FARSI: An Early-stage Design Space Exploration Framework to Tame the Domain-specific System-on-chip Complexity

BEHZAD BOROUJERDIAN, University of Texas at Austin and Meta, USA
YING JING, University of Illinois at Urbana-Champaign , USA
DEVASHREE TRIPATHY, Harvard University , USA
AMIT KUMAR, Meta, USA
LAVANYA SUBRAMANIAN, Meta, USA
LUKE YEN, Tenstorrent Inc., USA
VINCENT LEE, Meta, USA
VIVEK VENKATESAN, Meta, USA
AMIT JINDAL, Peloton Interactive Inc., USA
ROBERT SHEARER, Meta, USA
VIJAY JANAPA REDDI, Harvard University and The University of Texas at Austin , USA
https://github.com/facebookresearch/Project_FARSI

Domain-specific SoCs (DSSoCs) are an attractive solution for domains with extremely stringent power, performance, and area constraints. However, DSSoCs suffer from two fundamental complexities. On the one hand, their many specialized hardware blocks result in complex systems and thus high development effort. On the other hand, their many system knobs expand the complexity of design space, making the search for the optimal design difficult. Thus to reach prevalence, taming such complexities is necessary. To address these challenges, in this work, we identify the necessary features of an early-stage design space exploration (DSE) framework that targets the complex design space of DSSoCs and provide an instance of one such framework that we refer to as FARSI. FARSI provides an agile system-level simulator with speed up and accuracy of 8,400× and 98.5% compared to Synopsys Platform Architect. FARSI also provides an efficient exploration heuristic and achieves up to 62× and 35× improvement in convergence time compared to the classic simulated annealing (SA) and modern Multi-Objective Optimistic Search (MOOS). This is done by augmenting SA with architectural reasoning such as locality exploitation and bottleneck relaxation. Furthermore, we embed various co-design capabilities and show that, on average, they have a 32% impact on the convergence rate. Finally, we demonstrate that using development-cost-aware policies can lower the system complexity, both in terms of the component count and variation by as much as 60% and 82% (e,g., for Network-on-a-Chip subsystem), respectively.

Authors' addresses: Behzad Boroujerdian, behzadboro@gmail.com, University of Texas at Austin and Meta, USA; Ying Jing, yingj4@illinois.edu, University of Illinois at Urbana-Champaign , USA; Devashree Tripathy, devashree.tripathy@gmail.com, Harvard University , USA; Amit Kumar, amit34@fb.com, Meta, USA; Lavanya Subramanian, lavanyas@fb.com, Meta, USA; Luke Yen, lyen@tenstorrent.com, Tenstorrent Inc., USA; Vincent Lee, vtlee@fb.com, Meta, USA; Vivek Venkatesan, vvenkatesan@fb.com, Meta, USA; Amit Jindal, amit.jindal@onepeloton.com, Peloton Interactive Inc., USA; Robert Shearer, robshearer@fb.com, Meta, USA; Vijay Janapa Reddi, vj@eecs.harvard.edu, Harvard University and The University of Texas at Austin, USA
https://github.com/facebookresearch/Project_FARSI.

XXXX-XXXX/2022/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

PS: This paper targets the Special Issue on Domain-Specific System-on-Chip Architectures and Run-Time Management Techniques.

CCS Concepts: • **Computer systems organization → System on a chip**; **System on a chip**.

ACM Reference Format:
Behzad Boroujerdian, Ying Jing, Devashree Tripathy, Amit Kumar, Lavanya Subramanian, Luke Yen, Vincent Lee, Vivek Venkatesan, Amit Jindal, Robert Shearer, and Vijay Janapa Reddi. 2022. FARSI: An Early-stage Design Space Exploration Framework to Tame the Domain-specific System-on-chip Complexity. 1, 1 (June 2022), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the end of Moore's law, general-purpose processors do not provide a path forward for domains with stringent power/performance/area constraints. This is due to the performance and energy inefficiencies of these systems/processors, such as the high instruction fetch and decode overhead. Domain-specific SoCs (DSSoCs) are a sound alternative solution. These are SoCs that use domain-specialized hardware blocks to keep the computation/communication fast and efficient. Consequently, they provide the performance and energy scalability required for highly constraints domains.

DSSoC Design and Development Challenges: Although efficient, the development of DSSoCs suffers from two sources of complexities, namely system, and design space complexity. These systems suffer from "system complexity" and thus demand high development efforts to design. This complexity is driven by the number and the variation (heterogeneity) of the processing elements and the intricate topological structure connecting them. From the compute perspective, increasingly customized accelerators accompany general-purpose processors; from the communication perspective, both the Network-on-a-Chip (NoC) and memory subsystems see a complexity rise to keep the data local and the movement low energy [76]. DSSoCs also suffer from "design space complexity", as the sheer number of design knobs per component dramatically expands the design space making the search for the optimal design difficult. For example, a simple system with five simple workloads and six total knobs (e.g., frequency, bus width, accelerator hardening knobs) totals over a million design points. Naive brute force sweeps are infeasible for the design space exploration of DSSoCs.

To manage the system and design space complexities, we need a design space exploration (DSE) technique that is agile/efficient to navigate the space. DSEs typically consist of two components, a simulator (Figure 1, top left) to capture the design behavior and an exploration heuristic (Figure 1, bottom left) to navigate the design space efficiently. To this end, we provide an efficient early-stage (pre-synthesis) DSE, called FARSI, that contains an agile simulator and efficient heuristic targeting DSSoCs complexity. Our work is open-sourced: https://github.com/facebookresearch/Project_FARSI.

DSSoC Simulation Requirements: DSSoC simulators must estimate system-level dynamics since profiling components in isolation cannot accurately measure their system-level impact. Furthermore, these simulators must be agile to enable sufficient coverage of the vast design space. Many state-of-the-art simulators either focus on single components, e.g., accelerators [73, 75] and memory [67], or the one with system wide focus do not provide sufficient agility [74], or accuracy [34]. In contrast, FARSI's system-level simulator captures the complexity of accelerator-rich systems. It also achieves high agility/accuracy by combining analytical models' speed and the phase-driven simulation's accuracy. Our analytical models expand roofline-based Gables [34], and our phase-driven simulation uses the "phase" concept, the longest duration with a fixed system bottleneck. A phase can contain many transactions, thus giving us higher agility compared to state-of-the-art transactional models.

DSSoC Heuristic Requirements: DSSoC heuristics must have a host of features (Figure 1, bottom right, green). These heuristics need to be *"domain-aware"* to exploit workload-specific opportunities like loop-level parallelism. They need to apply *"architectural reasoning"* to identify and relax system

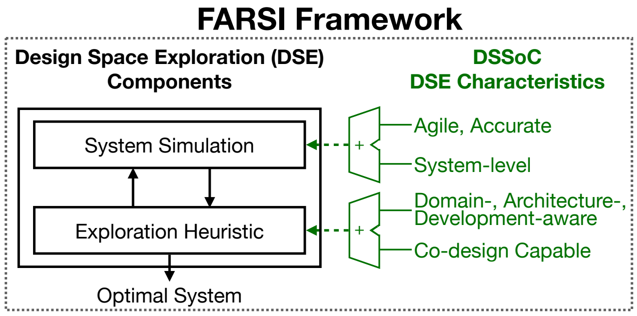


Fig. 1. DSE components need the characteristics shown in green to tame the DSSoC design space complexities. We introduce FARSI, an early-stage DSE framework with those characteristics.

bottlenecks and efficiently prune the design space. They need to apply *co-design* to optimize across boundaries such as workloads/design stages to accelerate convergence. Finally, they need to be *"development cost-aware"* to keep system complexity and thus the development effort low. However, state-of-the-art either lacks said features, only contains a subset of them or applies them to limited sub-system scope. For example, [30, 63] lack architectural reasoning, and [91] only applies them to the NoC design. [68, 82] only co-design across a subset of design stages. [18, 78] only targets fabrication cost instead of development cost, and [62] only reduces the processor's development cost.

FARSI exploits architectural reasoning and co-design and further applies them at the system-wide level to address the said shortcoming. FARSI uses architectural insights to find system bottlenecks and exploits parallelism or locality to relax them. This approach improves the convergence rate by quickly finding and navigating toward high potential neighboring designs (Section 4.3.2). FARSI also uses co-design across multiple design vectors, specifically (1) across design stages (topology generation, allocation, and mapping), (2) workloads, (3) metrics, and (4) computation-communication. Holistically co-designing across boundaries such as workloads boundaries improves the convergence rate (Section 4.3.3). Finally, FARSI deploys a development-aware heuristic by prioritizing low development-cost optimizations, which in turn lowers the system complexity (Section 5.1).

Evaluation: FARSI's framework is general and can be applied to any domain without modifications. Without the loss of generality, we evaluate FARSI for the augmented reality (AR) domain as it is a great candidate for DSSoC design. On the one hand, AR's tight budgets require a host of high-efficiency specialized hardware units [37]. On the other hand, its diverse sub-domains, including video, graphics, computer vision, and audio, seek holistic designs that go beyond each sub-domain.

We validate FARSI simulator's fidelity against Synopsys' Platform Architect [66], an industry-strength early-stage SoC simulator, for over 250 designs of different complexities and achieve an average speedup of 8,400× at the accuracy of 98.5%. In addition, we quantify our simulator's error sensitivity and speedup across multiple software/hardware knobs.

We compare FARSI's heuristic against the classic simulated annealing (SA) and more modern Multi-Objective Optimistic Search (MOOS) [16] and achieve up to 62× and 35× speedup, respectively. We also evaluate FARSI's efficiency for other metrics such as Quality Gain and Pareto Hyper Volume and demonstrate the impact of architectural awareness and various co-design vectors.

Finally, we provide two case studies demonstrating FARSI's capabilities. First, we show how development-awareness lowers the design complexity by decreasing the number and variation of components when system budgets such as latency are relaxed. Second, we compare FARSI with the divide and conquer approach that is commonly used to manage complexity, reveal its problems, and show how FARSI mitigates them by 56% and 52% for power and performance, respectively.

In short, FARSI reveals a methodology for efficiently managing the system and the search space complexity of DSSoCs. Our major contributions are outlined as follows:

- We provide the first Gables-based [34] characterization of a host of AR workloads from three different AR application primitives—audio decoding [37], CAVA [88], and edge detection [47] (Section 4.1 and Appendix).
- We present a hybrid estimation methodology that combines analytical models and phase-driven simulation and can provide both the agility (8,400× speed up) and the accuracy (98.5%) much needed for DSSoCs. Our simulator lowers the convergence time from (estimated) 3 years to 3 hours compared to Synopsys Platform Architect (Section 4.2).
- We show how architectural reasoning (e.g., locality exploitation or bottleneck relaxation) and co-design can improve the convergence time of classic search heuristics such as simulated annealing and modern data-driven ones such as MOOS (Section 4.3).
- We present a case for FARSI and show how development-aware policies (e.g., prioritizing incremental improvements) can exploit relaxed budgets and lower the final system complexity, both in terms of component counts and variations by more than 80% (Section 5).

2 RELATED WORKS

This section presents an overview of the prior work in the Design Space Exploration (DSE) landscape. DSEs have two components, a simulator and an exploration heuristic. We provide related work for each component and further detail FARSI's solution to address their shortcomings.

2.1 Simulators

Simulators targeting DSSoCs need to be agile and accurate. Here we detail related simulators and their suitability for DSSoCs and present explanations of how FARSI differs and improves upon prior art.

Cycle-accurate: Cycle-accurate simulators model components at a very low-level [8, 9] whereas FARSI uses high-level analytical+phase-driven models. Although cycle-accurate simulators offer higher accuracy, their extreme low agility is not suitable for DSSoCs.

Trace-driven: They address the cycle-accurate model's lack of agility by predicting the performance of the system using traces of the program. State-of-the-art trace-driven simulators such as [7, 44, 67, 73, 75] *only focus on individual components*, such as accelerators [44, 73, 75] or memory [7, 67]. In contrast, as DSSoCs demand, FARSI models the entire SOC's system-level dynamics. Although works such as [73, 75] are later augmented to enable system-level modeling, they suffer from non-agile cycle-level simulation.

Transaction-level (TLM): They lower the modeling fidelity from individual communication signals to transactions. This reduces the synchronization points, lowers simulation time, and thus renders them ideal for early-stage SoC analysis [10, 66, 72]. Instead of using transactions, FARSI relies on the concept of "phase", the longest duration with a fixed system bottleneck. A phase can contain many transactions, thus giving FARSI higher agility. Furthermore, unlike TLM-based simulators, FARSI combines event-driven simulators with analytical models to gain further agility.

Analytical modeling: They use mathematical models to capture the system behavior. Works such as [34] and roofline [87] use bottleneck analysis, others such as ETP [12], GPGPUAn [77] and logCA [4] use performance prediction to capture the execution time, and [5, 54] deploy a hybrid analytical model/iterative algorithm to improve their estimation fidelity. Relying solely on mathematical models lowers their fidelity. Furthermore, the static nature of these models fails to capture complex SoCs' behavior, such as the data flow of a complex task graph. To improve on them, FARSI augments these analytical models with a phase-driven simulator to capture said complexities.

2.2 DSE Optimization strategies

Heuristics targeting DSSoC needs to be equipped with the following features to efficiently navigate the design space complexity and lower the system complexity.

Co-design Across System Components, Design Stages, and Workloads: Design space exploration of DSSoCs involves multiple optimizations across system components, workloads, and design stages (e.g., topology generation, allocation, and mapping), all impacting the exploration convergence (Section 4.3.3). However, state-of-the-art solutions focus on a subset of these problems. For example, [38] only targets optimization of multi-core processors, [22] customizes out-of-order, and [6] customizes VLIW processors, [15] targets memory, and [16, 42] explore NoC design. In contrast, FARSI takes a holistic system view, optimizing all components depending on system needs. For example, if a design suffers from memory contention, it exploits task re-mapping to reduce the traffic, while for systems bottlenecked by processors, it swaps general-purpose cores with accelerators.

