SCIENCE CHINA
Information Sciences

• POSITION PAPER •

# Large Processor Chip Model

Kaiyan Chang, Mingzhi Chen, Yunji Chen*, Zhirong Chen, Dongrui Fan, Junfeng Gong, Nan Guo, Yinhe Han, Qinfen Hao, Shuo Hou, Xuan Huang, Pengwei Jin, Changxin Ke, Cangyuan Li, Guangli Li, Huawei Li, Kuan Li, Naipeng Li, Shengwen Liang, Cheng Liu, Hongwei Liu, Jiahua Liu, Junliang Lv, Jianan Mu, Jin Qin, Bin Sun, Chenxi Wang, Duo Wang, Mingjun Wang, Ying Wang*, Chenggang Wu, Peiyang Wu, Teng Wu, Xiao Xiao, Mengyao Xie, Chenwei Xiong, Ruiyuan Xu, Mingyu Yan, Xiaochun Ye, Kuai Yu, Rui Zhang, Shuoming Zhang & Jiacheng Zhao

*Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*

arXiv:2506.02929v1 [cs.AR] 3 Jun 2025

**Abstract** Computer System Architecture serves as a crucial bridge between software applications and the underlying hardware, encompassing components like compilers, CPUs, coprocessors, and RTL designs. Its development, from early mainframes to modern domain-specific architectures, has been driven by rising computational demands and advancements in semiconductor technology. However, traditional paradigms in computer system architecture design are confronting significant challenges, including a reliance on manual expertise, fragmented optimization across software and hardware layers, and high costs associated with exploring expansive design spaces. While automated methods leveraging optimization algorithms and machine learning (ML) have improved efficiency, they remain constrained by a single-stage focus, limited data availability, and a lack of comprehensive human domain knowledge. The emergence of large language models (LLMs) offers transformative opportunities for the design of computer system architecture and search paradigms. By leveraging the capabilities of LLMs in areas such as code generation, data analysis, and performance modeling, the traditional manual design process can be transitioned to a machine-based automated design approach. To harness this potential, we present the Large Processor Chip Model (LPCM), an LLM-driven framework aimed at achieving end-to-end automated computer system architecture design. The development of LPCM is structured into three levels: (1) Human-Centric, which assists in code generation and parameter tuning; (2) Agent-Orchestrated, facilitating cross-layer optimization through toolchain integration (e.g., LLVM, Gem5) and the autonomous execution of subtasks; and (3) Model-Governed, achieving full automation through the synthesis of hardware-software co-design, simulation, and iterative refinement. This paper utilizes 3D Gaussian Splatting (3D GS) as a representative workload and employs the concept of software-hardware collaborative design to examine the implementation of the LPCM at Level 1, demonstrating the effectiveness of the proposed approach. Furthermore, this paper provides an in-depth discussion on the pathway to implementing Level 2 and Level 3 of the LPCM, along with an analysis of the existing challenges.

**Keywords** Large Processor Chip Model, LLMs, Automated Design, 3D Gaussian Splatting

**Citation** Large Processor Chip Model. Sci China Inf Sci, for review

## 1 Introduction

**Computer System Architecture** is a fundamental discipline in the realms of computer science and engineering, concentrating on the design, organization, and performance optimization of computer systems. It comprises multiple components including compilers, CPUs, coprocessors, and RTL designs. From a software perspective, computer system architecture follows the layered architecture design philosophy and establishes multi-layer abstraction to bridge the gap between upper-layer applications and underlying physical hardware, providing a stable development environment for applications. From a hardware perspective, the evolution of computational demands in contemporary applications, coupled with advancements in algorithmic, continuously challenge the performance boundaries of underlying hardware, driving sustained innovation in underlying hardware technologies. As a bridge between software applications and

* Corresponding author (email: cyj@ict.ac.cn, wangying2009@ict.ac.cn)
† All authors contributed equally to this work and are listed in alphabetical order.

hardware technologies, computer system architecture fundamentally influences the performance, energy efficiency, reliability, and scalability of computing systems. The evolution and innovation in computer system architecture have become key drivers in advancing computing technology, playing a essential role in fostering technological innovation and addressing diverse computational demands.

### 1.1 The Evolution of Design Paradigms in Computer System Architecture

Since the advent of the first electronic computer ENIAC, in the 1940s, computer system architecture has undergone a structural evolution from a single form to a highly diversified landscape. From the 1940s to the 1960s, mainframe computers represented by ENIAC dominated the field of computing. With the advent of microprocessors and breakthroughs in integrated circuit technology, the 1970s to 1990s saw the widespread adoption of microcomputers and personal computers. The physical size of computing devices shrank to about 0.25 square meters, costs dropped to several hundred US dollars, and computing performance reached hundreds of millions of operations per second. During this period, computer architectures became increasingly complex due to functional generalization and performance demands, making hierarchical design and hardware-software co-optimization mainstream. To improve development efficiency, high-level languages such as C and compiler technologies advanced rapidly, freeing programmers from the burden of direct hardware manipulation. On the hardware side, considerations extended beyond arithmetic units to include CPU microarchitecture design elements like pipelines, multi-level caches, I/O systems, and bus architectures.

Entering the 21st century, especially after the resurgence of deep learning algorithms in 2012, the rapid development of emerging applications and advancements in semiconductor technology have shifted the focus from general-purpose architectures to domain-specific computer system [1]. Despite Moore's Law witnessed the reductions of the chip manufacturing cost, the escalating demand for specialized hardware tailored to a wide array of application requirements such as energy efficiency constraints, real-time performance requirements, has led to a substantial rise in chip development costs. However, current computer system architecture design not only heavily relies on domain experts but also suffers from long design cycles, making it difficult to meet the demands of rapidly evolving applications. Moreover, computer system architecture design encompasses a comprehensive process from high-level software interfaces to low-level hardware implementations, including compiler design, operating system optimization, hardware-software partitioning, micro-architecture design, RTL design, simulation, and verification, among other critical stages. This results in a vast design space with multi-objective optimization challenges, where traditional reliance on manual expertise severely constrains design performance and outcomes.

