H Brendan McMahan, and Ameet Talwalkar. “Expanding the Reach of Federated Learning by Reducing Client Resource Requirements”. In: *arXiv preprint arXiv:1812.07210* (2018).

[113] Anuj Pathania and Jörg Henkel. “HotSniper: Sniper-Based Toolchain for Many-Core Thermal Simulations in Open Systems”. In: *IEEE Embedded Systems Letters (ESL)* (2018). DOI: `10.1109/LES.2018.2866594`.

[114] Trevor E Carlson, Wim Heirman, and Lieven Eeckhout. “Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation”. In: *Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM. 2011. DOI: `10.1145/2063384.2063454`.

[115] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. “The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 10.1 (2013). DOI: `10.1145/2445572.2445577`.

[116] Gabriel Southern and Jose Renau. “Analysis of PARSEC Workload Scalability”. In: *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2016, pp. 133–142.

[117] Wim Heirman, Trevor E Carlson, Shuai Che, Kevin Skadron, and Lieven Eeckhout. “Using Cycle Stacks to Understand Scaling Bottlenecks in Multi-Threaded Workloads”. In: *Int. Symp. on Workload Characterization (IISWC)*. IEEE. 2011, pp. 38–49. DOI: `10.1109/IISWC.2011.6114195`.

[118] Santiago Pagani, Jian-Jia Chen, Muhammad Shafique, and Jörg Henkel. “MatEx: Efficient Transient and Peak Temperature Computation for Compact Thermal Models”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 1515–1520.

[119] Ankireddy Nalamalpu, Nasser Kurd, Anant Deval, Chris Mozak, Jonathan Douglas, Ashish Khanna, Fabrice Paillet, Gerhard Schrom, and Boyd Phelps. “Broadwell: A family of IA 14nm processors”. In: *Symposium on VLSI Circuits (VLSI Circuits)*. IEEE. 2015.

[120] Carl Ramey. “Tile-Gx100 Manycore Processor: Acceleration Interfaces and Architecture”. In: *Hot Chips Symposium (HCS)*. IEEE. 2011. DOI: `10.1109/HOTCHIPS.2011.7477491`.

[121] Linaro 96Boards. *HiKey970*. https://96boards.org/product/hikey970/.

[122] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. “Communication-Efficient Learning of Deep Networks from Decentralized Data”. In: *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 2017.

[123] Yuanming Shi, Kai Yang, Tao Jiang, Jun Zhang, and Khaled B Letaief. “Communication-Efficient Edge AI: Algorithms and Systems”. In: *IEEE Communications Surveys & Tutorials* 22.4 (2020).

[124] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. “A Survey of Accelerator Architectures for Deep Neural Networks”. In: *Engineering* 6.3 (2020), pp. 264–274.

[125] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet Classification with Deep Convolutional Neural Networks”. In: *Neural Information Processing Systems (NIPS)*. 2012, pp. 1097–1105.

[126] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. “Knights Landing: Second-Generation Intel Xeon Phi Product”. In: *IEEE MICRO* 36.2 (2016), pp. 34–46.

[127] Jack Dongarra, Stanimire Tomov, Piotr Luszczek, Jakub Kurzak, Mark Gates, Ichitaro Yamazaki, Hartwig Anzt, Azzam Haidar, and Ahmad Abdelfattah. “With Extreme Computing, the Rules have Changed”. In: *Computing in Science & Engineering* 19.3 (2017), pp. 52–62.

[128] Santiago Pagani, Heba Khdr, Jian-Jia Chen, Muhammad Shafique, Minming Li, and Jörg Henkel. “Thermal Safe Power (TSP): Efficient Power Budgeting for Heterogeneous Manycore Systems in Dark Silicon”. In: *IEEE Transactions on Computers (TC)* 66.1 (2017), pp. 147–162. DOI: 10.1109/TC.2016.2564969.

[129] Sai Manoj PD, Hao Yu, and Kanwen Wang. “3D Many-Core Microprocessor Power Management by Space-Time Multiplexing Based Demand-Supply Matching”. In: *IEEE Transactions on Computers (TC)* 64.11 (2015), pp. 3022–3036.

[130] Stijn Eyerman, Kristof Du Bois, and Lieven Eeckhout. “Speedup Stacks: Identifying Scaling Bottlenecks in Multi-Threaded Applications”. In: *International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE. 2012, pp. 145–155.

[131] Alaa R Alameldeen and David A Wood. “IPC Considered Harmful for Multiprocessor Workloads”. In: *IEEE Micro* 26.4 (2006), pp. 8–17.

[132] Henry Hoffmann, Jonathan Eastep, Marco D Santambrogio, Jason E Miller, and Anant Agarwal. “Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments”. In: *International Conference on Autonomic Computing (ICAC)*. 2010, pp. 79–88.

[133] Diederik P Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[134] Rem Gensh, Ali Aalsaud, Ashur Rafiev, Fei Xia, Alexei Iliasov, Alexander Romanovsky, and Alex Yakovlev. “Experiments with Odroid-XU3 Board”. In: *School of Computing Science Technical Report Series* (2015).

[135] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[136] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *Computer Vision and Pattern Recognition (CVPR).* IEEE. 2018, pp. 2704–2713.

[137] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. “An Energy Efficiency Feature Survey of the Intel Haswell Processor”. In: *Int. Parallel and Distributed Processing Symp. Workshop (IPDPSW).* IEEE. 2015, pp. 896–904.

[138] Sebastian Ruder. “An Overview of Multi-Task Learning in Deep Neural Networks”. In: *arXiv preprint arXiv:1706.05098* (2017).