Other works that take a holistic view similar to FARSI only target a subset of DSSoC's design stages. DSSoCs design stages include (1) system topology generation (also known as architectural template), (2) hardware allocation/customization, and (3) task to hardware mapping/scheduling. A DSSoC DSE must conduct all these steps and further exploit the co-design opportunities across them. Many solutions target one stage, for example [21, 36, 40, 49, 61, 71] target mapping and [43, 46, 53, 69, 81, 84] target scheduling/resource management. Others prior works combine a subset of these stages. [52, 68, 82] combine mapping and allocation, and [5] combine scheduling and allocation, ignoring system topology generations. [26] target all said stages similar to FARSI; however, they do them sequentially one after another and thus missing on co-optimization across them. In contrast, we simultaneously exploit co-design across all three stages. [39, 51] most resemble FARSI as they conduct all three stages simultaneously; however, they lack multi-workload considerations, and furthermore, their heuristics, i.e., genetic algorithms and ILPs, lack architectural reasoning, which, as detailed later, lowers their convergence rate. Please note that the current version of FARSI only deploys a *first come, first serve* scheduling algorithm, which we plan to augment in the future work using the more advanced list and table-based schedulers provided in [5, 84].

Other works capture domain-specific designs that run multiple concurrent applications and apply cross-workload optimizations. [90] extracts functional and structural similarities among applications, and [86] determines application combinations that improve the system performance. Although they exploit cross-workload opportunities, both said works only focus on the mapping stage, and thus, in contrast to FARSI, they can not exploit cross-design-stage opportunities.

Architectural Reasoning: SoC DSE should use architectural insights such as bottleneck relaxation and parallelism/locality exploitation instead of blind exploration to accelerate the convergence. Various heuristics have been used for SOC DSE such as simulated annealing [14, 19], integer linear programming [30, 60], particle swarm [63], genetic algorithms [17, 64], and reinforcement learning [46]; However, these heuristics' generality, i.e., lack of understanding of the design space, degrades their convergence rate. Solutions have been proposed to improve said heuristics with domain knowledge/reasoning. [25] uses Fuzzy logic for efficient space pruning, and [91] augment GA's mutation/cross-over with domain knowledge to increase solution quality. However, both of these solutions only target a more limited single component optimization of processor and NoC design, respectively. Others like [70] augments GA with special mutations according to task-hardware affinity, and [83] filter out symmetrical designs in GA to prune the space efficiently. However, both works focus on a more limited single-stage (mapping) optimization. In contrast, FARSI's architectural reasoning is applied across all three design stages for all components.

Development Awareness: Economic thinking has been deployed within the architecture community, and many works focus on minimizing silicon fabrication cost. [78] considers component cost, [18] considers 3D IC cost, and [13] shows the financial risk of process uncertainties on chip seller's profit. Others such as [11, 31, 32] focus on minimizing the data centers operating (electricity) cost. None of these target "development costs" and hence, complement FARSI.

Other works have modeled the development cost. [23] uses VHDL line of code, [2] uses the number of independent paths in the control data flow graph, [24] uses internal I/O signals and component count and [3] use engineering and new protocol cost as development cost. These works focus on modeling and, thus, in contrast to FARSI, do not provide DSE heuristics for cost reduction. [62] minimize the development cost and is the closest work to us. However, they target a more limited only processor optimization problem and thus do not address development reduction for other system components. In contrast, FARSI targets system-wide development cost reduction necessary for DSSoCs by lowering the heterogeneity and customization/allocation among all components.

3 METHODOLOGY

In this section, we detail FARSI DSE's methodology. FARSI is comprised of four stages (Figure 2), (1) database generation, (2) system simulation, (3) system generation, and (4) system selection. The first stage characterizes our workloads, collects (in isolation) IP performance/power/area (PPA) estimates, and populates our software/hardware database. This database is then continuously queried by our system simulator, system generator, and system selector as they work together to generate new systems and improve them. This process iteratively continues until the design's budgets (i.e., performance, power, and area) are met. FARSI's main contribution is in the last three stages as prior work such as [47, 75, 89] sufficiently addresses the first.

3.1 Database Generation

This stage populates a database with the characteristics of the domain's workloads and the PPA estimates of individual IPs. Both of these components are then used to estimate the system behavior.

Workload Analysis (Software Database): This involves generating a task dependency graph (TDG) shown in Figure 6. A task is the smallest unit of optimization and is typically selected from the computationally intensive functions since they significantly impact the system behavior. Selecting functions to build TDG is highly important as they determine the design space knobs. For example, separating functions that can run concurrently in different tasks exposes the workload parallelism, while fusing serial functions into one task lowers data movement and thus energy consumption. Note that such optimal task selection and automated task dependency identification is outside of this work's scope, and thus, for these efforts, we rely on application developers' insights. Concretely, we have asked ILLIXR [37] (the benchmark used in this paper) developers to identify important tasks and further specify the dependency among them.

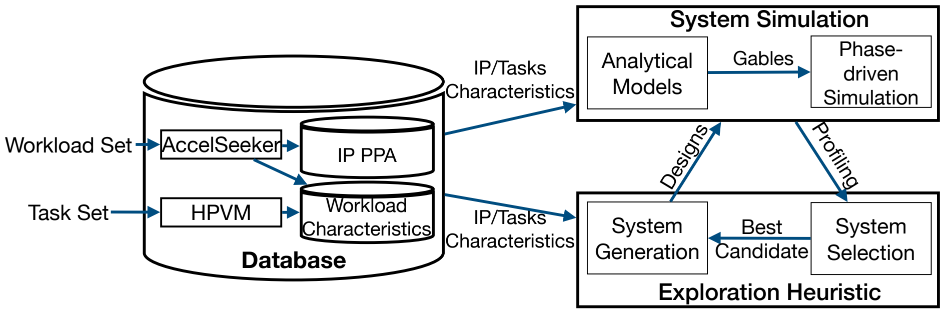


Fig. 2. FARSI's end-to-end methodology.

Variable Type	Variable	Description
Hardware	B_{peak}	Peak bandwidth (bytes/sec)
	B_{Mem} , B_{NoC}	Actual memory, NoC bandwidth (bytes/sec)
	P_{CPU_peak}	Peak performance of CPU (ops/sec)
	P_{CPU} , P_{IP}	Actual CPU, IP performance (ops/sec)
	A_{peak}	Peak acceleration (unitless)
	L	Number of links for a NoC (unitless)
Software	T	Task
	f_{IP}	Task's work for an IP (ops)
	f_{CPU}	Task's work for CPU (ops)
	D	Total task's data transferred (bytes)
	$Burst$	Burst size for a task (bytes)
	C_T	Completion time of task T (sec)
Timing	$\Phi_{Duration}$	Duration of a phase (sec)

Fig. 3. Analytical model parameters.

Once the tasks are selected and the TDG is built, TDG's nodes are augmented with computational characteristics (e.g., loop iteration and instruction counts), while edges augmented with communication data movement of tasks. For this effort, we profile our tasks using existing tools such as perf [57], AccelSeeker [89] and HPVM [47].

IP Analysis (Hardware Database): This involves PPA estimation of each task for different hardware mappings (e.g., to general-purpose processors or specialized accelerators). We build on top of an existing toolset called AccelSeeker [89] and collect PPA estimates for each mapping. Note that other pre-RTL tools such as Aladdin [74, 75] also can be deployed for this stage. We augment our database with many memory and NoC widths and frequencies.

3.2 System Simulation

This stage raises the view to the system level and enables holistic analysis. System view is necessary since profiling components in isolation cannot accurately measure their overall impact within complex and accelerator-rich DSSoCs. This stage considers computation (general-purpose processors and accelerators) and communication IPs (NoCs and Memory) as the lowest abstraction unit.

We deploy a hybrid estimation methodology combining analytical models and a lightweight phase-driven simulator. The former enables agile traversal and thus an extensive design space coverage. The latter improves the simulation fidelity by capturing the system dynamics. Here we discuss each.

Software Analytical Models: We build on top of Gables SoC-level roofline models [34]. These models, along with other roofline-based models, have been gaining traction for various DSEs such as CNN design [65] and cyber-physical co-design [48]. Gables combine bottleneck analysis and high-level computational/communication estimates to capture the system performance. Their simplicity yet capability to capture the system-level behavior makes them a prime candidate for DSSoCs. We augment Gables with the following improvements:

- *Finer Computation/Communication Granularity:* We lower the smallest execution unit from workload to task. This captures the workload's lower-level compute, memory, and communication characteristics and further provides extra within-workload optimization opportunities. Also, we introduce a communication burst size parameter that captures NoC congestion behavior more accurately.
- *Task Dependencies:* The Gables model assumes parallel execution of all workloads/tasks for simplicity. We use TDGs to model task-level parallelism (TaLP) accurately.
- *Computation/Communication Breakdown:* We augment Gables with loop iteration count and thus loop level parallelism (LLP). We also break down I into read and write operational intensity (I_{read}, I_{write}) as modern routers/memories support separate channels for each.

Hardware Analytical Models: Gabels models hardware with peak performance (P_{peak}) and bandwidth (B). It assumes a single NoC with a single channel. Our improvements include:

- *Computation/Communication Resource Sharing:* We integrate CPU's multi-tasking models (pre-emptive scheduling), and multi-channel routers for master-slave combinations.
- *Topology Improvement:* "many NoC" topology systems are used to improve congestion/locality.

Deploying Analytical Models: Our analytical models estimate each task's completion time by calculating its hardware blocks' processing rates, i.e., instruction per second for processors and bandwidth for memory/NoC. These rates are detailed in equations 1 through 4 with T and $Burst$ denote a task and its burst size, P_{peak_CPU} and P_{CPU} denote a general-purpose processor's peak and actual performance, A_{peak} and P_{IP} denote peak accelerator speedup and its actual performance, B_{peak_Mem}, B_{peak_NoC}, B_{Mem}, B_{NoC} denote memory and NoC peak and actual bandwidth, $Init$ denotes number of initiators in a NoC, and $|\ \ |$ denotes the cardinality (e.g., $|T|$ denote number of running tasks).

To calculate the processing rate of CPU (Eq. 1), and IP (Eq. 2) per task, their peak rate is divided by the number of tasks running on them. This is because preemptive scheduling of CPU/IP will allocate the rate equally among the tasks. To calculate the NoC's bandwidth (Eq. 3), its peak bandwidth is divided among the number of initiators (e.g., processors). This is because NoC's equal arbitration allocates the bandwidth equally among initiators.[1] Furthermore, for an initiator that runs concurrent tasks, each task's bandwidth is scaled proportionally to its burst size.[2] Memory bandwidth modeling follow NoCs (Eq. 4).

$$P_{CPU} = \frac{P_{peak\_CPU}}{|T|} \quad (\frac{ops}{sec}) \qquad (1) \qquad\qquad P_{IP} = \frac{A_{peak} * P_{peak\_CPU}}{|T|} \quad (\frac{ops}{sec}) \qquad (2)$$

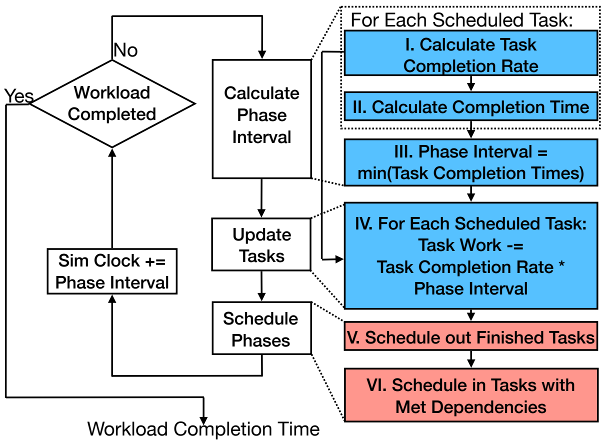
$$B_{NoC} = \frac{B_{peak\_NoC}}{(\frac{Burst_i}{\sum_i Burst}) * |Init|} \quad (\frac{byte}{sec}) \qquad (3) \qquad\qquad B_{Mem} = \frac{B_{peak\_Mem}}{(\frac{Burst_i}{\sum_i Burst})} \quad (\frac{byte}{sec}) \qquad (4)$$

$$C_T = \max(\frac{f_{IP}}{P_{IP}}, \frac{D}{B_{Mem}}, \frac{D}{B_{NoC}}, ...) \quad (sec) \qquad (5) \qquad\qquad \Phi_{Duration} = \min_i(C_{T_i}) \quad (sec) \qquad (6)$$

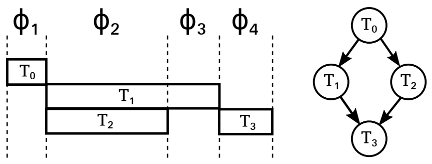
Once each block's processing rate per task is determined, the completion time of the task is determined by its slowest block (Equation 5). T and C, and f_{IP} denote a task, its completion time, and its work for an IP. Each component in the tuple calculates the execution time of a different block where each block's work (e.g., data or D for memory) is divided by its execution rate (e.g., bandwidth for memory). The maximum function finds the slowest of blocks, and the number and type of its inputs are determined by the blocks hosting the task.

Phase-driven Simulation: Gables is a static model that cannot accurately capture the dynamic flow of complicated task graphs. So we wrap our analytical models with a lightweight *phase-driven* simulation. A *phase* is a flexible time quantum with which we advance the simulation. It specifies the longest interval that the system behavior stays constant; Since our models use bottleneck analysis to estimate the system behavior, a new phase emerges when any task's hardware bottleneck either shifts or its processing rate changes. Such a change is triggered when new tasks are scheduled in or old ones scheduled out, resulting in an increase or relaxation of NoC/Memory/PEs pressure (Figure 4b). A phase duration is determined by the current fastest running task and thus estimated by the minimum of all tasks' completion times (Equation 6). $\Phi_{Duration}$, and C_{Ti} denote the phase duration and the task i's completion time.

[1]We assume a star topology if multiple initiators talk to the same target, otherwise a point to point one.
[2]Our models can be easily extended for cooperative scheduling and non-equal arbitration; Details are left out for brevity.