Automated computer system architecture design technology is an effective approach to enhancing the performance and efficiency of system architecture design. Automated design leverages various optimization algorithms, particularly those rooted in artificial intelligence, to rapidly explore the design space and achieve hardware-software co-optimization. This approach improves design performance, shortens design cycles, reduces development costs, and better meets the demand for customized system architectures in emerging fields such as artificial intelligence and high-performance computing. As a result, automated computer system architecture design is not only a key solution to current design challenges but also a significant driver of future computing technology innovation. Throughout the evolution of computer system architecture, the field has transitioned from manual design to automated design, with the capabilities of automation continuously strengthened by advancements in machine learning technologies, particularly deep learning. The evolution of automated design can be divided into three main categroies.

**Traditional Chip Logic Design Based on EDA (Electronic Design Automation) Tools** Traditional chip logic design based on EDA tools marks the starting point of automated design, signifying the transition from purely manual design to automated processes. Designers decompose and design internal functional modules of computer system architectures according to requirement specifications, translate these designs into hardware description languages (HDLs), and then utilize EDA tools for synthesis, verification, and analysis to generate logic circuits and ultimately physical layouts [2] [3] [4] [5] [6]. The introduction of EDA tools has enabled complex chip design tasks to be carried out more efficiently and accurately, significantly advancing the development of integrated circuits and computer system architectures. While traditional chip logic design based on EDA tools laid the foundation for modern automated computer system architecture design, this stage of design remained heavily reliant on manual rules and expertise, with the level of automation being highly limited.

**Optimization-Based Methods** Optimization-based methods represent a shift from traditional EDA-assisted automation to more intelligent and refined design optimization processes. At this stage, designers introduce advanced mathematical optimization techniques to explore a broader design space for optimal solutions while ensuring design accuracy and manufacturability. For example, design space exploration (DSE) employs heuristic search methods [7] [8] to identify the best trade-offs among performance, power, and area during architectural design and microarchitecture optimization. Logic synthesis methods generate optimized logic circuits through Boolean optimization [9], technology mapping [10], and Bayesian optimisation [11]. In the physical design phase, methods like placement and routing [12] and timing optimization [13] are used to optimize the physical implementation of processors [14]. These advancements have not only significantly improved design efficiency and precision but also laid the groundwork for subsequent machine learning-based design methods. However, these optimization-based approaches still rely heavily on human expertise, exhibit limited capabilities in handling complex designs, and often require substantial computational resources and optimization time, which can impact the effectiveness of the optimization process.

**Machine Learning-Based Methods** Machine learning-based methods represent a paradigm shift in automated design, transitioning from rule-driven and optimization-driven approaches to intelligent, data-driven methodologies. By learning from historical design data, machine learning can automatically generate new design solutions and even predict and address issues that are challenging for traditional optimization methods. For instance, techniques such as random forests [15] [16] and neural networks [17] have been integrated into design space exploration to predict the performance, power, and area of different configurations, thereby accelerating the exploration process and more efficiently identifying near-globally optimal solutions. Furthermore, reinforcement learning and neural networks (NNs) have been applied to physical design. In placement, reinforcement learning algorithms iteratively discover optimal micro placement strategies to minimize signal delay and congestion [18] [19]. NNs, on the other hand, contribute to performance prediction of timing, power, and congestion [20]. While these techniques apply machine learning methods to boost EDA efficiency or performance, they do not fundamentally transform the conventional design flow.

### 1.2 The key challenges preventing the evolution of design paradigms

Despite the significant advancements brought by machine learning methods to the automated design of computer system architectures, several challenges and issues remain.

First, computer system architecture encompasses a comprehensive design flow from high-level software interfaces to low-level hardware implementation, including stages such as compiler design, operating system support, processor architecture design, circuit design, physical design, verification, and evaluation. These stages are interconnected and mutually constrained, requiring a holistic consideration of their characteristics to achieve globally optimal designs. Nevertheless, existing automated design approaches are typically limited to single-stage optimization and cannot simultaneously handle multiple design stages, making end-to-end cross-stage optimization unattainable, and consequently restricting overall system performance improvement potential.

Second, traditional human-driven design processes have accumulated a wealth of design experience and rules, which provide valuable insights for achieving automation. However, current mainstream machine learning methods primarily rely on automatically extracting patterns from data, failing to effectively incorporate human design expertise and rules. Moreover, computer system architecture design involves multiple stages, each with its own unique design experiences and rules. The complexity and diversity of domain knowledge pose significant challenges for integrating human design knowledge into automated methods.

### 1.3 Large language models bring new opportunities

Recently, the emergence of large language models (LLMs) and agent systems has prompted researchers to explore the potential for automating the design of computer system architectures. These methods focus on utilizing LLMs to transform natural language descriptions of functional requirements or documentation into appropriate computer system architecture designs, which can lower the barriers to hardware development and improve the efficiency of research and development efforts.