[139] Begum Egilmez, Gokhan Memik, Seda Ogrenci-Memik, and Oguz Ergin. “User-Specific Skin Temperature-Aware DVFS for Smartphones”. In: *Design, Automation & Test in Europe Conf. & Exhibition (DATE).* 2015.

[140] Anuj Pathania, Heba Khdr, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. “QoS-aware Stochastic Power Management for Many-Cores”. In: *Design Automation Conf. (DAC).* 2018.

[141] Tomofumi Yuki and Louis-Noël Pouchet. *Polybench 4.0.* 2015.

[142] Quintin Fettes, Mark Clark, Razvan Bunescu, Avinash Karanth, and Ahmed Louri. “Dynamic Voltage and Frequency Scaling in NoCs with Supervised and Reinforcement Learning Techniques”. In: *IEEE Transactions on Computers (TC)* 68.3 (2019), pp. 375–389.

[143] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. “AI Benchmark: Running Deep Neural Networks on Android Smartphones”. In: *European Conference on Computer Vision (ECCV).* 2018.

[144] Rahul Jain, Preeti Ranjan Panda, and Sreenivas Subramoney. “Cooperative Multi-Agent Reinforcement Learning-based Co-Optimization of Cores, Caches, and On-Chip Network”. In: *ACM Tran. on Architecture and Code Optimization (TACO)* 14.4 (2017).

[145] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, H. Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. "Towards Federated Learning at Scale: System Design". In: *SysML Conference*. 2019.

[146] Yiqiang Chen, Xin Qin, Jindong Wang, Chaohui Yu, and Wen Gao. "FedHealth: A Federated Transfer Learning Framework for Wearable Healthcare". In: *IEEE Intelligent Systems* 35.4 (2020), pp. 83–93.

[147] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. "ImageNet Training in Minutes". In: *International Conference on Parallel Processing (ICPP)*. 2018.

[148] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. "On-device Machine Learning: An Algorithms and Learning Theory Perspective". In: *arXiv preprint arXiv:1911.00623* (2019).

[149] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. "Real-Time Video Analytics: The Killer App for Edge Computing". In: *Computer* 50.10 (2017), pp. 58–67.

[150] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. "Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1738–1762.

[151] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. "Live Video Analytics at Scale With Approximation and Delay-Tolerance". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2017, pp. 377–392.

[152] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. "Applied Federated Learning: Improving Google Keyboard Query Suggestions". In: *arXiv preprint arXiv:1812.02903* (Dec. 7, 2018).

[153] Chia-Heng Tu, Hui-Hsin Hsu, Jen-Hao Chen, Chun-Han Chen, and Shih-Hao Hung. "Performance and Power Profiling for Emulated Android Systems". In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 19.2 (2014).

[154] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. "Federated Learning: Challenges, Methods, and Future Directions". In: *IEEE Signal Processing Magazine* 37.3 (2020), pp. 50–60.

[155] Ahmed Imteaj, Urmish Thakker, Shiqiang Wang, Jian Li, and M Hadi Amini. "Federated Learning for Resource-Constrained IoT Devices: Panoramas and State-of-the-art". In: *arXiv preprint arXiv:2002.10610* (2020).

[156] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Neural Information Processing Systems (NeurIPS).* 2019, pp. 8024–8035.

[157] Zhuoran Song, Ru Wang, Dongyu Ru, Zhenghao Peng, Hongru Huang, Hai Zhao, Xiaoyao Liang, and Li Jiang. "Approximate Random Dropout for DNN Training Acceleration in GPGPU". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE. 2019.

[158] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. "Densely Connected Convolutional Networks". In: *Conference on Computer Vision and Pattern Recognition (CVPR).* 2017.

[159] Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, and Ilya Sutskever. *AI and Compute.* 2018. URL: `https://openai.com/blog/ai-and-compute/`.

[160] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, and Yingyan Lin. "HW-NAS-Bench: Hardware-Aware Neural Architecture Search Benchmark". In: *International Conference on Learning Representations (ICLR)* (2021).

[161] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. "Deep Networks with Stochastic Depth". In: *European Conference on Computer Vision (ECCV).* Springer. 2016.

[162] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. "Accelerating Neural Architecture Search using Performance Prediction". In: *arXiv preprint arXiv:1705.10823* (2017).

[163] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural Architecture Search: A Survey". In: *Journal of Machine Learning Research (JMLR)* 20 (2019).

[164] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II". In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197.

[165] Francesco Biscani and Dario Izzo. "A Parallel Global Multiobjective Framework for Optimization: pagmo". In: *Journal of Open Source Software* 5.53 (2020). DOI: `10.21105/joss.02338`. URL: `https://doi.org/10.21105/joss.02338`.

[166] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. "EMNIST: Extending MNIST to Handwritten Letters". In: *International Joint Conference on Neural Networks (IJCNN).* IEEE. 2017, pp. 2921–2926.

[167] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečný, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. "LEAF: A Benchmark for Federated Settings". In: *Conference on Neural Information Processing Systems (NeurIPS).* 2019.

[168] Alex Krizhevsky and Geoffrey Hinton. *Learning Multiple Layers of Features from Tiny Images.* 2009.

[169] Lisa Torrey and Jude Shavlik. “Transfer Learning”. In: *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques.* IGI Global, 2010, pp. 242–264.

[170] Jakob Richter, Junjie Shi, Jian-Jia Chen, Jörg Rahnenführer, and Michel Lang. “Model-Based Optimization with Concept Drifts”. In: *Genetic and Evolutionary Computation Conference (GECCO).* 2020, pp. 877–885.