For Each Scheduled Task:
I. Calculate Task Completion Rate
II. Calculate Completion Time
III. Phase Interval = min(Task Completion Times)
IV. For Each Scheduled Task: Task Work -= Task Completion Rate * Phase Interval
V. Schedule out Finished Tasks
VI. Schedule in Tasks with Met Dependencies
No
Yes
Workload Completed
Calculate Phase Interval
Update Tasks
Schedule Phases
Sim Clock += Phase Interval
Workload Completion Time

(a) Phase driven simulation.



(b) Phases for an example workload. ϕ and T denote a phase and task respectively.

Fig. 4. FARSI's simulation. It is based on the concept of phase (right) where each tick moves the simulation forward one phase. Analytical models shown in blue boxes estimate the phase duration/task progression rate and red-boxes schedule out/in old/new phases.

The phase concept is fundamentally different from transactional models, whose agility has made them a prime candidate for DSEs. For transactional models, the simulation advances one transaction at a time, thus scaling according to the transaction count. In contrast, a phase can contain many transactions if the bottleneck stays constant across many transactions, allowing us to tick through them quickly. This flexibility of phase enables agility when moving through many transactions at once and otherwise, provides accuracy when the bottleneck shifts a few transactions at a time. We quantify these characteristics later in detail.

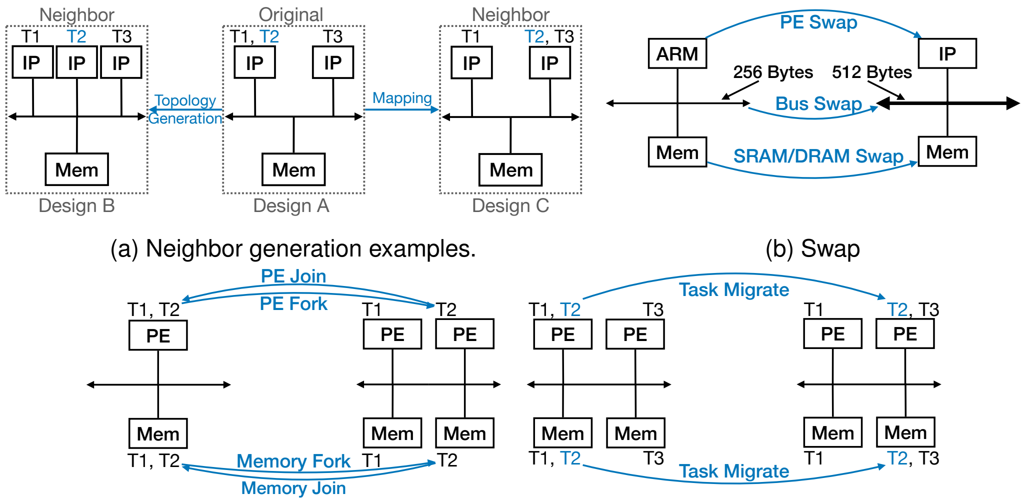
Hybrid Estimation: Our hybrid approach estimates the SOC performance by combining the said analytical models and phase-driven simulation as shown in Figure 4a. Blue color boxes denote stages where analytical models are deployed, while red boxes denote stages where the phase-driven simulator moves the simulation forward. At the beginning of each phase, the simulator schedules all the tasks whose dependencies are satisfied.[3] Then, our analytical models determine each task's completion time and use it to calculate the current phase duration (stages 1, 2, and 3). Once said duration is calculated, the simulator moves the simulation tick to the end of the phase, updates each task's progress according to the duration (stage 4), schedules out the completed task (stage 5), and schedules in ready tasks (stage 6). Currently, stage 6 models an OS with a first-ready, first-serve scheduling policy with tasks being scheduled upon being ready. It further models preemptive scheduling with equal time-sharing across tasks. Other scheduling policies can be easily incorporated into this stage if needed.

Power/Area Modeling: For power and area of IPs, we use the verified AccelSeeker [89] estimations, and for NoCs and memory, we have fully integrated CACTI [58] into FARSI.

3.3 Exploration Heuristic: System Generation

This stage exploits architectural insights to explore the design space. Without the loss of generality, we use simulated annealing (SA) as the search heuristic base as this classic search is used in many

[3]Note that for our simulated designs, task to hardware mapping is determined at design time and provided to the simulator as an input. Thus, scheduling involves only determining the time to run the task, not the hardware block to run it on.



(a) Neighbor generation examples. (b) Swap

(c) Fork/Join (d) Migrate

Fig. 5. Neighbour generation with "Optimization moves" supported in FARSI. Each move incrementally modifies the design. We have provided a move for allocation/customization (Swap), topology generation (Fork/Join) and software to hardware mapping (Migrate). PE = processing elements like CPU or accelerator (IP). Mem = Memory.

DSEs [20, 41, 41, 50].[4] We augment SA with various architectural insights. To explore the design space, we greedily generate a number of designs' neighbors (candidate designs in SA), simulate them and select the best one until the constraints are met (Figure 2). A design's *neighbor*, is a design whose mapping, allocation or topology has been incrementally modified (Figure 5a). For example, designs B and C are design A's neighbors as the former has an allocation change (an extra IP) and the latter a mapping change (Task 2).

We generate neighbors by selecting a five-tuple, i.e., a Metric, a Direction, a Task, a Block, and an Optimization Move to improve the said components.

Metric/Direction: To generate a neighbor, we target one metric per iteration (e.g., performance). We typically pick the metric farthest from its budget as it contributes to the distance (to goal) the most. Note that this metric can vary from iteration to iteration as we improve said metrics. Overall, FARSI currently targets performance, power, and area. In addition to the metric, we also target a direction to improve the metric with (e.g., decrease or increase if we have overshot).

Task/Hardware Block: To generate a neighbor, we target a task/block to improve, typically the one causing the highest distance to budget. For example, if targeting latency, the highest latency task and its block bottleneck are selected.

Optimization Move: *Move* is an optimization on a task/block to improve a metric. FARSI supports high-level optimizations such as hardware customization/allocation, topology generation, and software to hardware mapping and provides a move primitive for each, concretely swap, fork/join, and migrate.

Swap: Impacts allocation and enables customization by replacing a hardware block with a more specialized one (Figure 5b). For example, we replace an ARM core with an accelerator or a narrow bus with a wider one. Table 1 details our swap options. Note that the swap only incrementally modifies the original block (e.g., 100MHz→200MHz instead of 100MHz→800MHz) to minimize the impact on competing metrics (e.g., performance vs. power).

[4]Note that we are not limited to SA, and our ideas can be deployed in other heuristics, e.g., Tabu Search [27].

Table 1. Swap types. PE=processing element, GPP=general purpose processors, Acc = accelerator.

Block Type	Subtype	Freq (MHz)	Bus Width (Bytes)	Loop Unrolling
PE	GPP⇆Acc	[100,...,800]	N/A	According to the task
NoC	N/A	[100, ,...,800]	[4, ..., 256]	-
Mem	DRAM⇆SRAM	[100,...,800]	[4, ..., 256]	-

Fork/Join: Impacts the topology by duplicating an existing block and migrating some of its tasks over (Figure 5c). Duplication relaxes the block pressure, e.g., cutting down a NoC traffic and relaxing congestion. Join is the opposite of fork.

Migrate: Modifies software to hardware mapping by migrating a task from one block to another (Figure 5d). Migration improves data locality (moving data closer to where it is used), load balancing (mapping to an idling processor), and relaxes buses/memory pressure (mapping to bus/memory with the lower congestion). Currently, mapping is done statically.

Using Architectural Reasoning for Move Selection: FARSI's exploration heuristic is equipped with architectural reasoning, concretely the capability to apply parallelism, locality, and customization according to the system needs to improve the convergence rate. For example, to improve power/performance, it automatically detects if a task's data is multiple hops away and chooses "migrate" to bring the data closer (spatial locality reasoning). Furthermore, to improve performance, it automatically recognizes if a block runs concurrent, parallel tasks and uses "fork" or "migrate" to relax the pressure/congestion (parallelism reasoning). Alternatively, if the metric to optimize is power, it uses "join" to "serialize" to improve power or reduce area. This automatic capability to reason about system/targeted metric and applying appropriate optimizations improves the convergence rate and is necessary for complex design spaces of DSSoCs.

At a high level, to select optimization moves (Algorithm 1), FARSI first applies architectural reasoning to find moves that can improve the design (step I), prioritizes them according to their development cost (detailed in next paragraph) (step II), and probabilistically samples and applies the move (step III). This approach improves the random neighbor generation (or move selection) of SA and increases its convergence rate by more than an order of magnitude (section 4.3.2). Note that said reasonings is not tied to a specific search heuristic, and although this work applies them to simulated annealing, they can improve other heuristics' convergence rate. Also, note that FARSI users, i.e., system designers and architects, can easily extend said reasonings with their architectural insights to improve FARSI's intelligence.

Move Symmetry: Our optimization moves are symmetrical to enable backtracking and prevent the navigation from getting stuck. For example, we can swap up followed by a swap down (e.g., first widen and then narrow the bus), migrate back and forth, and join-in after forking-out.

Development-cost Awareness: Since highly customized SoCs are complex and thus financially costly to develop, we embed various policies into our heuristic to keep the development-cost low:

- We introduce move-precedence to select low-effort moves over the expensive ones. For example, we prioritize join over other moves as it reduces the hardware complexity by eliminating a hardware block. Furthermore, we prioritize software moves (migrate) over the hardware ones (fork, swap) as software manipulation is cheaper. Finally, within hardware moves, we prioritize fork over swap for processing elements as the former only requires duplication, whereas the latter involves porting (or hardening), which is more expensive. For memory and NoCs, the swap is prioritized if it does not increase the system heterogeneity, for example, creating a system with NoCs of different Frequency. A snipped of our move precedence is shown in `move_precedence` of Algorithm 1.

Algorithm 1 Optimization move selection.

```
// Step I: Apply reasoning based on parallelism, locality and customization to find optimization
moves that can improve the design.
if $metric$ = "latency" then
    if $block$ has parallel tasks then
        $optimization\_moves$ ← ["migrate", "fork"]
    else
        $optimization\_moves$ ← ["swap", "fork_swap"]
if $metric$ = "power" then
    if task can run in parallel with other blocks' tasks then
        if $block$ has no parallel tasks then
            $optimization\_moves$ ← ["migrate"]
        else
            $optimization\_moves$ ← ["join"]
    else
        $optimization\_moves$ ← ["swap", "fork_swap"]
if $metric$ = "area" then
    if $block$ = "PE" then
        $optimization\_moves$ ← ["join", "swap"]
    else
        $optimization\_moves$ ← ["migrate", "join", "swap"]

// Step II: Prioritize moves based on their development effort.
move_precedence: join > migrate > fork > swap > fork_swap

// Step III: Select moves.
$move\_chosen$ ← probabilistically choose from $optimization\_moves$, based on $move\_precedence$
order

PS: fork_swap is simply a fork followed by swap, and is introduced to accelerate navigation.
```

- FARSI starts with a very simple base design (one general-purpose processor, a NoC, and one DRAM bank) and advances its complexity incrementally only if needed. Since every new allocation or customization is costly (the former introduces new hardware and thus increases the development effort while the latter contributes to the system heterogeneity), we only apply one move at a time and thus incrementally modify the design. Furthermore, our moves are designed to only modify one knob at a time, e.g., migrate only one task. Overall, this approach allows us to increase the complexity in small steps and only if the design has not met the budget, thus keeping the development effort low.

We detail the impact of our development-aware policies in a case study (Section 5.1).

3.4 Exploration Heuristic: System Selection

We select the best of the generated and simulated neighbors. Neighbors' fitness is quantified using their design's (metric) normalized distance to budget with a dampening factor to the metrics already meeting the budget (Equation 7). m, Des_m, Bud_m and α denote, a metric, design value for the metric, budget value for that metric, and dampening factor when the budget is met.

$$distance\ to\ budget = \sum_m \alpha * \frac{(Des_m - Bud_m)}{Bud_m} \qquad m \in \{Performance, Power, Area\} \tag{7}$$

If no improved designs are found, we inform the system generation stage to target the task/block with the next highest distance (comparing to the last) in the next iteration. In addition, similar to SA, a standard Metropolis acceptance criterion [55] is used to choose the sub-optimal designs. This criterion uses a probability value based on temperature and distance to budget to sample a worse design. Temperature is dynamically lowered as the search advances to tighten the exploration perimeter.

4 EVALUATION AND RESULTS

This section sheds light on FARSI's components' fidelity and scalability. We start with our experimental setup and simulation/heuristic baselines, detail our workload benchmarks, and finally quantify FARSI's improvements compared to our baselines.

Simulation Baseline and Experimental Setup: We validate our simulation framework against *Synopsys Platform Architect (PA)* [66], a widely used industry-grade performance simulator. Like FARSI, PA's approximately timed (AT) mode targets agile/early-stage system-level estimation. For processors, PA uses a task-driven model with the task's cycle count acquired as an input parameter according to the processor mapping. It further uses first-ready, first-served with preemptive/equal time-sharing scheduling. For NoCs and memories, PA uses TLM-2.0 based AT models where system components communicate based on function calls with a payload instead of individual signals [33]. This reduces the number of synchronization points and improves simulation agility.

Our fidelity studies examine designs generated for our representative workloads and synthetic ones. Concretely, (1) we validate our simulator for 250 SoCs of different complexity for our representative workloads, namely Audio Decoder, CAVA, and Edge Detection. These designs are generated by FARSI's heuristic while converging to the tight AR constraints. Their complexity range from 1 to 13 processing elements, 1 to 8 memory blocks, and NoCs with 1 to 3 routers, with an execution latency ranging from 5 ms to 52 s (Table 3b). Then, (2) we conduct a series of experiments using synthetic designs/workloads where we vary software/hardware knobs such as the number of parallel tasks and routers in a NoC to dissect the cause of error further.