Research focusing on LLM-driven technical solutions can be categorized into several key areas. First, studies targeting the generation of processor components concentrate on designing various functional modules of processors. Due to the scarcity of data in the processor design domain, these works propose diverse methods for constructing processor design datasets such as ChipNeMo [21], RTLCoder [22] and fine-tune large language models such as BetterV [23] on these datasets to obtain specialized hardware code generation models capable of producing Verilog or VHDL code. Second, research aimed at designing comprehensive frameworks focuses on building system-level automated design workflows. These works implement complex or system-level hardware architecture design such as ChatEDA [24] and ChipGPT [25], through modules such as task decomposition and composition, code verification and feedback, performance optimization, and search strategies. Additionally, these methods often incorporate automatic feedback and correction mechanisms, using simulation or formal verification tools to detect errors in the generated code and iteratively optimize code quality. Finally, research targeting analysis and verification tools utilizes large language models to understand, analyze, and summarize hardware code, serving as verification tools to automatically identify complex errors and security issues. Building on this, methods for automatically repairing erroneous code or optimizing power, performance, and area (PPA) have been proposed, e.g., AssertLLM [26] and RTLFixer [27]. These advancements demonstrate the potential of LLMs in automating and enhancing various aspects of hardware design, from component generation to system-level frameworks and verification tools.

Despite the significant potential demonstrated by large language models (LLMs) in computer system architecture automated design, existing approaches still suffer from several critical limitations. Firstly, current methods primarily focus on individual design stages, such as RTL-level hardware code generation or code verification, and fail to achieve a multi-level, holistic design process that spans from high-level requirements to low-level implementation. Moreover, they lack the capability to enable co-design across software and hardware layers, including compilers, operating systems, and processor design. This limitation results in a lack of end-to-end co-optimization in the design flow, making it challenging to achieve global optimization across performance, power, and area (PPA) objectives. Secondly, the correctness of the design outcomes remains difficult to guarantee. Although LLMs can generate syntactically correct hardware code, the complexity of hardware design and the stringent timing and resource constraints often lead to logical errors or functional defects in the generated code. As a result, manual verification or formal tools are still required to correct these issues. These shortcomings restrict the practical application of existing LLM-based automated design methods in real-world engineering scenarios.

### 1.4 Towards next-generation paradigm: Large Processor Chip Model

To harness the powerful capabilities of large language models (LLMs) for the automated design of computer system architectures, we propose the Large Processor Chip Model (LPCM), which is built on LLM technology and domain-specific data from computer system architecture. LPCM aims to achieve end-to-end automated design of computer system architectures. The design of LPCM offers unique advantages and holds significant importance for realizing automated computer system architecture design. First, leveraging the robust knowledge-learning capabilities of LLMs, LPCM can extract and utilize human expertise and knowledge in system architecture design from vast datasets, such as rules and heuristics in compiler optimization, microarchitecture design, and physical implementation. This capability enables LPCM to quickly grasp complex design logic and apply it to automated design workflows. Second, when the model's parameter size and data volume reach a certain scale, LLMs exhibit emergent abilities, leading to qualitative leaps in performance and behavior. Building on the emergent capabilities of LLMs, LPCM can perform multi-level, cross-domain co-optimization by simultaneously considering design constraints and objectives across multiple layers, such as compilers, operating systems, hardware architectures, and physical implementation. This enables LPCM to generate system architecture designs that surpass human capabilities. Such global design capabilities not only significantly improve design efficiency but also achieve better trade-offs among performance, power, and area (PPA) objectives, driving computer system architecture design toward higher levels of intelligence and automation.

To realize LPCM, multiple modules need to be collaboratively designed to cover the full technology stack from high-level software interfaces to low-level hardware implementation. This includes several critical components such as compiler design, operating system support, hardware-software partitioning, microarchitecture design, RTL (Register Transfer Level) design, and simulator development. The compiler translates high-level language programs into machine instructions, enabling software to inter-

act efficiently with hardware. The operating system manages hardware resources and provides abstract interfaces, ensuring efficient resource allocation and system stability. Hardware-software partitioning determines whether specific functionalities are implemented in hardware or software, balancing performance, flexibility, and design complexity. Microarchitecture design focuses on optimizing processor performance and energy efficiency by defining the internal structure and data flow of the processor. RTL design implements the hardware logic, specifying the behavior of digital circuits at the register transfer level. Finally, simulators are used to verify and evaluate system performance, ensuring that the design meets functional and performance requirements before physical implementation. By integrating these modules, LPCM can achieve a comprehensive and automated design flow, enabling end-to-end optimization and validation of computer system architectures. This holistic approach not only enhances design efficiency but also ensures that the final system meets the desired performance, power, and area (PPA) objectives.

Achieving LPCM is a highly challenging task, whose complexity and interdisciplinary nature determine that this goal cannot be accomplished overnight but requires gradual advancement in stages. Based on the varying degrees of automation in the design process, the implementation can be divided into the following three levels.

**Level 1: Human-Centric Hierarchical Design and Optimization.** In the human-centric hierarchical design and optimization phase, LPCM serves as auxiliary tools to assist humans in designing computer system architectures. Humans, as the primary drivers of the design process, are responsible for setting goals of hierarchical design, such as instruction set architecture, memory hierarchy, compiler optimization strategies, and more. They accomplish the main design tasks with the help of existing system architecture design tools like LLVM, GEM5, Chisel, and Verilog simulators. The primary role of LPCM in this phase is to provide suggestions to humans, such as generating code snippets, proposing optimization algorithms, or offering hardware description language (HDL) templates. However, for the design of complex components, such as processor pipelines and cache coherence protocols, the decision and optimization still heavily rely on human expertise.

During this level, human involvement accounts for a significant portion of the work, and the use of design tools is frequent. The contributions of LPCM are limited, primarily focusing on repetitive tasks like code generation and parameter tuning, or knowledge retrieval tasks such as searching for relevant research papers, tool documentation, or design specifications. Human experts are responsible for the hierarchical verification and optimization of the design results, ensuring that the design goals at each level, such as compiler layer, hardware architecture layer, and hardware module layer, are consistent and efficient.

