

SMAUG: System-Level Modelling and Optimized Use of Disruptive Memory Technologies

Birte Friesel^[0000–0002–0688–9440], Marcel Lütke Dreimann^[0009–0007–2426–4798],
and Olaf Spinczyk^[0000–0001–9469–2367]

Osnabrück University, Wachsbleiche 27, 49090 Osnabrück, Germany
{birte.friesel,marcel.luetkedreimann,olaf}@uos.de
<https://ess.cs.uos.de>

Abstract. Efficient execution of software on modern, heterogeneous hardware is becoming more and more challenging. In addition to variability within software product lines (i.e., compile- and runtime configuration options), there is also a growing amount of variability in how this software is executed. Modern servers provide a plethora of different compute and memory components, and the placement of individual compute kernels and data objects on those can have drastic influence on application latency and throughput. Yet, there is no simple rule for optimal placement of application tasks and data objects – it depends on current system utilization, application characteristics, and characteristics of the underlying hardware. Simply treating those aspects as additional sources of variability within application-specific performance models does not scale. The SMAUG (“System-Level Modelling and Optimized Use of Disruptive Memory Technologies”) project aims to tackle this challenge by providing a holistic view on the entire system, and disentangling configuration-dependent software behaviour from placement-dependent software and hardware performance. So far, it has refuted several common placement heuristics, provided detailed performance models for high-bandwidth memory and near-memory computing devices, and utilized those to improve application performance. At present, we are working to combine application-independent performance models with hardware-independent software behaviour models for accurate, efficient, and scalable performance modelling of software on modern servers.

Keywords: Performance Models · Scheduling · Data Placement.

1 Introduction

Modelling runtime performance attributes of software product lines (and configurable software in general) is far from trivial, as it depends on much more than just static product line configuration. For instance, it is also affected by the workload (e.g., database query sequence), which must be either representative or treated as an explicit source of variability [4,19,21]. Additionally, performance depends not just on the underlying hardware, but also on the placement of individual application tasks and data objects on hardware components [10,12]. This is typically not modelled as an explicit source of variability [14,23,24,35].

Two different communities deal with aspects of this challenge: the software product line engineering community considers high-level static and run-time variability, and does not take hardware or placement variations into account. The performance engineering community, on the other hand, aims to optimize the placement of tasks and data objects without a variability or performance modelling approach, typically resorting to hand-crafted heuristics instead.

While those heuristics are often based on benchmarks and architectural considerations, they may exhibit bad performance due to bugs or pathological corner cases [28]. For instance, some applications benefit from counter-intuitive setups such as placing input data on *remote* rather than local memory, or using conventional (*low-bandwidth*) DDR memory rather than high-bandwidth memory (HBM) [12]. Heuristics would prefer local input data and HBM.

This is where the SMAUG (“System-Level Modelling and Optimized Use of Disruptive Memory Technologies”) project comes into play. Its key goals are efficient methods for performance modelling of arbitrary server hardware and of configurable applications on top of that, and then using those within model-guided task and data placement algorithms, thus improving application performance. In doing so, it aims to bridge the gap between the product line engineering and performance engineering communities, and combine approaches from both.

SMAUG is a project within the DFG’s (Deutsche Forschungsgemeinschaft / German Research Council) priority programme 2377 “Disruptive Memory Technologies”¹. It has been funded from 2023 to 2026, and has recently been extended for a second (final) three-year funding period until 2029.

In the first funding period, it focused on *building* performance models both on the platform (i.e., server components and topology) and on the application level. Now, in the second funding period, the main goal is to *use* those models for efficient, model-guided resource management strategies. As such, SMAUG is a prime example of applying variability and performance modelling techniques in a real-world context, namely, for improving system performance.

The next section describes the challenges on the way towards model-guided optimized use of memory technologies from a variability and performance modelling perspective. Section 3 follows up with details on the goals of the two funding periods, including how those address the aforementioned challenges. We present insights and impact in Section 4, and conclude in Section 5.

2 Challenges

From a scheduling and placement perspective, each application consists of a set of compute kernels (each of which is represented by a group of tasks) and corresponding data objects [33]. The goal is to place each task and data object in a way that reduces total application latency (*makespan*), energy, cost, or similar [22,25,36,37]. As this is an NP-complete problem even in simplified scenarios, *optimal* solutions are generally out of the question [34].