Heuristic Baselines and Experimental Setup: To measure FARSI's heuristic's agility, we compare it against *simulated annealing (SA)* and *Multi-Objective Optimistic Search (MOOS)* [16]. SA's greedy search generates new design points at random and accepts them if they minimize the distance to the objectives or based on Metropolis acceptance criterion [55]. We choose SA since it is a classic heuristic used in many hardware design explorations [20, 41, 41, 50].

MOOS [16] is a modern heuristic that learns to find optimal starting points for its greedy search. Concretely, a tree model is used to select an optimal starting point from a Pareto front. Then a local greedy search updates the Pareto front and expands the set of starting points. These two steps repeat in a loop until convergence. Starting points' optimalities are measured based on their impact on Pareto front expansion (measured in Pareto Front Hyper Volume [85]) and are maintained and continuously updated in a tree. We have selected this heuristic due to its ability to learn and adapt to the search space. In addition, it has shown improvements over other Machine learning-based algorithms such as MOO-STAGE [42].

For our heuristic comparison studies, we envision a system running all three workloads with performance, power, and area budget according to [80], and [37] as shown in Table 3a. Concretely, [80] specifies the required power and area of augmented reality glasses, and [37] provides workloads'

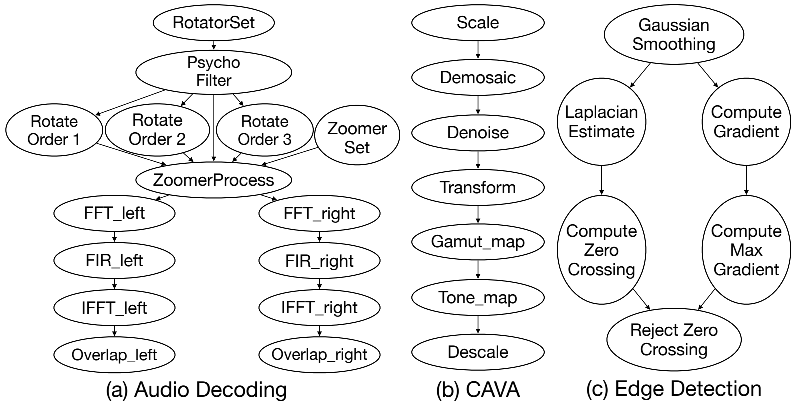


Fig. 6. Our workloads' task dependency graphs (TDGs).

latency to meet AR application's quality. Please refer to the appendix (Section C) for further calculations associated with the budget specs.

All data are collected on a Xeon Skylake Machine with 2.00 GHz frequency, and PA's time interval 10 µs (~ 1000-10000 cycles of our hardware blocks) for high accuracy.

4.1 Workloads

We evaluate FARSI for the Augmented Reality domain. However, note that FARSI's methodology is general and can be applied to other domains without modifications.

Augmented Reality (AR) Workloads: AR spans across many sub-domains and workloads. In this work, we target the primitive workloads, i.e., audio and image processing, as they are present in almost any AR applications [28, 29, 56]. Note that this work leaves out Facebook's proprietary internal workloads and only resorts to open-sourced libraries.

Audio Decoder (Audio) [37] is used to playback the audio based on the user pose. Concretely, said pose, obtained from the inertial measurement unit (IMU) integrator, is used to rotate and zoom the soundfield, which is then mapped to available listeners.

CAVA simulates a configurable camera vision pipeline and is used to process raw images (pixels) from sensors to feed the vision backend (e.g., a CNN). The default image signal processing (ISP) kernel is modeled after Nikon-D7000 camera [1] and is developed by Harvard [88].

Edge Detection (ED) [47], one of the key steps in image analysis and computer vision, processes the image and finds sharp changes in brightness to capture significant events and changes in properties of the world.

Workload Characteristics: To characterize each workload's execution flow, we split them into a set of tasks (e.g., Tone Map task in CAVA workload) where a task is the smallest unit of simulation and is selected from the workload's functions. The execution dependency between the tasks is captured in the task dependency graph (TDG), where each node specifies tasks and edges specify their dependencies. Figure 6 provides each workload's TDG. Audio has the most tasks (15), while Edge Detection has the least (6). In general, this distribution of tasks is in line with several internal proprietary workloads.

To characterize the tasks, we use Gables [34]. For the first time, we provide a Gables profile of a set of AR workloads. Gables is a set of abstract analytical models that captures high-level software computational and communicational characteristics. Concretely, it models each task's computation with its work (f = instruction count) and its communication with operational intensity (I =operation count per memory access, which is split to I_{read} and I_{write} corresponding to read and

Table 2. Gables-based characteristics. K=kilo, M=million.

Workload	Audio	CAVA	ED
f (M.ops)	13	24,252	1,098
(I_{read}, I_{write}) (ops/byte)	(8, 12)	(67K, 74K)	(126, 1.23M)
Average Data Movement (M.bytes)	0.19	0.33	7.01
Average LLP	2,392	151	1,365,376
Average TaLP	48	1	4

write respectively). Table 2 quantifies each workload's Gable relevant variables with each value averaged over all the tasks of a workload. TaLP and LLP values quantify (Ta)sk (L)evel and (L)oop (L)evel parallelism, respectively. The former quantifies the number of task combinations that can run in parallel, and the latter quantifies the average number of independent loop iterations. As seen, Audio uses both loop and task-level parallelism (highest TaLP among all), CAVA only uses loop-level parallelism, and Edge Detection has task level and highest LLP. Edge Detection is also the most communication-intensive benchmark. The task by task details for all workloads is in the appendix.

4.2 FARSI's Simulator Evaluation and Analysis

We validate our simulator fidelity and speedup for 250 SoC designs of different complexity levels for our AR workloads.

Overall Fidelity Results: Table 3b illustrates our simulation fidelity averaged over the said designs. We achieve high accuracy of 98.5% on average with a low standard deviation of 2.5% compared to PA. This is due to our hybrid methodology that deploys the phase concept to improve analytical models and capture system dynamics. To dissect this error further, Figure 7a details the error scalability with respect to the quantity/type of hardware blocks. Processors (PEs) and memory modules exhibit a minimal error (less than a percent) regardless of their count. We believe this error is due to a lack of modeling certain features such as "moving between clock-domains." NoCs, however, seem to impact the error the most. Increasing the number of routers in the NoC to 3 can result in as much as 5% error. Next, we investigate the cause of such high relative error.

NoC's Error Studies: We use synthetic workloads/designs to investigate the NoC's error systematically. We target software/hardware knobs that impact the NoC's behavior, collectively examining the system's communication boundedness (hardware + software knob), NoC size, hop count, hop latency (hardware knobs), and workload parallelism (software knob) impact on the error. For the hardware studies, we keep the synthetic workload simple with a fully serial task graph similar to CAVA, while for the workload parallelism studies, we examine more complex graphs. Also, unless otherwise stated, our system is simple, containing one processor, NoC, and memory module to isolate studying one variable at a time.

Communication-boundedness Impact: Since NoCs are communication blocks, we examine the impact of communicationally bounded tasks, i.e., tasks bottlenecked by NoC/memory, and observe a

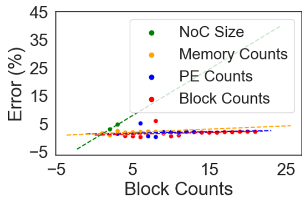
Table 3. Workload's budget and validation results.

(a) Workload's budget (5nm node).

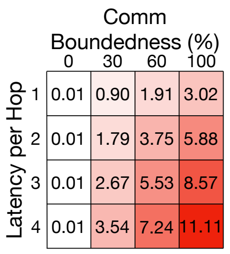
Workload	Audio	CAVA	ED
Latency (ms)	21.0	34.0	34.0
Power (mW)	8.737		
Area (mm^2)	17.475		

(b) Validation results.

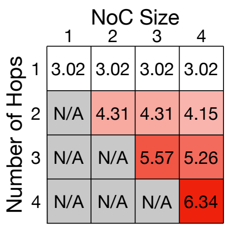
Error (%)		Speed Up	Workload Execution Latency (s)	
Avg	Std	Avg	Min	Max
1.5	2.5	8,400x	0.005	52



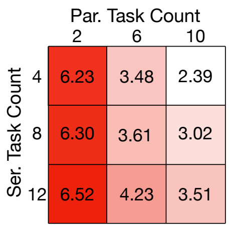
(a) Block type/count.



(b) Hop latency/Comm boundedness.



(c) Hop Count/NoC size.



(d) Parallelism.

Fig. 7. Error scalability with respect to various hardware/software knobs. All values in the heatmaps are percentage error. Comm=communicatoin, Par=parallel, Ser=serial

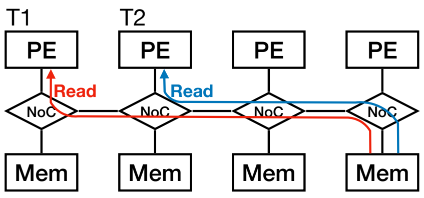
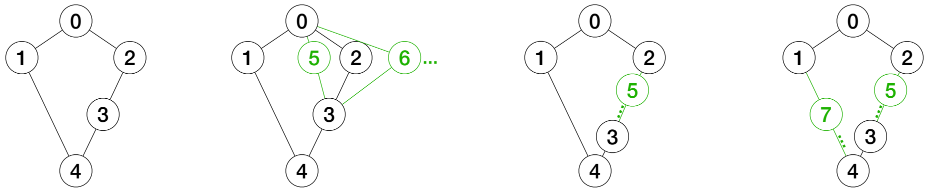


Fig. 8. System template for Hop/NoC studies. E.g., Task1 (T1) requires 4 hops while T2 needs 3.

direct relationship with error. This study uses a serial task graph similar to CAVA (Fig. 6(b)) with 7 tasks and with task's data movement mimicking "Denoise" if communicationally bounded (~500 KB) and "Scale" (~50 KB) otherwise. Initially, we force all the tasks to be computationally-bounded, i.e., to mimic Scale, and measure the simulation fidelity. Incrementally, we increase the number of tasks that are communication-bounded in the workload by swapping Scale tasks with Denoise, and measure the simulation fidelity. The heat map in Fig. 7b shows the impact on the error with the y-axis denoting the percentage of the workload's tasks that are communication-bounded. For example, the second column (indicated with 30) has a workload where 30% of the tasks are communication bounded (i.e., mimic Denoise) and 70% are computation bounded (i.e., mimic Scale). Increasing the communication boundedness from 0% to 100% (left to right) increases the error from .01% to 3.02% (first row) since more communication-bounded tasks stress the NoCs longer, thus increasing the error. Next, we study this error's root cause.

Hop Latency Impact: We examine and observe that NoC's hop latency has a direct relationship with error (Figure 7b). Hop latency is the propagation delay associated with a router within a NoC. As shown in the heatmap, increasing hop latency from 1 to 4 (top to bottom) can increase the error from 3% to 11% (last column). This is because current FARSI's NoC models (Eq. 4) do not capture this variable; thus, the error increases with an increase in variable value. Note that when communication boundedness is zero, i.e., the first column, the error is constant as the NoC is not the bottleneck to impact the error.

Hop Count and NoC size Impact: We examine the impact of Hop count and observe a direct relationship with error (Figure 7c). Hop count is the number of hops traversed by a task and depends on NoC size, number of routers in a NoC, and task mapping. For example in the NoC system of Figure 8, Task1 (T1) traverses 4 hops due to its mapping while T2 traverses 3. For Hop-NoC studies, we use a system template similar to Figure 8 while sweeping the NoC size and hop counts independently. We also use a 7 task, fully communicationally bounded workloads, to stress test the system. Increasing the number of hops from 1 to 4 (Figure 7c, top to bottom) increases the error from 3% to 6.3% since the aforementioned lack of hop latency modeling accumulates per hop. Increasing the NoC size for the same number of hops (left to right) does not increase the error. This establishes



(a) Template. (b) Audio like. (c) CAVA like. (d) ED like.

Fig. 9. Synthetic task graph templates for parallelism studies. ED= Edge Detection.

lack of hop latency modeling as the main cause of the error. Note that we plan to investigate the cause behind the small error reduction observed while increasing the NoC size.

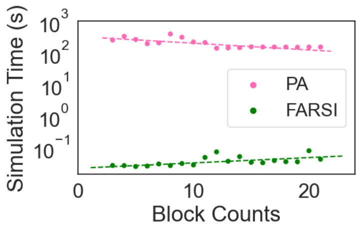
Parallelism Impact: Finally, we examine the impact of workload parallelism and observe an inverse relationship with error (Figure 7d). Template task graph of Figure 9a is used to vary parallelism while maintaining similarity with our 3 representative workloads. Concretely, tasks can be added, (1) in parallel with node 2 (Figure 9b) to achieve audio alike parallelism with a short hammock; (2) in serial with node 2 (Figure 9c) similar to high serialism of CAVA, and (3) simultaneously in serial with node 1 and 2 (Figure 9d) similar to Edge Detection parallelism with a longer hammock. We use a 4 hop scenario with fully communication-bounded tasks to stress test the system. The heat map of Figure 7d shows the results for audio style parallelism as we increase the number of parallel tasks (left to right) and CAVA style serialism as we increase the number of serial tasks (top to bottom). Increasing the number of parallel tasks and reducing the number of serial tasks reduces the error from 6.52% to 2.39. We postulate this is because more parallelism or less serialism hides NoC latency as overlapping concurrent tasks hide each other's latency at the workload level. Edge detection style parallelism shows the same trend and is left out due to brevity.

Performance Speed up/Scalability: We achieve an average simulation speedup of 8,400× over Synopsys PA (Table 3b). We owe this agility to our hybrid estimation methodology that combines analytical models and flexible phase-based simulation. To further investigate our simulator speed, we measure its sensitivity with respect to the number of blocks (system complexity), the number of tasks (workload complexity), and workload execution latency (Figure 10).

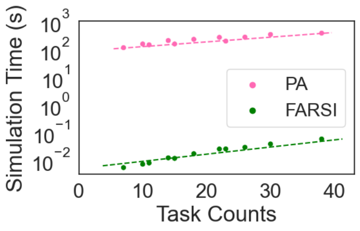
We incur a small 3× slowdown for a 20× increase in the number of blocks (Figure 10a). More blocks increase the analysis time required for bottleneck detection, thus magnifying the simulation time. For Synopsys PA, we do not see a meaningful relationship as the simulation time sometimes lowers and then plateaus with a higher number of blocks.