In this level, LPCM possess limited domain-specific knowledge. It may directly employ general-purpose large language models like ChatGPT or DeepSeek without undergoing deep fine-tuning for the computer system architecture domain or integration with domain-specific tools like LLVM, Gem5, or Chisel. As a result, the outputs of these models require rigorous review and optimization by human experts, especially in cross-layer designs such as hardware-software co-design.

**Level 2: Agent-Orchestrated Cross-Layer Design and Optimization.** In the agent-orchestrated cross-layer design and optimization phase, LPCM act as intelligent agents capable of independently completing certain subtasks, such as automatically generating compiler optimization passes, designing simple processor microarchitectures, and creating operating system scheduling algorithms. Humans only need to define the design objectives for these subtasks, such as performance metrics and power constraints, while LPCM autonomously handle the design, evaluation, and error correction of these subtasks. For example, through domain-specific fine-tuning, LPCM can generate hardware description language (HDL) code, design pipeline configurations, and cache hierarchies for processor microarchitectures. For cross-layer optimization tasks like hardware-software co-design, LPCM can coordinate design goals across different layers, such as the compiler layer, hardware architecture layer, and hardware module layer, and automatically integrate and invoke toolchains like LLVM and Gem5 to accomplish specific tasks. In terms of evaluation, LPCM can utilize simulation tools such as Gem5 and Verilog simulators to assess hardware design performance, generate performance reports on throughput, latency, and power consumption, and conduct corresponding analyses. LPCM can also automatically detect issues in the design, such as performance bottlenecks, functional errors, or power consumption violations, and generate corrective solutions, such as adjusting pipeline configurations, modifying cache coherence protocols, or optimizing scheduling algorithms, followed by regenerating the design. However, for highly complex tasks, such as designing multi-core cache coherence protocols, human expert intervention and optimization are still required.

During this level, human involvement significantly decreases, and the reliance on design tools is reduced,

while the responsibilities of LPCM increase substantially. LPCM play a central role in exploring the design space and can rapidly generate and evaluate multiple design solutions, such as different processor pipeline configurations and memory hierarchies. Additionally, LPCM can automate verification tasks, including formal verification and simulation testing, to ensure design consistency and correctness.

In this level, LPCM possess extensive domain-specific knowledge, having been fine-tuned using domain data such as research papers, open-source projects, and toolchain documentation from the computer architecture field. They are capable of understanding and generating code and design documents that comply with domain-specific standards. Furthermore, LPCM can propose optimization suggestions based on cross-layer design requirements, such as matching compiler optimizations with hardware characteristics or coordinating operating system scheduling with processor microarchitecture, thereby achieving overall performance improvements.

**Level 3: Model-Governed Autonomous Design and Optimization.** In the model-governed autonomous design and optimization phase, LPCM achieve full automation of the entire design process, independently completing all subtasks such as compiler optimization, operating system scheduling, instruction set architecture design, and processor microarchitecture design. LPCM also establish a complete closed loop encompassing system design, evaluation, and error correction. LPCM can autonomously invoke toolchains like LLVM, Gem5, and Verilog simulators to perform design, simulation, verification, and optimization, entirely replacing manual design and the use of traditional design tools. Through cross-layer optimization in hardware-software co-design, LPCM ensure consistency in design goals across all levels and achieve optimal overall performance.

During this level, humans only need to propose high-level objectives, such as "design a power-efficient RISC-V processor", and LPCM can autonomously complete the entire design process. The involvement of LPCM approaches 100%, with almost no need for human intervention. LPCM can automatically decompose tasks, generate design solutions, and iteratively optimize them using simulation and verification tools like Gem5 and Synopsys VCS. Based on design goals and constraints such as performance, power consumption, and area, LPCM can explore the design space and identify optimal solutions.

In this level, LPCM possess extensive domain-specific knowledge, reaching or even surpassing the level of domain experts in terms of design efficiency and quality. It can integrate the latest research advancements, such as novel memory technologies and quantum computing architectures, to propose innovative design solutions, driving the forefront of computer system architecture development.

## 2 Compiler meets LLM

### 2.1 Motivation

Compilers play a crucial role as crucial intermediaries between human-readable source code and machine-executable code, ensuring reliable application performance across diverse hardware architectures. With the emergence of domain-specific applications, specialized architectures have been designed to enhance performance and reduce power consumption. However, significant challenges persist in the collaborative design of software and hardware, as iterative updates typically demand months or even years to complete. This bottleneck has prompted researchers to investigate automated methods for compiler construction.

The traditional compiler toolchain process, ranging from lexical analysis and parsing to optimization and code generation, is well-established but demands substantial expert knowledge, especially when adapting to new architectures. This dependency results in a bottleneck, primarily attributed to the scarcity of experienced compiler engineers. The LLM compiler is designed to tackle this limitation by automating the adaptation of compilers for emerging domain-specific architectures, thereby facilitating rapid development iteration cycles between hardware modules and software modules within the LPCM framework.

Large language models have demonstrated exceptional capabilities in code generation and pattern recognition, offering a promising approach to addressing the challenges in compiler development. Within the LPCM framework, the compiler module serves as a critical bridge connecting software applications, hardware specifications, and hardware architectures. This connection has the potential to significantly accelerate development cycles by enabling automated transformation and optimization. While AI-powered tools like GitHub Copilot, TabNine, and Cursor have enhanced general code development productivity, they often exhibit limitations in comprehending complex system architectures and compiler design princi-