¹ spp2377.uos.de

From a variability modelling perspective, we could define the configuration and placement variability of each task, perform benchmarks, and use the resulting performance model for task and data placement decisions. After all, we already know how to build performance models for software product lines, and can apply most of those algorithms to *any* kind of configurable software system [32]. However, building and using those models would be anything but practical.

First, applications contain anything from dozens to thousands of compute kernels and data objects, each of which would need to be benchmarked and modelled separately. Second, such models would have to be adjusted whenever the application or the underlying system (hardware, drivers, or similar) changes.

For instance, Fig. 1 shows a feature model excerpt for a single kernel in such an application. It can be run with a variable number of threads, and its input and output data can be placed on different types of memory. Memory is either on the same node as the kernel itself (Local), on the same socket (SameSocket), or on a different one (CrossSocket). Thus, *each* kernel increases the size of the feature model by a factor of more than 3^4 . With dozens of kernels, and conventional static and runtime variability on top, this leads to complex variability models and infeasibly large configuration spaces for performance modelling – and we are not even considering shared data between kernels (i.e., feature dependencies) yet.

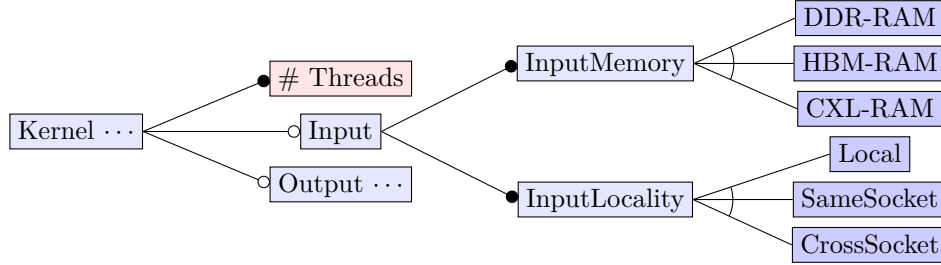


Fig. 1. Feature model excerpt for the runtime variability of a single compute kernel. Output sub-tree (same as Input) and cross-tree constraints left out for brevity.

At the same time, all variability and performance models used in this context should not just be accurate, but also *interpretable*: they should allow engineers to understand how their hardware behaves and why a specific task or data object should be placed on a specific compute or memory node. Furthermore, if the models are to be used at runtime, they must also be *fast*: deciding where to place a task or data object should not take more than a few microseconds, and especially not longer than the latency improvement that can be gained from such a model-guided placement decision. Otherwise, it would be a net loss.

Altogether, while existing variability and performance modelling techniques offer many useful building blocks for this endeavour, they are not capable of solving it on their own. We now outline how the SMAUG project intends to achieve model-guided optimized use of complex compute and memory hierarchies.

3 Goals

The key idea is to *disentangle* runtime application behaviour from raw system performance, and use operating system or hardware-level *API functions* as interface between the two. Application behaviour (e.g., function calls and function call arguments) depends on configuration and runtime variability, but is independent of the underlying hardware. Meanwhile, raw system performance (i.e., the latency and throughput of hardware / system API functions) is independent of which application (or kernel) is using these functions.

Hence, SMAUG’s variability and performance modelling approach consists of two layers: a *platform model* that only describes the raw hardware capabilities of the system, and *application models* that describe application-specific runtime behaviour and interaction with hardware components. The platform model is further subdivided into *component models* that describe variability and performance attributes of individual hardware components, and a *system topology model* that takes care of links between those. See Fig. 2 for an overview.

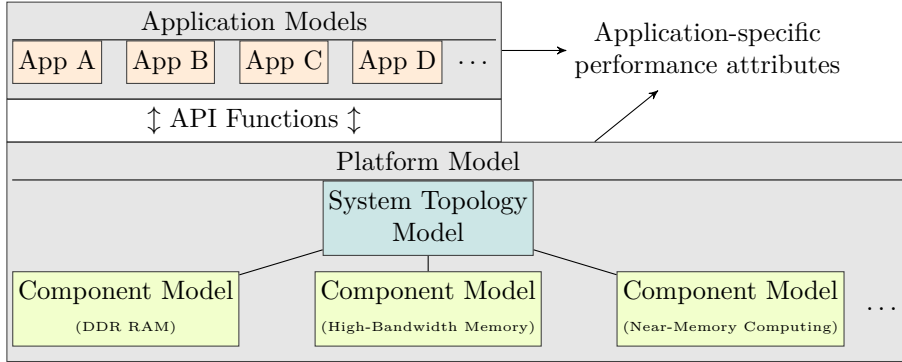


Fig. 2. Variability models used within the SMAUG project: component models exclusively deal with hardware performance, whereas application models exclusively deal with runtime behaviour. They are subject to different sources of variability.