We observe a linear relationship between number of tasks and simulation speed (Figure 10b). For this study, we use our synthetic workloads and sweep the number of parallel (Fig. 9b) and serial (Fig. 9c) tasks. Both FARSI and PA exhibit linear scalability. For FARSI, increasing the number of tasks by 5× scales the simulation time by around 11×, from 5ms to 55ms, exhibiting around a 2× scalability. This is because FARSI's simulation time scales with the number of phases, and an extra task can increment the phase count by two, one for scheduling it in and one for scheduling it out. PA's simulation time also scales by about 4×. We did not see a simulation time difference when increasing the number of serial tasks vs. increasing the number of parallel tasks in the workload.

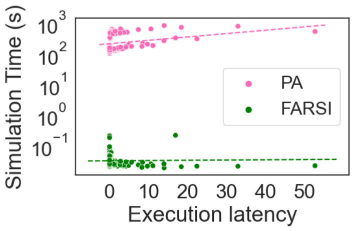
Finally, Figure 10c shows a direct relationship between simulation time and workload's execution latency. PA shows a high sensitivity to this variable where an increase of .005(s) to 50(s) of execution latency raises the simulation time from 101(s) to 814(s). FARSI also experiences a slow down of .018 (s) to .21(s) for the same increase, still maintaining high agility.



(a) Block count vs scalability.

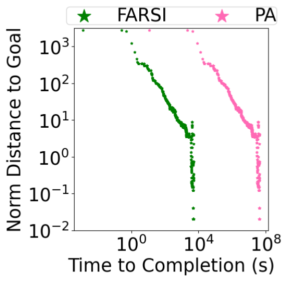


(b) Task count vs scalability.



(c) Workload latency vs scalability.

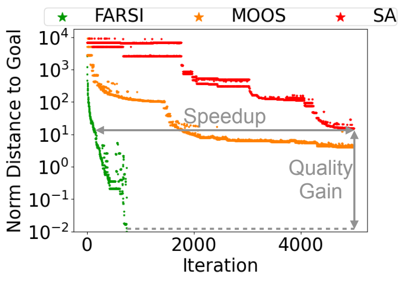
Fig. 10. Simulation time scalibility.



(a) Simulation Agility.

Metric	MOOS	FARSI
Quality Gain (%)	70	99
Speedup	1.7	62
PHV	1.10	1.12

(b) Heuristic comparisons.



(c) Heuristics' example runs.

Fig. 11. FARSI's agility. Convergence comparison to PA simulator (left, log-log scale) and heuristics example runs (right, linear-log scale). Middle table shows heuristic improvements with respect to SA.

4.3 FARSI's Exploration Heuristic Evaluation and Analysis

As FARSI manages the design space complexity by using (1) agile system-level simulation, (2) architecture awareness, (3) heavy use of co-design, and (4) domain awareness, we quantify the impact of each. We envision SoCs running all three workloads together with budgets of Table 3a.

4.3.1 Simulation Agility's Impact on Convergence: Here we quantify the impact of FARSI's agile simulator on the convergence time and fast coverage of the design space. Note that convergence time is defined as the time taken for the DSE to meet the design budgets. We compare two DSEs using the same heuristic, but one using FARSI's simulator and the other using PA. For the latter, we estimate the convergence time by increasing each iteration's time by the slow-down mentioned in Table 3b. This is because, as we will show, actually running PA takes too long and is infeasible.

Figure 11a shows the difference in the convergence time with y-axis and x-axis, showing the distance [5] to the goal or budget (Equation 7) and time to convergence, respectively. FARSI takes 3 hours to converge by exploring more than 750 designs, and PA takes more than 3 years. Note that by the time FARSI has converged, PA has only looked at 4 designs. FARSI's simulation agility makes it an ideal candidate for a DSSoC DSE as it enables high coverage of the complex/big design spaces. Authors grant that FARSI's simulation error can result in suboptimal design decisions, thus degrading the converged design's quality. However, resorting to more accurate/low agility simulators like PA is infeasible in large design spaces. We suspect that a hybrid methodology that uses FARSI to aggressively prune the space followed by PA examining certain neighborhoods with higher accuracy can provide an alternative solution. We leave the investigation of this thread to future work.

4.3.2 Architecture Awareness: Given the number of knobs across topology generation, allocation, and mapping, the design space is too great to be navigated blindly. For example, a system with 6 tasks and 2 knobs per processor/memory/NoC results in more than a million design points, and

[5]Normalized city block distance to budget across all metrics.

our AR complex with more than 28 tasks results in a space greater than the number of stars in the universe. Thus, optimal navigation demands architectural insights for guidance.

To highlight the importance of architecture-awareness, we compare FARSI against the classic simulated annealing (SA) and more modern MOOS. The former samples the space at random and greedily selects the best neighbor, while the latter's data-driven approach continuously learns to adjust the search starting point. We evaluate FARSI's efficiency across 3 metrics, i.e., quality gain, speedup, and Pareto Hyper Volume (PHV). We use SA as the baseline for all metrics, thus normalizing w.r.t SA; Furthermore, since all heuristics are sampling-based and thus exhibit stochastic behavior, we collect and average the data over 15 runs per heuristic.

Quality Gain: For a design's quality, we use its city block distance to the budget (Equation 7) as a proxy. Thus, the quality gain is the difference between a heuristic's quality of the best solution and the baseline's counterpart, normalized to the baseline ($\frac{d-d_{baseline}}{d_{baseline}}$, with d denoting city distance).

Speedup: The number of iterations to convergence for baseline (SA) over the corresponding value of FARSI or MOOS. The iteration count is measured for the best solution uncovered by the baseline.

PHV: Pareto Hyper-Volume (PHV) [92], i.e., the size of the objective space dominated by the Pareto set solutions, is often used [16, 42, 45] to evaluate the Pareto front's quality. We normalized this metric to the baseline's PHV ($\frac{PHV}{PHV_{baseline}}$). Bigger values are better.

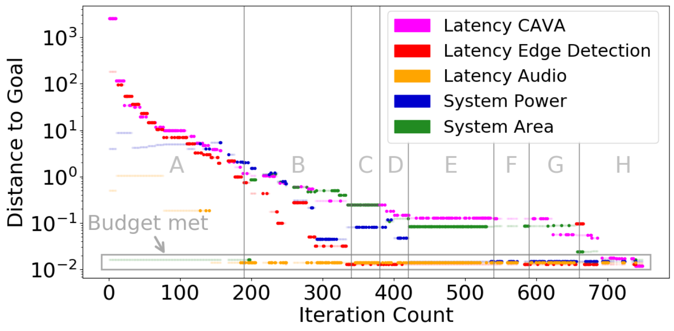
Figure 11b quantifies the heuristics efficiencies using the said metrics averaged over all runs, and Figure 11c provides an example run for each heuristic. For Quality Gain, MOOS exhibits a 70% improvement over SA (Fig. 11b) as its data-driven model learns from past search iterations, penetrates the space deeper, and thus uncovers higher quality solutions. However, as seen in the example runs of Figure 11c, MOOS, similar to SA does not zero the distance to budget and cannot meet AR's tight constraints. In contrast, FARSI uses its architectural awareness (e.g., use of locality and parallelism) to improve the design quality even further by 99%, and meet the budget,[6] thus making it more ideal for DSSoCs with tight constraints.

MOOS shows a 1.7X speedup over SA (Fig. 11b). It prunes the space faster than SA by continuously selecting efficient starting points for its greedy search. As shown in the example run, this approach provides an overall more steady reduction in distance and thus faster convergence. In contrast, SA often gets stuck in local optimum and relies on random sampling to be pulled out. FARSI achieves a much higher speed up, up to 62X. Its exploitation of architectural insights in neighbor generation exposes it to more high potential neighbors, which accelerates the convergence.

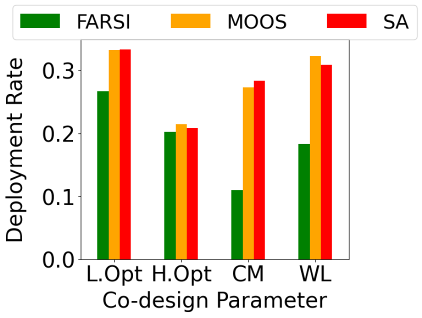
Finally, for PHV, MOOS and FARSI generate higher quality Pareto fronts over SA with bigger normalized PHVs, 1.10 and 1.12, respectively. Note that FARSI's improvement is not much bigger than MOOS because it prioritizes exploitation over exploration. Concretely, FARSI's reliance on architectural knowledge (instead of random sampling) for finding improved neighbors lowers its exploration degree. This, in turn, suppresses its capability to expand the Pareto front much beyond MOOS, specifically for neighborhoods farther away from the targeted budget, consequently observing an incremental improvement. Please note that we have not provided a plot of our heuristics' Pareto fronts since they are 5 dimensional (with the axis of the area, power, and 3 different performance for each workload) and thus can not be visualized.

4.3.3 Co-design: A DSE without a co-design cannot exploit cross-boundary optimization opportunities necessary for convergence. For example, without a cross-workload co-design, a DSE misses on area reduction enabled by inter workload memory sharing. FARSI uses co-design by not being fixated on one optimization for too long and rather continuously moving from one optimization to another. Our co-design occurs within each of the following vectors: (1) across

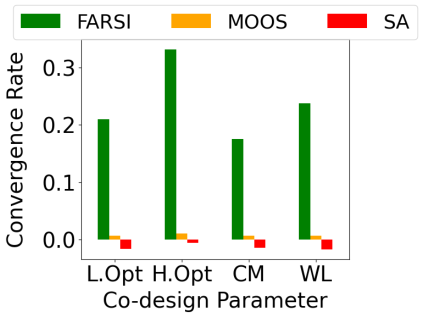
[6]When the budget is met (distance=0) in the plot, we set it to 10^{-2} (as 0 cannot be shown with log).



(a) Co-design over time across metrics and workloads.



(b) Co-design deployment rate.



(c) Co-design resulted convergence rate.

Fig. 12. Co-design impact and rate. H.Opt=high-level optimization, L.Opt=low-level optimization, CM=communication/computation, WL=workload.

metrics (i.e., among power, performance, area), (2) across high-level optimization (i.e., among mapping/allocation/customization), and across low-level optimizations (frequency modulation, bandwidth modulation, hardening, etc.), (3) across computation/communication, and (4) across workloads. Here we quantify their impact.

Figure 12a as an example shows FARSI's co-design progression across two of these vectors, i.e., metrics and workloads. X-axis and y-axis show iteration count and each metric's distance to the goal (log-scaled), respectively. To emphasize when FARSI targets a metric/workload, we make the color of its curve bold; otherwise, make it see-through. As shown in the figure, the exploration goes through various zones (A through H), each with a different exploration focus. For example, zone A mainly targets latency and exploits cross-workload opportunities between Edge Detection and CAVA. Zone C shifts its focus toward system power and area co-design. Note that Audio latency is also targeted (although already met) as it can be increased to help with power/area. Finally, FARSI splits its attention between latency (of Audio and CAVA) and system area in zone E. Note that different metrics meet their budget (reach 10^{-2}) throughout the exploration at different times, but a continuous trade-off between them is exploited until all distances are zero. Such co-design exploitation is a significant contributor to the fast convergence previously shown in Figure 11c. Note that Figure 13 shows the normalized iteration breakdown of our other co-design vector, namely high- and low-level optimizations. However, their co-design progression plots are left out due to space limitations.

Figure 12b shows how often (deployment rate) co-design occurs within each of our vectors. For example, a co-design rate of .2 for workloads means that FARSI changed its focus from one workload to another 2 times every 10 iterations on average. Note that these rates are not tuned by the tool user but determined dynamically by FARSI as it explores the space. For example, if FARSI sees that one workload demands more attention (i.e., its distance to the PPA budget is high), it will select and focus on it long before switching to others. FARSI applies co-design across both high-level (topology generation, allocation, and software to hardware mapping) and low-level (frequency modulation, memory allocation, etc.) optimization options the most. Concretely, it changes its focus between these options around every 4-5 iterations. The workload co-design rate is lower, indicating a difference in each workload's convergence difficulty. Finally, FARSI applies even a longer attention length (>10 iterations) for computation and communication, indicating their higher imbalance. Note that both SA and MOOS deploy a higher co-design rate for all vectors as they randomly generate neighbors; however, next, we show that this strategy is not optimal.

To quantify the impact of the said co-design approaches, we measure the average convergence rate (i.e., an average of improved distance per iteration) resulted from each. Figure 12c reveals that computation/communication co-design leads to the highest improvement, i.e., an average convergence rate of 35%. Overall on average, co-design improves the distance by 32%. Note that the MOOS convergence rate due to co-design is very small, as co-design is not systematically embedded in the heuristic, and the search rather stumbles on it from iteration to iteration. SA's convergence rate due to co-design is even worse, in fact, negative, as a complete random switching between, for example, workloads can hurt the distance. This shows that a principled co-design is necessary.

4.3.4 Domain Awareness: DSEs that target domain-specific SoCs need to be domain aware, i.e., 1) extract workloads' characteristics and then 2) direct their optimizations to exploit workload's inherent opportunities (e.g., TaLP). Here, we first detail our workload's characteristics and then showcase FARSI's awareness of them. We run each workload individually and lower their latency to their limits to stress test FARSI toward exploring all possible optimization opportunities, e.g., Task Level Parallelism (TaLP) and Loop Level Parallelism (LLP).

Figure 14 (table on the right) sheds light on each workload's computation and communication characteristics, concretely, parallelism (loop- and task-level), and data movement. Edge Detection (ED) has the highest LLP and data movement, and Audio has the highest TaLP. Figure 14 (left) further details the computation/communication boundedness of each workload. The y-axis shows the (normalized) breakdown of the hardware bottlenecks FARSI encounters during exploration. This information guides the DSE in selecting their target stage (i.e., communication or computation). CAVA is the most computation bounded while the other two show more balanced boundedness.