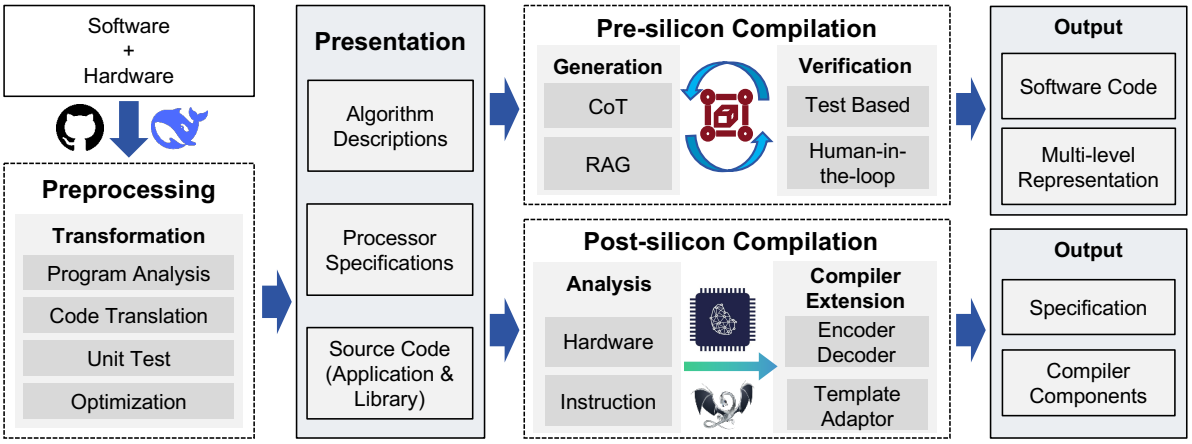


**Figure 1** LLM Compiler in Large processor chip model: an overview

ples. However, their effectiveness remains constrained by limited exposure to compiler source code during training. Thus this field continues to depend heavily on scarce expert knowledge. The LLM compiler approach aims to address this limitation by developing specialized models with deeper understanding of compiler construction. This not only alleviates the engineering bottleneck but also enables more efficient code generation tailored for iteratively evolving hardware architectures.

There is an increasing demand for specialized large language models in compiler-related tasks within system architecture research. These expert-level LLMs are crucial for adapting to changes in emerging domain-specific architectures, facilitating collaboration between hardware and software development, and enhancing the co-design process within the LPCM. By training with datasets that incorporate compiler knowledge, such as source codes and assembly instructions, the automation and intelligence of system software within a LPCM can be significantly enhanced, thereby improving development workflows significantly. When a new ISA is produced—either a novel design or an extension—there is typically no immediately available compiler toolchain to support it. To this end, the compiler module must undergo iterative updates to ensure efficient code generation and execution tailored to the new architecture. The connection pathway of the compiler module with its adjacent components within the LPCM framework is described as follows: *1) Inputs.* The compiler module receives the software source code from upstream modules containing new algorithm implementations and partition information, and hardware basic specifications for further implementation, which include parameters that the compiler should consider during the generation and optimization process. *2) Outputs.* The compiler module generates the transformed source code and appropriate intermediate representations for downstream modules, such as DSE modules, and constructs compiler components that can be integrated into conventional compiler systems during iterative design refinement.

## 2.2 The overview of LLM Compiler

To leverage the capabilities of LLMs and realize fully automated compiler design without human intervention, we propose the **LLM Compiler**, an advanced compilation toolchain powered by LLMs, tailored specifically for compilation and programming tasks within LPCM. However, achieving the goal of unmanned intervention is a long-term process that cannot be accomplished overnight. To support this objective, we define three levels based on technological development trends and the varying capabilities of LLMs as follows:

- **Level 1: Assisted Compiler Development**. In this process, LLMs utilize compilation knowledge obtained during training or fine-tuning, take engineers' requirements and the code context to be edited as input, and produce edited code as output, including source code analysis and transformation, as well as suggestions for compiler component generation. This capability effectively accelerates engineers' development work, as demonstrated by systems like Cursor.
- **Level 2: Semi-Autonomous Compiler Components construction**. LLMs can fulfill one or more critical roles in three primary compiler tasks: decision-making, integration with the existing compilation system software, and execution of specialized compiler functions. By employing an agent-based framework, the entire workflow, spanning from application intake to compilation and hardware

adaptation, can be partially automated. However, human intervention remains essential for refining and adjusting inputs and outputs across modules, as the compilation process is inherently error-prone.

• **Level 3: End-to-End Compiler Generation and Execution**. LLMs can autonomously manage the entire compilation and deployment process. Human users are only required to specify task objectives or provide inputs, after which the system delivers the final results.

**Related Work.** Most contemporary research and development efforts remain at Level 1, where LLMs provide supportive functions while humans maintain primary control over the compiler development process. Representative code completion and automatic programming tools, such as GitHub Copilot and TabNine, belong to this category. These tools generate code snippets and provide completion suggestions but exhibit limited understanding of the knowledge in compiler domains. The source code translation approaches presented in [28] and [29], as well as code LLM models such as CodeBERT [30] and CodeT5 [31], similarly function at Level 1. These methods provide support to developers but lack autonomous decision-making capabilities. Emerging work is progressively advancing to Level 2, where LLMs assume semi-autonomous roles in constructing compiler components. ComBack [32] showcases the LLM-driven automation of instruction selection and register allocation, whereas specialized models introduced in [33] are capable of translating high-level code into extended ISA instructions with minimal supervision. VeGen [34], which focus on SIMD vectorization, also belong to this category. These approaches partially automate compiler components but still require human intervention for seamless integration. Systems at Level 3, which are capable of end-to-end compiler generation and execution with minimal human intervention, remain largely theoretical. Recent neural compilation methods [35,36] provide promising foundations for potentially realizing such systems. The progression from Level 2 to Level 3 signifies the current research frontier, demanding substantial advancements in LLMs' capabilities to reason about intricate compiler architectures, manage cross-component interactions effectively, and produce both reliable and optimized code generation pathways.