As one of the ideas behind SMAUG is to apply methods from variability and performance modelling research to the systems domain in order to improve resource management decisions within operating systems, we devised our approach so that is compatible with prior work in compute and data placement research, including associated tooling [29].

Ultimately, we want to use these models to train machine learning-based placement predictors. Given a feature vector $\mathbf{x}$ that provides information about application configuration and runtime behaviour (e.g., data object size and type of memory access pattern) as well as system status (e.g., current utilization of each available compute and memory unit), their output (prediction) should indicate where a given task should be scheduled or where a given data object

should be placed. However, we must first devise the underlying models, including fully-automated benchmarking and learning algorithms.

3.1 Platform Model

Component models rely on a conventional variability and performance modelling approach. We build a variability model for each component and API function, combine it with benchmark applications and/or code generation for data acquisition, and then train latency and throughput prediction models. In order to maintain interpretability, we build upon *regression model trees*, which have already been used for both software and hardware product line analyses [14,15].

For instance, the `memcpy` function can transfer data between different memory nodes, i.e., different system components such as DDR RAM and HBM. Its variability (feature vector $\mathbf{x}$) consists of data size, number of concurrent data transfer threads, source and destination memory type (DDR RAM, HBM, ...), and the distance between transfer threads and input/output memory node (i.e., locality / system topology). For execution of tasks on GPU or near-memory computing devices, we also consider functions for device management: allocation, de-allocation, and application upload.

We also model the raw read and write performance for each compute and memory placement combination. Here, aspects such as the *access pattern* are an additional source of variability: linear and certain strided read patterns benefit from caching and prefetching, whereas random accesses expose the raw latency of the underlying memory technology and interconnect.

The topology model defines links between different compute and memory components [10]. For instance, CPU tasks can directly access data in HBM, DDR, or CXL RAM, but not data on a GPU; tasks scheduled on a GPU face the inverse challenge. Placement algorithms must account for this, initiate a data transfer to a directly reachable memory if necessary, and consider the system and runtime configuration-dependent latency penalty of doing so.

3.2 Application Models

Application models must predict *how* applications will interact with the available hardware components. Here, we assign each *application* with a feature vector $\mathbf{x}$ that we use to predict its runtime behaviour, and then predict the latency of the resulting API function calls and primitive memory operations.

An application-specific feature vector includes compile- and runtime application configuration (i.e., conventional product line features). However, it also covers workload-specific features, such as the size and type of input data, and *how* the application interacts with it – e.g., whether it utilizes a linear, strided, or random access pattern. Still, engineers should not have to provide manual access pattern annotations and the likes. Instead, we intend to devise these automatically, using a combination of static analysis and application profiling [30].

In order to predict feature-dependent application behaviour, including different paths through callsites, we can build upon *labelled transition systems* and

their *feature guards* that link features with runtime behaviour [3,4]. In addition to those, we must annotate each function call and hardware access primitive with a fully-featured performance model for latency prediction. We intend to achieve this by linking each of them with the corresponding component model.

3.3 Runtime Usage

SMAUG models are meant to be compatible with both *offline* and *online* compute and data placement applications. In the former case, algorithms can combine application and platform models to predict the latency of application components for any valid compute/data placement, and choose a suitable one. For the latter, this is not feasible: runtime placement decisions must be made within a few microseconds, and selecting a suitable placement by predicting the latency of all placements and choosing the best one would be far too slow.

So far, researchers have typically used hand-crafted heuristics [2,5,26,27]. By contrast, we envision a paradigm shift towards fully *model-guided* placement decisions. Specifically, we want to use our models to train predictors that directly decide on the placement of individual tasks and data objects. So, given an input vector $\mathbf{x}$ that describes both system status / capabilities and application configuration, workload, and characteristics, a predictor should indicate where each task and data object should be placed. We intend to utilize platform and application model-based simulations in order to provide training data for the placement predictor (see Fig. 3), thus achieving manageable learning times.