Figure 15 illustrates FARSI's response to the above characteristics. Concretely, Figure 15a shows FARSI's uses of parallelism (in the number of iterations). TaLP denotes the number of iterations

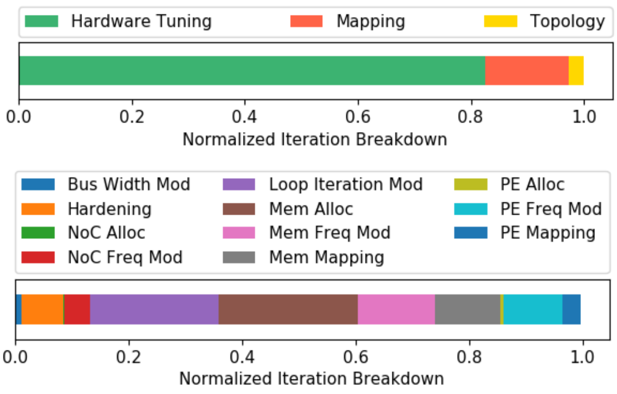
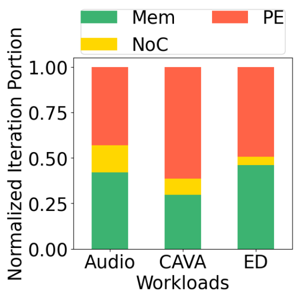
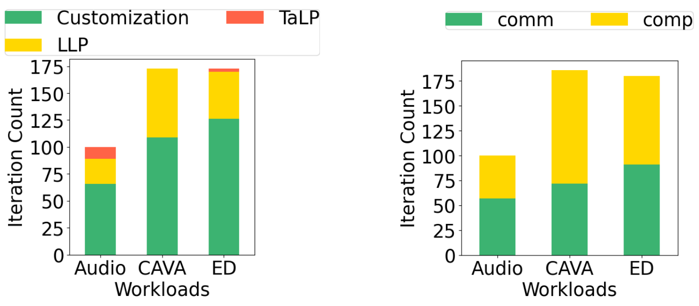


Fig. 13. FARSI's use of high- (top) and low-level (bottom) optimizations. Alloc = Allocation, Mod = Modulation, Freq = Frequency, PE = Processing Element, Mem = Memory.



Work-load	Audio	CAVA	ED
Data Movement (MB)	0.190	0.325	7.000
Average LLP	2,392	151	1,365,376
Average TaLP	48	1	4

Fig. 14. Communication(comm)/computation(comp) boundedness (left) and Task (TaLP) and loop-level (LLP) parallelism (right), of each workload. ED shorten for Edge Detection.



(a) Parallelism sensitivity.

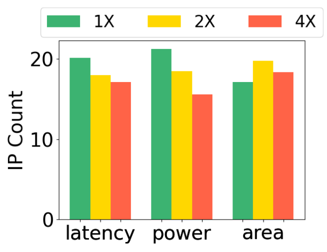
(b) Communication(comm) / computation(comp) boundedness sensitivity.

Fig. 15. FARSI's domain awareness. FARSI exploits workload characteristics such as TaLP, LLP and computation/communication boundedness for convergence.

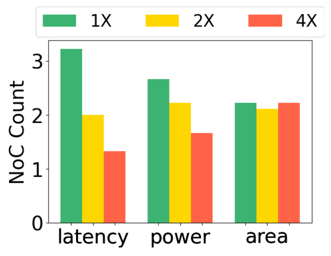
that task-level parallelism is exploited to improve the design either through a new topology or software to hardware mapping. LLP denotes iterations where loop unfolding has been exploited, and customization denotes hardware customizations such as frequency tuning, bus width tuning, etc. FARSI applies task-level parallelism on Audio the most and CAVA the least, as Audio's TDG provides the highest TaLP opportunities and CAVA provides none. FARSI targets loop-level parallelism for Edge Detection more than Audio correlated with their LLP. Note that FARSI's excessive use of LLP for CAVA is inevitable as it has no other parallelism option. Moreover, Figure 15b shows FARSI's relative focus on computation and communication as a response to these bottlenecks. CAVA has a high computation boundedness tendency, and thus, FARSI targets its computation the most while a more balanced approach is used for the other two according to their needs. FARSI's focus on communication is correlated with each workload's data movement, with Edge Detection having the highest (7 MB on average) and Audio having the least (0.18 MB on average) data movement.

5 CASE STUDIES

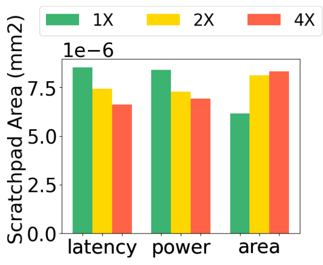
To show FARSI's capability with the design challenges of complex SoCs, we provide two case studies. First, we show how FARSI's cost-aware methodology balances "system complexity" (and thus the development effort) with the product quality. Then, we shed light on the inefficiencies of the "divide and conquer" approach often used to tame the complexity and discuss FARSI-enabled optimal designs. For both case studies, we assume a system that runs all three workloads simultaneously.



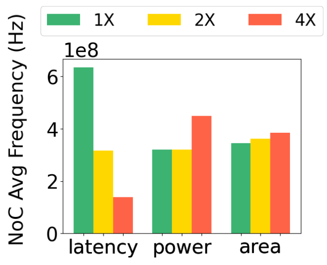
(a) IP count.



(b) Number of NoCs.



(c) Scratchpad memory size.



(d) NoC average frequency.

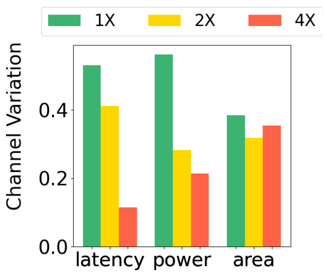
Fig. 16. System implications for different budgets. FARSI lowers block count and thus lowers design complexity when the latency and power budgets are relaxed by 2× and 4×. FARSI is also careful about keep hardware knobs low such as memory size/frequency when possible.

5.1 Budget Implications on Development Effort

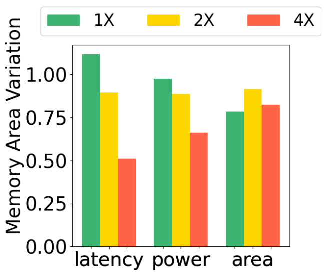
Since a DSSoC requires a high-development effort, an optimal DSE framework must use various trade-offs to lower this design effort. An example of such a trade-off is the product quality and development effort. Concretely, an optimal DSE must lower the development effort whenever a product quality is relaxed. This study showcases FARSI's capability in this balancing act. To this end, we proxy the product quality with performance/power/area (PPA) budgets. This is because for example, in AR, performance budget, e.g., frame-rate, directly impacts user experience as it determines how smooth user-world interactions are; power budget impacts battery life and thus determines whether AR glasses can be reasonably operational; and finally area impacts the die cost and thus the overall product cost. We also proxy development effort by the number and the variation (heterogeneity) of hardware blocks as increasing either magnifies the development effort. We relax Section 4.3 PPA budgets (Table 3a) with 1×, 2×, and 4×, run FARSI for each budget showing how it lowers hardware block counts and variations when faced with larger budgets (e.g., 4× budget).

Component Implications: Figures 16a and 16b show the impact of different budgets on IP and Network-on-a-Chip (NoC) subsystems. Increasing the latency budget from 1× to 4× reduces the IP and NoC counts (NoCs belonging to different subsystems, e.g., audio subsystem, or image processing subsystem) by 15% and 60% respectively, which implies that FARSI lowers the block count when system budgets are relaxed. In addition, this indicates that NoC design/integration should be prioritized in delivering performance, as the performance has a higher sensitivity to NoC than IP count. We also observe that a power budget increase of 1× to 4× results in a 26% and 37% reduction in IP and NoC count. This means that IP's impact on power reduction is higher than performance. However, the reverse is true for NoC's. We do not observe a meaningful relationship with the area.

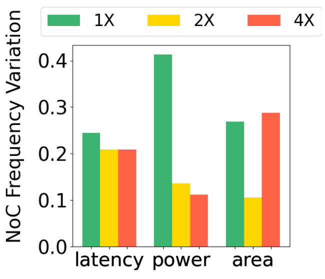
Figure 16c shows memory size sensitivity to the budgets. Increasing the latency and power budget from 1× to 4× results in a total on-chip scratchpad memory size reduction of 22% and 17%,



(a) NoC's link count.



(b) Scratchpad size.



(c) NoC frequency.

Fig. 17. System heterogeneity. FARSI lowers the block heterogeneity and thus design complexity when the latency and power budgets are relaxed by 2× and 4×.

respectively. This is because the relaxed budget can tolerate higher global data movement latency and energy of DRAM. However, for the same budget increase in area, the (low-density) SRAM area can be freely increased by 25% to move the data back locally. System designers can use FARSI and conduct such studies to find optimal memory allocations across SRAM and DRAM.

System designers can also use FARSI to investigate the impact of budget on NoC Frequency. Figure 16d details this sensitivity. Concretely, we observe that as the latency budget is relaxed to 4×, NoC frequency is scaled down by 78%. This is because lower performance budgets do not require high-frequency operating NoCs. In contrast, relaxing the power budget by 4× instead scales up the frequency by 70% since for higher power budgets, the system can tolerate higher frequency NoCs. Overall, we show that FARSI exploits system-budget trade-offs and tunes the design accordingly.

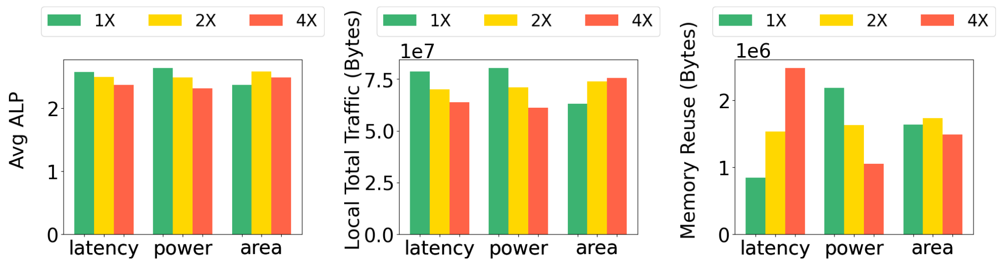
System Heterogeneity: System heterogeneity (block variations) is also a determinant of the design complexity/development effort. To quantify a system variable's heterogeneity (e.g., scratch pad size variation), we use the *coefficient of variation* ($CV = \frac{standard\ deviation}{mean}$) [59] which captures the variable's deviation from its mean. Higher CV means higher variation; for example, in Figure 17b, the height of leftmost bar, i.e., 1.1, indicates a 110% mean variation of the system's scratchpad sizes.

Figure 17a shows that increasing the performance or power budgets by 4× lead to 82% and 60% reduction in the link (channel) count heterogeneity among NoCs i.e. lower variation in the number of links across NoCs. This lowers the design effort as variation in NoC design increases the design complexity. We also observe that higher NoC customization is necessary for the power budget compared to performance budget variations. This is because the lowest and the highest power budgets (1× and 4×) tend to cause more heterogeneity than their corresponding latency scaling values.

A similar trend can be observed in both memory size (Figure 17b) and NoC frequency heterogeneity (Figure 17c). The former experiences a 54% and 32% variation reduction as a result of relaxing the system performance and power budget. The latter experiences a 15% and 73% reduction for relaxation of the same metrics. This renders memory variation more critical for latency delivery while bus frequency variation more critical for power. Note that memory size and NoC frequency variations impact complexity as the former increases the optimization or customization effort, and the latter increases the clock tree and PLL complexity.

Overall, we see that FARSI lowers system heterogeneity in response to budget relaxation. In addition, system designers can use FARSI to investigate the product quality impact on system heterogeneity and make various product decisions while keeping the development effort in mind.

System Dynamics: We provide designers with system dynamics analysis to guide the design planning. Figure 18a quantifies the accelerator level parallelism, i.e., the average number of accelerators that run in parallel [35]. As shown, an increase of 8% and 13% are needed to meet the tighter 1× performance and power budgets compared to 4×. Such parallelism increases the local traffic by



(a) Accelerator-level parallelism. (b) Scratchpad memory traffic. (c) Scratchpad memory reuse.

Fig. 18. System dynamics. FARSI guides decision making by profiling various system dynamics.

23% and 31% (Figure 18b) and demands a 107% increase in memory reuse (how many times a memory module is re-accessed, measured in bytes) to keep the memory size small, specifically when power efficiency is required (Figure 18c). However, to deliver for performance, FARSI requires lower memory reuse of 65% to prevent memory contention. Note that the reuse's opposite direction between performance and power indicates the importance of memory mapping and its delicate balancing act. Such optimizations can only be achieved using an agile (rather than manual) methodology with a holistic system-level lens to explore sufficient scenarios.

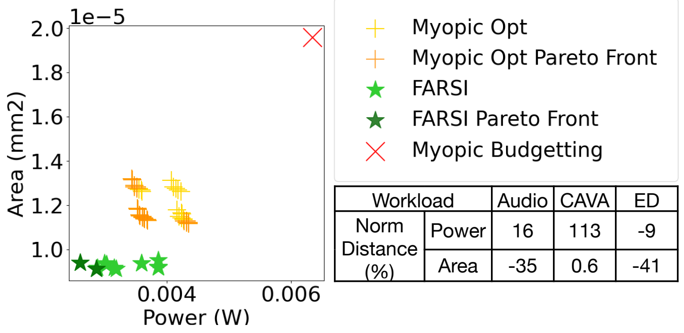
5.2 Divide and Conquer Suboptimality

Lack of access to holistic design methodologies forces designers to tame the complexity by taking a divide and conquer approach. This means splitting the system into subsystems (one for each workload), imposing ad hoc (estimated) power and area budgets on each, and finally using best efforts to reach them in isolation. This is common in domains that consist of a diverse set of sub-domains, such as AR (video, audio, graphics, ...); however, our results show that it leads to sub-optimal designs due to myopic budget estimations (Problem 1) and optimizations (Problem 2).