Our LLM Compiler, currently at Level 2 and targeting advancement to Level 3, encompasses two distinct approaches specifically designed to meet different development requirements: **LLM as Compiler** and **LLM generates Compiler**.

#### 2.2.1 *LLM as Compiler*

LLMs can directly serve as translation components for converting source code to an extended ISA design, functioning as an LLM Compiler. In this design paradigm, the process differs based on two distinct scenarios. In the first scenario, when working with an existing Domain-Specific Architecture (DSA) that incorporates new hardware components, the LLM Compiler identifies code segments suitable for acceleration on the new hardware while maintaining compatibility with the established DSA framework. In the second scenario, when confronting an entirely new DSA or significant instruction set extensions, the LLM Compiler must perform comprehensive analysis and translation of the source code to fully leverage the novel architectural capabilities, effectively bridging the gap between conventional programming paradigms and the innovative instruction set.

The LLM first functions as an analyzer, assessing the characteristics of the source code and partitioning it into multiple sub-regions (using either basic blocks, control blocks, or functions as granular units). For each sub-region, the LLM evaluates whether it can be mapped to extended instructions or operators. If mapping is feasible, we assume that translating to extended instructions will yield benefits. The cost model provided by the ISA design is then used to select the mapping approach that offers the greatest advantage.

Subsequently, the LLM acts as a translator, translating the source code according to the selected mapping scheme. It is important to recognize that this translation process is prone to errors; therefore, it requires constraint information and examples from the ISA design as few-shot guidance. Additionally, the results must be verified for functional equivalence against the original program—specifically, by comparing the output of the compiled code to that produced when executed on a pure CPU—to ensure the correctness of the generated code.

Looking toward the future, LLM Compilers within LPCM will likely evolve to incorporate reinforcement learning from verification results, allowing them to continuously improve translation accuracy and optimization strategies. They may also develop capabilities to suggest architectural modifications based on observed software patterns, effectively participating in the co-design process rather than merely implementing it. As these models mature, they could potentially generate entire compiler toolchains for new

architectures automatically, eliminating one of the major bottlenecks in deploying novel computational paradigms.

#### 2.2.2 *LLM generates Compiler*

LLMs can also function as code generators, contributing to the development of compiler components. Traditionally, as hardware architectures evolve through successive iterations, adapting compiler backends to new ISAs requires manual modifications, leading to substantial development costs. However, we observe that despite varing ISA features, compiler backends exhibit consistent structural patterns and invariant operational norms across their analysis and transformation passes.This consistency provides the foundation for LLM-automated compiler generation.

Using LLVM as an example, its compiler backend processes maintain four canonical processing phases regardless of target architecture: instruction selection, register allocation, instruction scheduling, and code emission. Each phase follows deterministic implementation patterns while utilizing Architecture-specific information and stable transformation algorithms. The hardware characteristics required by these phases are uniformly described in target description `.td` files using TableGen, a DSL.

However, LLM faces challenges when generating system-level compiler code, especially when processing natural language requirements that often fail to highlight extended instruction set specifications. To address this challenge, we propose a systematic approach of analyzing existing LLVM-supported ISAs to construct structured few-shot examples linking natural language requirements, TableGen modifications, and Pass adaptation code generation. This framework aims to effectively guide LLMs in extending the LLVM compiler for new ISAs.

Furthermore, the characteristics of compiler backends allow us to extract specific implementation patterns from the workflow in LLVM. We aim to build a compiler backend-oriented fine-tuning dataset to enhance open-source LLM's accuracy in generating compiler backend code.

In future work, these capabilities will be integrated into LPCM, supporting automated adaptation of the compiler module, thereby enabling prototype architecture to be rapidly assessed. This approach complements the LLM as Compiler method by addressing distinct phases of the compiler development lifecycle within the LPCM, facilitating rapid hardware-software co-evolution.

## 3 Binary Translation meets LLM

### 3.1 Motivation

The rapid evolution of processor architectures has brought unprecedented challenges and opportunities to software ecosystem construction for emerging processors. The binary translation module is designed to enable seamless migration and efficient execution of applications across heterogeneous Instruction Set Architectures (ISAs), thereby breaking down ecosystem barriers and accelerating the adoption of new processor platforms.

Traditional binary translation approaches face two main challenges: (1) **High development cost and long cycles**, as translators must be manually crafted and tuned for every new ISA combination and scenario, and (2) **Limited adaptability and scalability**, with heavy reliance on expert knowledge and hand-crafted rules, making it hard to keep up with the rapid proliferation of ISAs and emerging hardware-software co-design requirements. These challenges underscore the urgent need for automated, intelligent binary translation that can efficiently adapt to new architectures and workloads.

What sets our module apart is its integration of Large Language Models (LLMs) to achieve automation throughout the entire binary translation workflow. Leveraging LLMs, the module can not only automate the generation of binary translators but also deliver tailored outputs—such as customized instruction sets and hardware optimization suggestions—based on dynamic program behaviors and hardware features. This marks a paradigm shift from traditional, labor-intensive approaches to a data-driven, intelligent automation framework.

Within the LPCM framework, our module plays a pivotal role: it aims to dramatically reduce development time, automate the generation and optimization of binary translators, and provide the critical outputs as shown in Figure 2. which include key information—including binary translation hardware extension descriptions, hardware-software co-optimization suggestions, and tailored instruction sets. This information can be delivered to the modules of Architecture simulator and Design Space Exploration

for simulation and implementation, supporting upstream hardware design optimization and downstream software migration and deployment, and facilitating efficient interaction among LPCM modules.

## 3.2 Overview of binary translation tool

### 3.2.1 *Development Stages and Related Work*

Across the three levels of LPCM development, we systematically leverage LLMs to introduce intelligence into the binary translation module, focusing on the unique needs and characteristics of binary lifting, instruction mapping and equivalence transformation, as well as dynamic behavior analysis and hotspot detection.