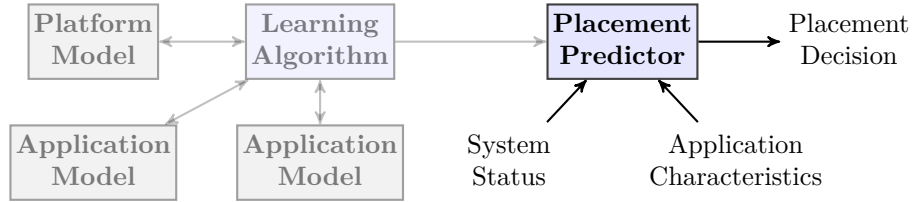


Fig. 3. SMAUG approach for efficient runtime usage of performance models by embedding them into a system management / task and data placement predictor.

While similar approaches have already been considered in the past [7], research thus far has focused on *which* task to schedule next, and not *where* to place it. We now move on to our variability and performance modelling insights.

4 Insights

At the time of writing, we have already completed the majority of work on platform models, and are currently refining the application model concept. Runtime usage of these models is still in an early stage, and thus not covered here.

4.1 Platform Models

So far, we have built, analyzed, and applied component models for DDR RAM, HBM, RAM with near-memory computing (NMC) capabilities, and CXL memory extensions. We have also linked those by means of a topology model, and used them within a scheduling simulator for heterogeneous hardware [29].

Near-Memory Computing Here, we examined UPMEM PIM, which was the only (briefly) commercially available implementation of near-memory computing at the time of writing. It builds upon 8 GiB DDR4 memory modules, each of which consists of two 4 GiB ranks and provides a total of 128 DRAM Processing Units (DPUs). Each DPU has exclusive access to a 64 MiB segment of RAM; there is no shared memory or inter-DPU communication.

This setup can speed up application kernels by up to three orders of magnitude thanks to its massive parallelism [20]: a fully-equipped system with 160 GiB of NMC RAM provides 2,560 DPUs, all of which can work in parallel. However, these findings neglect a significant source of variability (and, thus, latency) by assuming that all input data is already present in NMC RAM. In practice, data in NMC RAM cannot be seamlessly accessed from main memory, and instead has to be moved (and converted) between DDR RAM and NMC RAM [11].

So, in order to decide whether NMC execution is beneficial, variability models, performance models, and placement algorithms must also consider whether data has already been transferred to NMC RAM, how many consecutive operations will be performed on a certain data set, its size, and the number of available NMC ranks (i.e., DPUs). This is further complicated by the fact that NMC resource allocation itself is also not trivial: simply allocating all available NMC ranks may be slower than only using a limited amount.

COUNT Query – NMC vs. 4 CPU Threads																																							
# Rows	2 ³²	-	-	-	-	-	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2			
	2 ³¹	-	-	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2			
	2 ³⁰	-	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2			
	2 ²⁹	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2			
	2 ²⁸	7	3	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3			
	2 ²⁷	8	3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5			
	2 ²⁶	9	4	4	4	4	4	4	4	4	5	5	5	6	7	7	8	8	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9			
	2 ²⁵	13	5	5	6	6	6	7	7	8	9	10	11	12	13	14	15	17	18	19	20	21	22	24	25	26													
		1	2	3	4	5	6	7	8	9	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40													
		# Ranks																																					

Fig. 4. The minimum amount of consecutive SQL COUNT operations required for an NMC-offloaded query to be faster than quad-core CPU execution, depending on number of allocated NMC ranks and column size.

For instance, we found that offloading an SQL COUNT query to NMC memory only pays off when at least two consecutive queries operate on the same data set [13]. As Fig. 4 shows, the optimal amount of NMC ranks also depends on column size, making proper performance models all the more important.

Note that we started our NMC analysis with a conventional variability and performance modelling approach that did not yet disentangle application behaviour from hardware performance. While this worked [11,12], it did not scale well to more complex applications, and the resulting models were not interpretable. By combining behaviour models and platform models, we were able to understand bottlenecks in NMC applications, and identify how updates of the UPMEM SDK changed those [16,18].

NUMA-Aware Data Placement The Linux kernel assumes that, ideally, a thread should only access local data objects. As such, for a thread that is scheduled on NUMA node n , it attempts to allocate all data on NUMA node n , and may even migrate threads or data objects from other NUMA nodes. With this in mind, performance models for runtime attributes of software systems typically do not consider NUMA as a source of variability: the kernel will do its best to achieve optimal latency and throughput anyway.