Problem 1, Myopic Budget Estimation: In the absence of automated DSEs, each workload's power/area budget is decided manually using architects' insights and back-of-the-envelope estimates. If the estimates for a workload are too tight, the entire chip budget needs to be expanded to incorporate the workload's best design. However, if a workload's budgets are too loose, optimization opportunities targeting a tighter budget are left unexploited. (Note that the extra budget can then be redistributed to the other workloads in need). We emulate this problem by setting the power budgets as per isolated power estimates provided in [37] while for the area, we use power as a proxy and budget each workload's area according to its relative power. Further details are provided in the Appendix (Section C). Latency values are set similar to previous sections to ensure a high-quality user experience. FARSI finds a design that meets this pre-determined budget for each workload.

Problem 2, Myopic Optimizations in Isolation: Focusing on each workload in isolation misses out on the cross-workload optimization opportunities such as memory sharing. To isolate this issue and relieve the said budgeting problem (problem 1), for each workload, we sweep all budget values from 0 to the SoC budget with increments of 5%. For each workload in isolation, we run FARSI with the mentioned budget sweep and generate a power/area Pareto front. Then, final SoCs (running all workloads) are put together by combining the designs' permutations of the workload's Pareto Fronts.

System Degradation Associated with Problem 1 and 2: Here we quantify the degradation by comparing the methodology used in Problems 1 and 2 with full-fledged FARSI. Note that at the high level, full-fledged FARSI automatically solves problem 1 as it does not require individual workload budgets. Instead, it finds the optimal sub-budgeting as it explores the space. Furthermore, FARSI circumvents Problem 2 by conducting cross-workload optimizations.



Workload		Audio	CAVA	ED
Norm Distance (%)	Power	16	113	-9
	Area	-35	0.6	-41

Fig. 19. Divide and conquer sub-optimality. Myopic optimization/budgeting (left) and details (right).

Figure 19 illustrates the suboptimality of both approaches, with the x-axis and y-axis denoting the power and area associated with the designs generated for each methodology. Myopic Budgeting predictably does the worst with its point (far top right) experiencing an average power and area degradation of 56% and 52% respectively, compared to FARSI's Pareto Front points. The table investigates this issue by presenting the distance between Myopic Budgetting's best design and its pre-determined budget (normalized and multiplied by 100). Negative values mean the design never met the budget, and thus, the budget was too tight and vice versa. As shown, the power/area budgets for Edge Detection and the area for Audio were too tight. However, CAVA's area budget was optimal, and the power budget was set too loose and should have been distributed across the other two workloads.

Myopic Optimization (Myopic Opt) provides suboptimal solutions as well. Both its Pareto Front and all generated designs are less optimal compared to full-fledged FARSI (Figure 19). Concretely, points on the Pareto Front, on average, experience a 27% and 21% power/area degradation compared to FARSI. Myopic optimization cannot use cross-workload mapping solutions to share memory and save area or keep the congestion low and improve performance without improving frequency.

6 CONCLUSION AND FUTURE WORK

This work presents FARSI, a DSSoC DSE equipped with an agile system simulator and an automated heuristic with built-in and expandable architectural reasoning. We identify critical ingredients of an optimal DSSoC DSE and quantify their impact on the convergence time, and further show FARSI's heavy use of them. We achieve 8,400×, and 98.5% simulation speed up and accuracy compared to Platform Architect. We also achieve 62× and 35× convergence speed up exploiting architectural reasoning and co-design compared to simulated annealing and MOOS, respectively. We further present two case studies show-casing FARSI's importance in the design challenges of future complex systems. Going forward, we plan to apply FARSI to other domains such as autonomous vehicles and robotics and further investigate multi-SoC designs where the computation is distributed across multiple chips, for example, in a distributed chip network within a car.

REFERENCES

[1] Nikon d7000. https://en.wikipedia.org/wiki/Nikon_D7000, 2021.
[2] Rasmus Abildgren, Jean-Philippe Diguet, Pierre Bomel, Guy Gogniat, Peter Koch, and Yannick Le Moullec. A priori implementation effort estimation for hardware design based on independent path analysis. *EURASIP Journal on Embedded Systems*, 2008, 12 2008.
[3] A. Agarwal and Ramamurti Shankar. Cost feasibility analysis for embedded system development and the impact of various methodologies on product development cycle. 2008.

[4] Muhammad Shoaib Bin Altaf and David A. Wood. Logca: A high-level performance model for hardware accelerators. *SIGARCH Comput. Archit. News*, 45(2):375–388, June 2017.

[5] Samet E. Arda, Anish Krishnakumar, A. Alper Goksoy, Nirmal Kumbhare, Joshua Mack, Anderson Luiz Sartor, Ali Akoglu, Radu Marculescu, and Umit Y. Ogras. Ds3: A system-level domain-specific system-on-chip simulation framework. *IEEE Transactions on Computers*, 69:1248–1262, 2020.

[6] Giuseppe Ascia, Vincenzo Catania, Alessandro G Di Nuovo, Maurizio Palesi, and Davide Patti. Efficient design space exploration for application specific systems-on-a-chip. *Journal of Systems Architecture*, 53(10):733–750, 2007.

[7] Mario Badr and Natalie Enright Jerger. Synfull: Synthetic traffic models capturing cache coherent behaviour. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 109–120, 2014.

[8] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, PPoPP '12, page 11–22, New York, NY, USA, 2009.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[10] Lukai Cai and Daniel Gajski. Transaction level modeling: An overview. pages 19– 24, 11 2003.

[11] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. *SIGOPS Oper. Syst. Rev.*, 35(5):103–116, oct 2001.

[12] T. Chen, A. Rucker, and G. E. Suh. Execution time prediction for energy-efficient hardware accelerators. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 457–469, 2015.

[13] Weilong Cui and Timothy Sherwood. Estimating and understanding architectural risk. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 651–664, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Piotr Czyzżak and Adrezej Jaszkiewicz. Pareto simulated annealing—a metaheuristic technique for multiple-objective combinatorial optimization. *Journal of Multi-criteria Decision Analysis*, 7:34–47, 1998.

[15] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. *SIGARCH Comput. Archit. News*, 41(1):381–394, March 2013.

[16] Aryan Deshwal, Nitthilan Kanappan Jayakodi, Biresh Kumar Joardar, Janardhan Rao Doppa, and Partha Pratim Pande. Moos: A multi-objective design space exploration and optimization framework for noc enabled manycore systems. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019.

[17] R.P. Dick and N.K. Jha. Mogac: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, (10), 1998.

[18] Xiangyu Dong, Jishen Zhao, and Yuan Xie. Fabrication cost analysis and cost-aware design space exploration for 3-d ics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29:1959–1972, 2010.

[19] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alex Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems*, 2, 07 1997.

[20] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Des. Autom. Embedded Syst.*, 2(1):5–32, jan 1997.

[21] C. Erbas, S. Cerav-Erbas, and A.D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Transactions on Evolutionary Computation*, 10(3):358–374, 2006.

[22] S. Eyerman, L. Eeckhout, and K. De Bosschere. Efficient design space exploration of high performance embedded out-of-order processors. In *Proceedings of the Design Automation Test in Europe Conference*, volume 1, 2006.

[23] W. Fornaciari, F. Salice, U. Bondl, and E. Magini. Development cost and size estimation starting from high-level specifications. In *Ninth International Symposium on Hardware/Software Codesign. CODES 2001 (IEEE Cat. No.01TH8571)*, pages 86–91, 2001.

[24] W. Fornaciari, F. Salice, U. Bondl, and E. Magini. Development cost and size estimation starting from high-level specifications. In *Ninth International Symposium on Hardware/Software Codesign. CODES 2001 (IEEE Cat. No.01TH8571)*, pages 86–91, 2001.

[25] Arpad Gellert, Adrian Florea, Ugo Fiore, Paolo Zanetti, and Lucian Vintan. Performance and energy optimisation in cpus through fuzzy knowledge representation. *Information Sciences*, 476, 03 2018.

[26] A. Gerstlauer, J. Peng, D. Shin, D. Gajski, A. Nakamura, D. Araki, and Y. Nishihara. Specify-explore-refine (ser): From specification to implementation. In *2008 45th ACM/IEEE Design Automation Conference*, pages 586–591, 2008.

[27] Fred Glover and Manuel Laguna. *Tabu search I*, volume 1. 01 1999.

[28] Godot. Material testers. https://github.com/godotengine/godot-demo-projects/tree/master/3d/material_testers, 2020.

[29] Godot. Platformer 3D. https://github.com/godotengine/godot-demo-projects/tree/master/3d/platformer, 2020.

[30] S. Graf, M. Glaß, J. Teich, and C. Lauer. Multi-variant-based design space exploration for automotive embedded systems. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2014.

[31] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. Navigating heterogeneous processors with market mechanisms. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 95–106, 2013.

[32] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. Strategies for anticipating risk in heterogeneous system design. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 154–164, 2014.

[33] Soonhoi Ha and Jürgen Teich, editors. *Handbook of Hardware/Software Codesign*. Springer, 2017.

[34] Mark Hill and Vijay Janapa Reddi. Gables: A roofline model for mobile socs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 317–330, 2019.

[35] Mark D. Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *CoRR*, abs/1907.02064, 2019.

[36] Hongsik Lee, Dong Nguyen, and J. Lee. Optimizing stream program performance on cgra-based systems? In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[37] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, Boyuan Tian, Hengzhi Yuan, Jeffrey Zhang, and Sarita V. Adve. Illixr: Enabling end-to-end extended reality research. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 24–38, 2021.

[38] Hanhwi Jang, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. Rpstacks-mt: A high-throughput design evaluation methodology for multi-core processors. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 586–599. IEEE Press, 2018.

[39] Z. J. Jia, A. D. Pimentel, M. Thompson, T. Bautista, and A. Núñez. Nasa: A generic infrastructure for system-level mp-soc design space exploration. In *2010 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia*, pages 41–50, 2010.

[40] Zai Jian Jia, Antonio Núñez, Tomás Bautista, and Andy D Pimentel. A two-phase design space exploration strategy for system-level real-time application mapping onto mpsoc. *Microprocessors and Microsystems*, 38(1):9–21, 2014.

[41] Yiming Jing, Jishun Kuang, Jiayi Du, and Biao Hu. Application of improved simulated annealing optimization algorithms in hardware/software partitioning of the reconfigurable system-on-chip. In Kenli Li, Zheng Xiao, Yan Wang, Jiayi Du, and Keqin Li, editors, *Parallel Computational Fluid Dynamics*, pages 532–540, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[42] Biresh Kumar Joardar, Ryan Gary Kim, Janardhan Rao Doppa, Partha Pratim Pande, Diana Marculescu, and Radu Marculescu. Learning-based application-agnostic 3d noc design for heterogeneous manycore systems. *IEEE Transactions on Computers*, 68(6):852–866, 2019.

[43] Prathmesh Kallurkar and Smruti Sarangi. Schedtask: a hardware-assisted task scheduler. pages 612–624, 10 2017.

[44] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[45] Ryan Gary Kim, Janardhan Rao Doppa, and Partha Pratim Pande. Machine learning for design space exploration and optimization of manycore systems. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2018.

[46] Y. G. Kim and C. J. Wu. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[47] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. HPVM: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–80, 2018.

[48] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. The sky is not the limit: A visual performance model for cyber-physical co-design in autonomous machines. *IEEE Computer Architecture Letters*, 19(1):38–42, 2020.

[49] Liang-Yu Lin, Cheng-Yeh Wang, Pao-Jui Huang, Chih-Chieh Chou, and Jing-Yang Jou. Communication-driven task binding for multiprocessor with latency insensitive network-on-chip. In *Proceedings of the ASP-DAC 2005. Asia and South Pacific Design Automation Conference, 2005.*, volume 1, pages 39–44 Vol. 1, 2005.

[50] Zhonghai Lu, Lei Xia, and Axel Jantsch. Cluster-based simulated annealing for mapping cores onto 2d mesh networks on chip. In *2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pages 1–6, 2008.

[51] M. Lukasiewycz, M. Streubuhr, M. Glass, C. Haubelt, and J. Teich. Combined system synthesis and communication architecture exploration for mpsocs. In *2009 Design, Automation Test in Europe Conference Exhibition*, 2009.

[52] Martin Lukasiewycz, Michael Glass, Christian Haubelt, and Jurgen Teich. Efficient symbolic multi-objective design space exploration. In *2008 Asia and South Pacific Design Automation Conference*, pages 691–696, 2008.

[53] Joshua Mack, Sahil Hassan, Nirmal Kumbhare, Miguel Castro Gonzalez, and Ali Akoglu. Cedr - a compiler-integrated, extensible dssoc runtime. *ACM Trans. Embed. Comput. Syst.*, mar 2022. Just Accepted.

[54] Sumit K. Mandal, Raid Ayoub, Michael Kishinevsky, and Umit Y. Ogras. Analytical performance models for nocs with multiple priority traffic classes. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019.

[55] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[56] Monado. Sponza scene in Godot with OpenXR addon. https://gitlab.freedesktop.org/monado/demos/godot-sponza-openxr, 2019.

[57] N/A. perf. https://perf.wiki.kernel.org/index.php/Main_Page, 2020.

[58] N/A. Hp labs: Cacti. https://www.hpl.hp.com/research/cacti/, 2021.

[59] N/A. Variation coefficient wikipedia. https://en.wikipedia.org/wiki/Coefficient_of_variation, 2021.

[60] Ralf Niemann and Peter Marwedel. An algorithm for hardware/software partitioning using mixed integer linear programming. In *Proceedings of the EDTC*, pages 165–193. Kluwer Academic Publishers, 1996.

[61] Heikki Orsila, Tero Kangas, Erno Salminen, Timo D. Hämäläinen, and Marko Hännikäinen. Automated memory-aware application distribution for multi-processor system-on-chips. *J. Syst. Archit.*, 53(11):795–815, November 2007.