• **Level 1: LLM-Assisted Binary Analysis and Lifting**. In this level, the binary translation toolchain is mainly developed and refined by human experts, with LLMs serving as auxiliary analysis engines. The LLM assists with tasks such as binary lifting—converting binaries to intermediate representations (e.g., LLVM IR)—and the extraction of reduced instruction sets for application-specific translation. The output includes a manually crafted binary translator and a reduced instruction set, which is provided to the Design Space Exploration (DSE) module.

• **Level 2: Agent-Orchestrated Binary Translation and Optimization**. At this level, LLMs evolve from auxiliary tools to orchestrating agents, coordinating multiple submodules such as program analysis and hardware optimization. The LLM-driven agent invokes relevant analysis tools, identifies hot code regions, performs instruction slicing, and generates hardware optimization descriptions.

• **Level 3: End-to-End Autonomous Binary Translator Generation and Optimization**. In this advanced level, the LLM autonomously governs the entire pipeline, model-governed autonomous design and Optimization from binary analysis to translator generation, based on architectural models of both source (guest) and target (host) processors. This enables the rapid, automatic production of highly compatible translators and tailored hardware/software optimization recommendations. Although few current systems have reached this level, it represents the ultimate goal of fully automated, intelligent binary translation.

**Related Work.** Situating recent binary translation research within our three-level automation framework helps clarify the state of the field and the trajectory of progress. The majority of contemporary efforts remain at level 1, where the core binary lifting and translation toolchains are still primarily developed by human experts, with LLMs or AI serving as auxiliary engines. Tools and frameworks such as BOLT [37] and Lightning [38] exemplify this phase: they leverage sampled runtime information and predefined rules for code layout optimization or binary rewriting, but the pipeline is fundamentally rule-based, requiring extensive manual engineering to adapt to new architectures or applications. Recent years have witnessed a growing body of research advancing towards level 2, where LLMs and machine learning models orchestrate and automate increasingly significant portions of the binary translation process. For example, Wong et al. [39] employ GPT-4 as the core of an end-to-end decompilation framework, enabling self-refinement of generated code. Projects like Forklift [40] and LLM4Decompile [41] illustrate alternative strategies: Forklift trains an end-to-end code-lifting model with enhanced scalability across architectures, while LLM4Decompile augments end-to-end models with refinement modules built atop traditional tools, demonstrating that targeted refinement can outperform naive end-to-end approaches. True level 3 systems—where models autonomously govern the entire binary translation and optimization pipeline, from analysis to code generation and hardware adaptation—are still largely aspirational.

### 3.2.2 *Proposed framework design*

Our module aims to drive the transition towards LLM-governed binary translation systems within LPCM. We plan to:

• Develop a unified LLM-driven framework that adapts to diverse ISAs and application domains, supporting both static and dynamic translation needs.

• Deeply integrate program behavior analysis and hardware co-design, supplying actionable insights for both software migration and hardware optimization.

• Continuously provide high-quality, standardized outputs—customized translators, optimization suggestions, and tailored ISAs—to other LPCM modules, expediting system-wide co-evolution and ecosystem expansion.

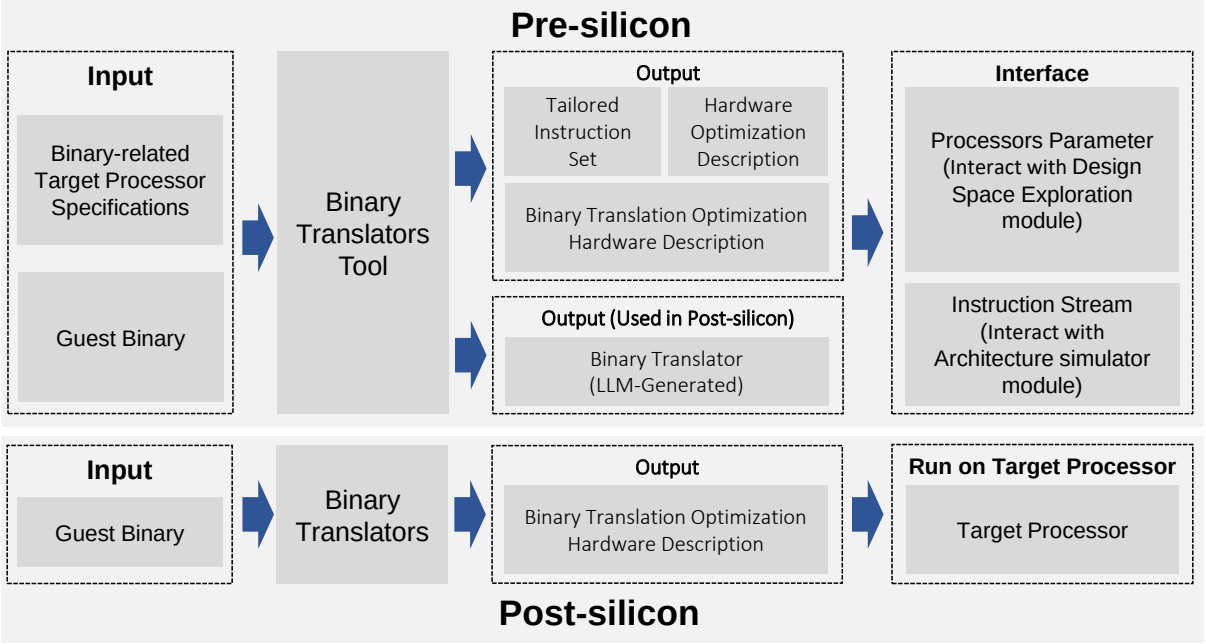


**Figure 2** The hierarchical workflow of the LLM-driven binary translation module within LPCM.

Our work aims to build an advanced binary translation tool that intervenes as early as the pre-silicon design phase to introduce a paradigm shift, as illustrated in Figure 2. The primary objective is to automate the generation of binary translators, thereby significantly reducing or even eliminating the need for manual development. Tailored instruction sets, Hardware optimization descriptions and Binary translation optimization hardware description interact with other modules, while the tool continuously acquiring the latest processing data and performance evaluation results to support the iterative development of the advanced binary translation tool itself. In line with the three levels of LPCM development, the output process of the binary translation tool is also divided into three iterative phases within the workflow.