Intuitively, this makes sense: as our models confirm, in isolation, accessing remote NUMA nodes comes with both higher latency and lower bandwidth. However, this does not hold for all access patterns. For instance, the performance of DDR RAM suffers whenever applications frequently alternate between read and write commands, as is the case in memory-bound workloads such as multi-threaded vector addition. We found that this operation is, in fact, faster when its input data is placed on a *remote* NUMA node and only output data is stored locally [12]. This is relevant in practice, as the Linux kernel provides APIs that allow applications to override its placement heuristics.

This finding underlines the importance of considering the underlying hardware topology when building variability and performance models for software product lines: compute and data placement is a source of variability, and should be used as input to performance prediction models. It also shows that established heuristics are not ideal in general.

High-Bandwidth Memory Our next target was high-bandwidth memory (HBM), as provided by, e.g., certain Intel server CPUs and certain FPGAs.

Here, we found that, while it does indeed provide much higher bandwidth than conventional DDR RAM, this comes at the cost of significantly higher random access latency. Depending on application-specific memory access patterns, this may *slow down* applications when data is placed on high-bandwidth memory. One such instance was a common matrix-vector multiplication kernel [12].

Again, this underlines the importance of proper variability modelling, and helped refute an established heuristic. In this case, the prevalent assumption was that HBM is always a better option than DDR RAM, and should be used, e.g., as a cache in front of it [5,6].

4.2 Application Models

The goal of our application models is to predict configuration- and workload-dependent runtime behaviour. To this end, we have taken inspiration from state machine-based behaviour models that are already being used in the variability modelling domain: labelled transition systems, weighted featured transition systems, and featured weighted automata [3,4,8,31]. These annotate transitions with configuration-dependent feature guards and constant annotations for, e.g., latency prediction. We extended them with support for numeric product line features, arbitrary performance models, and the ability to deal with finite loops.

We first examined a straightforward approach, where a single, application- and workload-specific feature vector $\mathbf{x}$ is used both within feature guards (i.e., to determine runtime behaviour) and as input to transition-specific performance models (i.e., to determine application latency) [17]. While this worked well with manually-designed behaviour models, it lacked a learning algorithm.

Following these findings and feedback from the product line engineering community, we split the variability model into two parts. Each *application* is associated with a feature vector $\mathbf{x}$ that we use to predict its runtime behaviour (i.e., API calls), and each *callsite* is associated with a separate feature vector $\mathbf{x}'$ that serves as input to its performance model. Now, the behaviour model incorporates not just configuration-dependent feature guards, but also models that predict the feature vectors $\mathbf{x}'$ of each callsite from $\mathbf{x}$ [18].

At this point, we had successfully disentangled application behaviour from hardware performance, and were able to use a single behaviour model and application-independent platform models to understand and accurately predict workload-dependent application latency even after hardware changes [18]. However, *learning* the behaviour model proved challenging. While FL* offers an adaptation of L* for automated learning of featured automata, we were unable to apply it in this context, partially due to its limitation to boolean features [1,9].

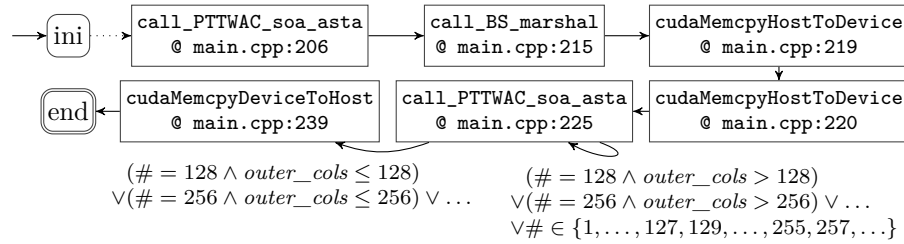


Fig. 5. Excerpt from a featured control flow graph (FCFG) for GPU-accelerated matrix transposition. Feature guards determine how often `call_PTTWAC_soa_asta` is called.

Ultimately, we decided to take inspiration from control flow graphs, and move function calls (and corresponding performance models) from transitions into states. Fig. 5 shows an example for such a *featured control flow graph* (FCFG).