[62] Saptadeep Pal, Daniel Petrisko, Rakesh Kumar, and Puneet Gupta. Design space exploration for chiplet-assembly-based processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(4):1062–1073, 2020.

[63] G. Palermo, C. Silvano, and V. Zaccaria. Discrete particle swarm optimization for multi-objective design space exploration. In *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 641–644, 2008.

[64] M. Palesi and T. Givargis. Multi-objective design space exploration using genetic algorithms. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, pages 67–72, 2002.

[65] Chan Park, Sungkyung Park, and Chester Sungchung Park. Roofline-model-based design space exploration for dataflow techniques of cnn accelerators. *IEEE Access*, 8:172509–172523, 2020.

[66] Platform Architect. https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html, 2021.

[67] R. Pellizzoni, A. Schranzhofer, Jian-Jia Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 741–746, 2010.

[68] A.D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[69] Raghavendra Pradyumna Pothukuchi, Joseph L. Greathouse, Karthik Rao, Christopher Erb, Leonardo Piga, Petros G. Voulgaris, and Josep Torrellas. Tangram: Integrated control of heterogeneous computers. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 384–398, New York, NY, USA, 2019. Association for Computing Machinery.

[70] Wei Quan and Andy D. Pimentel. Towards exploring vast mpsoc mapping design spaces using a bias-elitist evolutionary approach. In *2014 17th Euromicro Conference on Digital System Design*, pages 655–658, 2014.

[71] Wei Quan and Andy D. Pimentel. A hybrid task mapping algorithm for heterogeneous mpsocs. *ACM Trans. Embed. Comput. Syst.*, 14(1), jan 2015.

[72] Víctor Reyes, Tomás Bautista, Gustavo Marrero, Pedro P Carballo, and Wido Kruijtzer. Casse: a system-level modeling and design-space exploration tool for multiprocessor systems-on-chip. In *Euromicro Symposium on Digital System Design, 2004. DSD 2004.*, pages 476–483. IEEE, 2004.

[73] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi. gem5-salam: A system architecture for llvm-based accelerator modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 471–482, 2020.

[74] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Wei, and D. Brooks. Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[75] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108, 2014.

[76] Yakun Sophia Shao and Yu Emma Wang. Die photo analysis. http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/, 2015.

[77] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. *SIGPLAN Not.*, 47(8):11–22, February 2012.

[78] Kyle L. Spafford and Jeffrey S. Vetter. Automated design space exploration with aspen. *Sci. Program.*, 2015, jan 2015.

[79] Stereolabs. ZED Mini - Mixed-Reality Camera. https://www.stereolabs.com/zed-mini/.

[80] Dean Takahashi. Oculus chief scientist mike abrash still sees the rosy future through ar/vr glasses. https://venturebeat.com/2018/09/26/oculus-chief-scientist-mike-abrash-still-sees-the-rosy-future-through-ar-vr-glasses/, September 2018.

[81] Radu Teodorescu and Josep Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *2008 International Symposium on Computer Architecture*, pages 363–374, 2008.

[82] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia mp-socs. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 9–14, 2007.

[83] Mark Thompson and Andy Pimentel. Exploiting domain knowledge in system-level mpsoc design space exploration. *Journal of Systems Architecture: the EUROMICRO Journal*, 59:351–360, 08 2013.

[84] Augusto J. Vega, Aporva Amarnath, John-David Wellman, Hiwot Tadese Kassa, Subhankar Pal, Hubertus Franke, Alper Buyuktosunoglu, Ronald G. Dreslinski, and Pradip Bose. Stomp: A tool for evaluation of scheduling policies in heterogeneous multi-processors. *ArXiv*, abs/2007.14371, 2020.

[85] Lyndon While, Philip Hingston, Luigi Barone, and Simon Huband. A faster algorithm for calculating hypervolume. *IEEE transactions on evolutionary computation*, 10(1):29–38, 2006.

[86] Stefan Wildermann, Felix Reimann, Jürgen Teich, and Zoran Salcic. Operational mode exploration for reconfigurable systems with multiple applications. In *2011 International Conference on Field-Programmable Technology*, pages 1–8, 2011.

[87] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[88] Yuan Yao and Saketh Rama. yaoyuannnn/cava. https://github.com/yaoyuannnn/cava.

[89] Georgios Zacharopoulos, Lorenzo Ferretti, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca Carloni, and Laura Pozzi. Compiler-assisted selection of hardware acceleration candidates from application source code. pages 1–9, 2019.

[90] Jinghan Zhang, Hamed Tabkhi, and Gunar Schirner. Ds-dse: Domain-specific design space exploration for streaming applications. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 165–170, 2018.

[91] Yin Zhen, Yuan Hau, Nasir Shaikh Husin, Trias Andromeda, and Muhammad Nadzir Marsono. Energy-aware network-on-chip application mapping based on domain knowledge genetic algorithm. *Proceeding of the Electrical Engineering Computer Science and Informatics*, 1, 08 2014.

[92] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms - a comparative case study. pages 292–301. Springer, 1998.

Workloads	Audio Decoder					-
Tasks	Rotator Set	Psycho Filter	Rotate Order 1	Rotate Order 2	Rotate Order 3	-
f_i	216	2,645,938	219,130	663,496	1,611,608	-
I_{read}	0.00387	10.651	3.216	8.692	19.066	-
I_{write}	0.00387	10.651	17.833	32.397	56.208	-
Workloads	Audio Decoder					-
Tasks	Zoomer Set	Zoomer Process	FFT_left	FIR Filter_left	IFFT_left	-
f_i	54	2,918,392	658	1,358,412	833,970	-
I_{read}	0.218	15.154	0.00502	4.124	2.532	-
I_{write}	0.218	11.127	0.00200	4.124	2.532	-
Workloads	Audio Decoder					-
Tasks	Overlap_left	FFT_right	FIR Filter_right	IFFT_left	Overlap_right	-
f_i	81,916	658	1,358,412	833,970	81,916	-
I_{read}	0.249	0.00502	4.124	2.532	0.249	-
I_{write}	19.999	0.00200	4.124	2.532	19.999	-
Workloads	CAVA				-	-
Tasks	Scale	Demosaic	Denoise	Transform	-	-
f_i	46,221,100	33,528,040	696,574,288	94,791,760	-	-
I_{read}	507.629	92.056	1,912.552	260.265	-	-
I_{write}	126.907	92.056	1,912.552	260.265	-	-
Workloads	CAVA			-	-	-
Tasks	Gamut_map	Tone_map	Descale	-	-	-
f_i	162,608,100,840	41,367,960	6,244,079,520	-	-	-
I_{read}	446,465.522	113.582	17,144.080	-	-	-
I_{write}	446,465.522	113.582	68,576.318	-	-	-
Workloads	Edge Detection					
Tasks	Gaussian Smoothing	Laplacian Estimate	Compute Zero Crossing	Compute Gradient	Compute Max Gradient	Reject Zero Crossing
f_i	3,234,201,600	842,137,600	874,905,600	855,244,800	29,498,368	753,664,000
I_{read}	246.746	128.500	133.500	130.500	4.501	115.000
I_{write}	246.750	128.500	133.500	130.500	7,374,592.000	115.000

Table 4. Gable specification of our workload set.

Table 5. Workload's budget.

Workload	Audio	CAVA	ED
Latency (ms)	21.0	34.0	34.0
Power (mW)	8.737		
Area (mm^2)	17.475		

Appendices

A DETAILED WORKLOAD CHARACTERISTICS

Table 4 gives the detailed Gables-based characteristics of each task of the workloads. f denotes the instruction count of a task, and I denotes operation count per memory access, which is split into I_{read} and I_{write} corresponding to read and write, respectively.

B WORKLOAD'S DATA DEPENDENCY

Variations in the workload's input can influence the processing payload and thus the system's latency, energy, and power. We identify two input factors that impact the system's computation and communication payload.

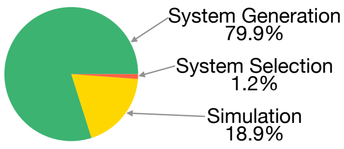
Input Parameters: Input parameters such as image size and resolution can impact the payload size. In this paper, we used each workload's input dataset provided by the benchmark developers ([37, 47, 88]). These datasets came with pre-determined, statically set parameters. Note that integrating a new dataset with different parameter values does not require a modification to our general methodology.

Systems such as AR glasses can experience payload variations by deploying runtime frameworks that dynamically adjust said parameters to improve system efficiency. This paper mainly targets systems without such flexibility and leaves these optimizations to future work.

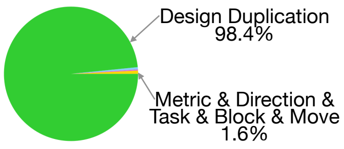
Operating Environment: Changes in the system's operating environment can result in payload variations. For example, variation in the number of traceable objects in an environment can impact SLAM's processing latency. To accommodate for this, we need to repeat the workload analysis stage in Database Generation (Section 3.1) for a sequence of input samples. This generates a sequence of TDGs with different instruction counts and data movement (payload) for nodes and edges, respectively. We then simulate each TDG and collect corresponding PPA values. Finally, we can apply a statistical function such as max (e.g., for worst-case design) or average to reduce each metric result to a scalar value. In this paper, our workloads are not prone to such variations. We leave the exploration of workloads with such a dependency to the future work.

C BUDGETING

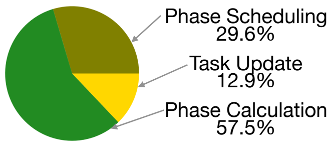
The latency budgets for the workloads are set according to the values suggested by [37] and [79]. The maximum target SoC power and area budgets are set to 0.1 W and 100 mm^2 according to [80]. However, these values assume an AR system containing 9 different workloads, whereas our system only contains 3 of such workloads. Thus, the following steps are taken to adjust the power/area budget. We profiled the power consumption of each workload in ILLIXR[37] individually on a desktop. The ratio of Audio power over all 9 workloads is then used to estimate Audio's budget in a .1W system. CAVA and Edge Detection power budget is also deduced according to their relative power consumption to Audio. System power is then calculated as the sum of all three workloads' power. For area budgets, we use power as a proxy and similarly use the breakdown in [37] to guide the budgeting. Note that these values are specified for the target technology of 5nm TSMC.



(a) FARSI system breakdown.



(b) System generation details.



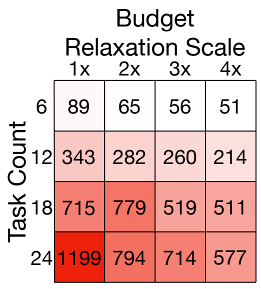
(c) System simulation details.

Fig. 20. FARSI time breakdown.

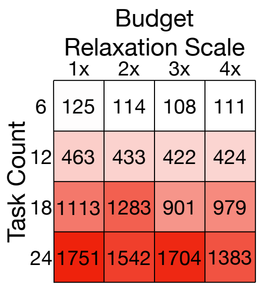
D PERFORMANCE BREAKDOWN

We profile FARSI's last three iterative stages to show what needs to be sped up. Note that the first stage (Database Generation) occurs only once, does not change across different runs, and further requires a minor annotation effort. The two tools used within this stage, i.e., accelSeeker and HPVM populate the database in less than a second.

Figure 20a shows the time breakdown of the last three stages. Most of the time (79.9%) is spent on the system generation stage. Further breakdown of this stage (Figure 20b) reveals that sub-stages such as task, block, or metric selection that do actual system modifications only consume a small percentage (2%<). Instead, design duplication, which copies the original design object to be modified/improved, consumes the most time. This step can be sped up by lowering the memory footprint for a faster `memcpy`. Simulation time breakdown is also shown in figure 20c. "Phase Interval Calculation" consumes most of our simulation time since it heavily uses the analytical models to iterate through the running tasks and corresponding hosting blocks, finds their bottlenecks, and calculates the phase duration accordingly.



(a) Iteration count.



(b) Neighbourhood size.

Fig. 21. Heuristic scalability with respect to the task count and system constraints.

E HEURISTIC SCALABILITY

We measure our heuristics scalability with respect to the workload complexity and system constraints. We use the number of tasks as the proxy for the former, and for the latter, we use the PPA budgets. In these studies, we use Edge Detection as the workload template to stress test the system as we have found experimentally that FARSI has the most difficulty converging for this workload. For workload complexity studies, we start with a system that runs one instance of Edge Detection (with 6 tasks) and incrementally increase the number of workloads and thus tasks running on the system. For system constraints studies, we start with the budget shown in table 5 and incrementally relax it, i.e., increase the budget across PPA by a scaling factor.

Heat map 21a shows the impact of these two variables on FARSI's convergence efficiency, measured in the number of iterations to convergence. Increasing the workload complexity by increasing the number of tasks (top to bottom) increases the number of iterations to convergence. On average 4× increase in this variable results in a 12.5× increase in the number of iterations. This is intuitively understandable as more tasks require more optimization for convergence. Similarly, tightening the budget (right to the left) increases the number of iterations. On average, 4× tightening of this budget results in a 1.7× increase in the iteration count. This is intuitively understandable as stricter constraints demand more system optimizations to meet the tighter budgets.

To go beyond intuition and concretely understand the relationship between these variables and convergence, we need to observe their impact on the "neighborhood size" of the designs encountered. A design's neighbor is a design whose mapping, allocation, or topology has been incrementally modified (please refer to the paper for more details). Thus, a design's neighborhood includes all the designs reachable through said incremental optimizations. The cardinality of this variable depends on (1) the number of tasks in the system (workload complexity) as each task can be incrementally optimized and (2) the number of IPs in the system (system complexity) as each IP can be optimized.

Since FARSI improves the design by greedily searching through neighbors, larger neighborhoods lead to more extensive search and thus higher iterations. Heatmap 21b shows the impact of task count and budget on the average neighborhood size encountered by FARSI while searching the space. Increasing the number of tasks by 4× on average results in a 14× increase in the number of neighbors. As we discussed in heatmap 21a, this in turn results in 12.5× increase in the number of iterations. Similarly, tightening the budget by 4×, increases the average neighborhood size by 1.2×, leading to 1.7× increase in the number of iterations.