In addition to the feature vector $\mathbf{x}$, feature guards may reference the special “#” variable, which counts how often each transition has already been passed. This allows us to deal with loops and predict their iteration counts.

Thanks to this change, we could devise a trace-based learning algorithm, which we have already successfully applied to performance modelling for CPU and NMC execution of database workloads [16], and are now extending to further domains. The algorithm learns all parts of the model from these traces: state machine structure, transition guards (including iteration counts), models for prediction of API function arguments, and performance models to transform those into function call latencies. We are currently revising and further improving the learning algorithm, and seeking feedback on it (and on FCFGs in general) from the software product line engineering and performance modelling communities.

5 Conclusion

We have presented SMAUG (“System-Level Modelling and Optimized Use of Disruptive Memory Technologies”), a project within DFG’s (Deutsche Forschungsgemeinschaft / German Research Council) priority programme 2377 “Disruptive Memory Technologies”. It has been funded from 2023 on, and will run until 2029.

SMAUG addresses the challenge that runtime performance attributes of software product lines (and software in general) are subject to far more variability than just compile-time configuration options and runtime command line flags, especially on servers with multiple NUMA regions and heterogeneous compute and memory technologies. Specifically, hardware topology and the placement of individual application tasks and data objects have a significant influence on the throughput and latency of software systems, and should be treated as explicit sources of variability if accurate performance prediction is desired. While established heuristics often help, they can also be detrimental.

The SMAUG project splits runtime performance models into configuration-dependent, hardware-independent behaviour models and placement-dependent, software-independent platform models. The former focus on software, the latter on hardware. In doing so, it disentangles two sources of variability, and shrinks the configuration space of both models. Its behaviour models build upon featured transition systems, and have already explored several ways of extending their feature guards and performance annotations, dealing with loops, and learning them from application traces.

Although the project itself is rooted within the systems community, its goals overlap with two research domains, and it aims to combine approaches from both: product line engineering and variability modelling on one hand, and performance modelling on the other. SMAUG is currently in the second of its two three-year funding periods, and we expect many more interesting results to come from this cross-disciplinary approach towards improving application performance on servers with heterogeneous, disruptive memory technologies.

Acknowledgments. This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 502565817.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. *Information and Computation* **75**(2), 87–106 (1987). [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
2. Benoit, A., Perarnau, S., Pottier, L., Robert, Y.: A performance model to execute workflows on high-bandwidth-memory architectures. In: *Proceedings of the 47th International Conference on Parallel Processing, ICPP '18, Association for Computing Machinery* (2018). <https://doi.org/10.1145/3225058.3225110>
3. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite Runs in Weighted Timed Automata with Energy Constraints. In: *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems*. pp. 33–47. *FORMATS'08, Springer Berlin, Heidelberg* (2008). https://doi.org/10.1007/978-3-540-85778-5_4
4. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F.: Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* **39**(8), 1069–1089 (2013). <https://doi.org/10.1109/TSE.2012.86>
5. Das, R., Agrawal, K., Bender, M.A., Berry, J., Moseley, B., Phillips, C.A.: How to Manage High-Bandwidth Memory Automatically. In: *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. p. 187–199. *SPAA '20, Association for Computing Machinery, New York, NY, USA* (2020). <https://doi.org/10.1145/3350755.3400233>
6. DeLayo, D., Zhang, K., Agrawal, K., Bender, M.A., Berry, J.W., Das, R., Moseley, B., Phillips, C.A.: Automatic HBM Management: Models and Algorithms. In: *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*. p. 147–159. *SPAA '22, Association for Computing Machinery, New York, NY, USA* (2022). <https://doi.org/10.1145/3490148.3538570>
7. Dias, S., Naik, S., Sreepraneeth, K., Rmaan, S., Namratha, M.: A machine learning approach for improving process scheduling: A survey. *International Journal of Computer Trends and Technology* **43**(1) (2017). <https://doi.org/10.14445/22312803/IJCTT-V43P101>
8. Fahrenberg, U., Legay, A.: Featured Weighted Automata. In: *Proceedings of the 5th International FME Workshop on Formal Methods in Software Engineering*. pp. 51–57. *FormaliSE '17, IEEE* (2017). <https://doi.org/10.1109/FormaliSE.2017.2>
9. Fortz, S.: Variability-aware behavioural learning. In: *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B*. p. 11–15. *SPLC '23, Association for Computing Machinery, New York, NY, USA* (2023). <https://doi.org/10.1145/3579028.3609007>
10. Friesel, B., Lütke Dreimann, M., Porrmann, M., Spinczyk, O.: Towards Task-Based Scheduling on Truly Heterogeneous Systems. In: *Tagungsband des FG-BS Frühjahrstreffens 2026. Gesellschaft für Informatik e.V., Bonn* (2 2026). <https://doi.org/10.18420/fgbs2026f-03>
11. Friesel, B., Lütke Dreimann, M., Spinczyk, O.: A Full-System Perspective on UP-MEM Performance. In: *Proceedings of the 1st Workshop on Disruptive Memory Systems*. pp. 1–7. *DIMES '23, Association for Computing Machinery, New York, NY, USA* (10 2023). <https://doi.org/10.1145/3609308.3625266>
12. Friesel, B., Lütke Dreimann, M., Spinczyk, O.: Performance Models for Task-based Scheduling with Disruptive Memory Technologies. In: *Proceedings of the 2nd Workshop on Disruptive Memory Systems*. pp. 1–8. *DIMES '24, Association*

- for Computing Machinery, Austin, TX, USA (Nov 2024). <https://doi.org/10.1145/3698783.3699376>
13. Friesel, B., Lütke Dreimann, M., Spinczyk, O.: Lightning Talk: Feasibility Analysis of Semi-Permanent Database Offloading to UPMEM Near-Memory Computing Modules. In: Datenbanksysteme für Business, Technologie und Web – Workshopband, pp. 355–366. BTW '25, Gesellschaft für Informatik, Bonn, Germany (3 2025). <https://doi.org/10.18420/BTW2025-140>
 14. Friesel, B., Spinczyk, O.: Black-Box Models for Non-Functional Properties of AI Software Systems. In: Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI. pp. 170–180. CAIN '22, Association for Computing Machinery (5 2022). <https://doi.org/10.1145/3522664.3528602>
 15. Friesel, B., Spinczyk, O.: Regression Model Trees: Compact Energy Models for Complex IoT Devices. In: Proceedings of the Workshop on Benchmarking Cyber-Physical Systems and Internet of Things. pp. 1–6. CPS-IoTBench '22, IEEE (5 2022). <https://doi.org/10.1109/CPS-IoTBench56135.2022.00007>
 16. Friesel, B., Spinczyk, O.: Overhead Prediction for PIM-Enabled Applications with Performance-Aware Behaviour Models. In: Proceedings of the 1st Cross-disciplinary Conference on Memory-Centric Computing. pp. 1–8. CCMCC '25, IEEE (10 2025). <https://doi.org/10.1109/CCMCC67628.2025.11380800>
 17. Friesel, B., Spinczyk, O.: Performance-Aware Behaviour Models for Feature-Dependent Runtime Attributes in Product Lines. In: Proceedings of the 19th International Working Conference on Variability Modelling of Software-Intensive Systems. pp. 131–135. VaMoS '25, Association for Computing Machinery, New York, NY, USA (2 2025). <https://doi.org/10.1145/3715340.3715435>
 18. Friesel, B., Spinczyk, O.: Understanding Product Line Runtime Performance with Behaviour Models and Regression Model Trees. In: Proceedings of the 29th ACM International Systems and Software Product Line Conference - Volume A. pp. 142–148. SPLC-A '25, Association for Computing Machinery, New York, NY, USA (9 2025). <https://doi.org/10.1145/3744915.3748472>
 19. Göttmann, H., Luthmann, L., Lochau, M., Schürr, A.: Real-time-aware reconfiguration decisions for dynamic software product lines. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line - Volume A. SPLC '20, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3382025.3414945>
 20. Gómez-Luna, J., Hajj, I.E., Fernandez, I., Giannoula, C., Oliveira, G.F., Mutlu, O.: Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* **10**, 52565–52608 (2022). <https://doi.org/10.1109/ACCESS.2022.3174101>
 21. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic Software Product Lines. *Computer* **41**(4), 93–95 (2008). <https://doi.org/10.1109/MC.2008.123>
 22. Hosseinzadeh, M., Azhir, E., Lansky, J., Mildeova, S., Ahmed, O.H., Malik, M.H., Khan, F.: Task Scheduling Mechanisms for Fog Computing: A Systematic Survey. *IEEE Access* **11**, 50994–51017 (2023). <https://doi.org/10.1109/ACCESS.2023.3277826>
 23. Jamshidi, P., Velez, M., Kästner, C., Siegmund, N.: Learning to Sample: Exploiting Similarities across Environments to Learn Performance Models for Configurable Systems. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 71–82. ESEC/FSE '18, Association for Computing Machinery, New York, NY, USA (11 2018). <https://doi.org/10.1145/3236024.3236074>

24. Jamshidi, P., Velez, M., Kästner, C., Siegmund, N., Kawthekar, P.: Transfer Learning for Improving Model Predictions in Highly Configurable Software. In: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 31–41. SEAMS '17 (2017). <https://doi.org/10.1109/SEAMS.2017.11>
25. Jayanetti, A., Buyya, R.: J-OPT: A Joint Host and Network Optimization Algorithm for Energy-Efficient Workflow Scheduling in Cloud Data Centers. In: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing. p. 199–208. UCC'19, Association for Computing Machinery (2019). <https://doi.org/10.1145/3344341.3368822>
26. Laghari, M., Unat, D.: Object placement for high bandwidth memory augmented with high capacity memory. In: Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). pp. 129–136. SBAC-PAD '17 (2017). <https://doi.org/10.1109/SBAC-PAD.2017.24>
27. Lasch, R., Legler, T., May, N., Scheirle, B., Sattler, K.U.: Cost modelling for optimal data placement in heterogeneous main memory. *Proc. VLDB Endow.* **15**(11), 2867–2880 (jul 2022). <https://doi.org/10.14778/3551793.3551837>
28. Lozi, J.P., Lepers, B., Funston, J., Gaud, F., Quéma, V., Fedorova, A.: The Linux scheduler: a decade of wasted cores. In: Proceedings of the Eleventh European Conference on Computer Systems. EuroSys '16, Association for Computing Machinery, New York, NY, USA (2016). <https://doi.org/10.1145/2901318.2901326>
29. Lütke Dreimann, M., Friesel, B., Spinczyk, O.: HetSim: A Simulator for Task-based Scheduling on Heterogeneous Hardware. In: Companion of the 15th ACM/SPEC International Conference on Performance Engineering. pp. 261–268. ICPE '24 Companion, Association for Computing Machinery, New York, NY, USA (5 2024). <https://doi.org/10.1145/3629527.3652275>
30. Noll, S., Teubner, J., May, N., Böhm, A.: Analyzing memory accesses with modern processors. In: Proceedings of the 16th International Workshop on Data Management on New Hardware. DaMoN '20, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3399666.3399896>
31. Olachea, R., Fahrenberg, U., Atlee, J.M., Legay, A.: Long-term average cost in featured transition systems. In: Proceedings of the 20th International Systems and Software Product Line Conference. p. 109–118. SPLC '16, Association for Computing Machinery, Beijing, China (2016). <https://doi.org/10.1145/2934466.2934473>
32. Sincero, J., Schirmeier, H., Schröder-Preikschat, W., Spinczyk, O.: Is the Linux kernel a software product line? In: van der Linden, F., Lundell, B. (eds.) Proceedings of the International Workshop on Open Source Software and Product Lines. SPLC-OSSPL '07, Kyoto, Japan (9 2007)
33. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* **13**(3), 260–274 (2002). <https://doi.org/10.1109/71.993206>
34. Ullman, J.D.: NP-complete scheduling problems. *Journal of Computer and System sciences* **10**(3), 384–393 (1975)
35. Valov, P., Petkovich, J.C., Guo, J., Fischmeister, S., Czarnecki, K.: Transferring Performance Prediction Models Across Different Hardware Platforms. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering. p. 39–50. ICPE '17, Association for Computing Machinery, New York, NY, USA (2017). <https://doi.org/10.1145/3030207.3030216>

36. Wang, M., Wang, H., Qiao, S., Chen, J., Xie, Q., Guo, C.: Heterogeneous system list scheduling algorithm based on improved optimistic cost matrix. *Future Generation Computer Systems* **164**, 107576 (2025). <https://doi.org/10.1016/j.future.2024.107576>
37. Zhang, L., Li, K., Li, C., Li, K.: Bi-objective workflow scheduling of the energy consumption and reliability in heterogeneous computing systems. *Information Sciences* **379**, 241–256 (2017). <https://doi.org/10.1016/j.ins.2016.08.003>