In the post-silicon verification phase, where the target processor design has been finalized, the binary translation tool shifts its focus to the construction of the runtime environment. It facilitates the stable and efficient execution of applications and dependent libraries originally developed for other ISAs or environments on the finalized hardware. Specifically, the binary translation tool supports rapid application porting and deployment, ensures backward compatibility, continuously optimizes hardware performance, and contributes to building a comprehensive software ecosystem for the new hardware. This comprehensive approach ensures that both hardware and software are fully integrated to maximize performance and usability across diverse computing environments.

In summary, the binary translation module serves as a crucial enabler for cross-ISA application migration, hardware/software co-optimization, and the intelligent evolution of system architectures within LPCM, ultimately bridging the gap between emerging hardware and rich software ecosystems.

## 4 Simulator meets LLM

### 4.1 Motivation

Computer architecture simulators are software tools that model the structure and performance of computer systems or components such as CPUs and memory. As semiconductor technology advances, chip architectures become increasingly complex, making these simulators essential for research and development. They facilitate experimental analysis, rapid design iteration, and cost minimization before finalizing designs. However, as research in domain-specific acceleration evolves rapidly, the traditional approach to developing these simulators, which relies on human expertise, struggles to keep up with the pace of innovation. This development paradigm faces several significant challenges:

(1) **High Learning Curve**. Mature simulators like GEM5 [42,43], booksim [44], and QEMU [45] have

developed extensive and complex codebases over the past decade, posing challenges for new developers. Understanding their architectural design and implementation can require $2\tilde{4}$ weeks, hindering efficient modifications and research in hardware architecture.

(2) **Simulator Composition Challenges**. Computer system architecture comprises several modules, including the CPU, GPU, NPU, DRAM, and SSD. However, existing simulators typically focus on individual components and employ distinct simulation methods and communication interfaces. This complexity can hinder development efficiency, underscoring the need for an automated integration scheme to streamline the process.

(3) **Prolonged Development Cycles**: The construction of traditional simulators typically involves multi-level abstract modeling, which spans from ISA functional validation to micro-architecture timing simulation, and is generally carried out manually. However, as the complexity of computer architecture system design continues to grow, the conventional manual implementation of simulators faces challenges such as tedious programming, intricate verification issues, and a time-consuming design space exploration process. These factors hinder the swift iteration of computer architecture system design.

Fortunately, the capabilities of large language models, particularly their advanced functionalities in code generation, code analysis, and debugging, offer an opportunity to address the aforementioned challenges and reform the development paradigm of computer architecture simulators. Thereby, it is time to leverage LLMs to design an efficient framework for the automated generation of computer architecture simulators, enabling automated transformation of user requirements or research papers into computer system architecture simulator. The input and output of an automated design framework for computer system architecture simulator should have the following functions:

**Input**: The framework primarily accepts simulation configurations and simulated workloads as input. These configurations can be represented in various modalities, including but not limited to natural language, documents, tables, and structured formats such as JSON and XML. The simulated workloads are generated by the upstream compiler module and typically appear in binary form or as user-defined intermediate representations.

**Output**: The output of the framework mainly includes two parts: complete simulator project codebase and user-required performance metrics & analysis results. The former is the engineering code corresponding to the generated architecture simulator, including source code, documentation, etc., provided to users for code verification or secondary modification, while the latter is the simulation results required by users to verify the effectiveness of the design and achieve design optimization iteration.

### 4.2 LLM-driven computer architecture simulator automation design framework

Despite the remarkable progress in LLM technology in recent years, developing an LLM-driven automated design platform for computer system architecture simulators with comprehensive end-to-end capabilities remains a significant challenge that cannot be overcome in a short time. In this context, this paper presents an automation grading standard specifically for the design of computer system architecture simulators powered by LLMs. As shown in Figure 3, the automated design process is organized into three levels, categorized according to the dimensions of functional realization and scenario requirements.

• **Level 1: Intelligent Configuration of Simulators Based on General-Purpose LLMs**. The objective of Level 1 is to simplify and alleviate the cumbersome process of manually configuring simulators by harnessing the capabilities of advanced LLMs. By developing an efficient and intelligent mechanism that automates the conversion of natural language commands into precise simulator configurations, researchers can shift their focus away from the tedious aspects of simulator setup, allowing them to dedicate more energy and creativity to the computer system architecture design.

• **Level 2: Agent-Driven Simulator Composition**. The L1 level is dedicated to the automatic configuration of parameters for a single simulator, whereas the L2 level builds on this foundation by focusing on the automated interconnection of multiple simulators. The primary goal of Level 2 is to utilize large language models (LLMs) to bridge gaps among existing simulators in terms of simulation mechanisms, communication protocols, and interface standards. Attaining Level 2 has the potential to significantly reduce the time and effort researchers invest in simulator integration, while also improving the overall efficiency and flexibility of simulator design.

• **Level 3: Autonomous Simulator Design, Generation, and Optimization**. By utilizing the vast and varied data resources collected during Level 2, such as simulator source codes, workloads, and dynamic performance benchmark datasets, we can create domain-specific LLMs tailored for simulators.

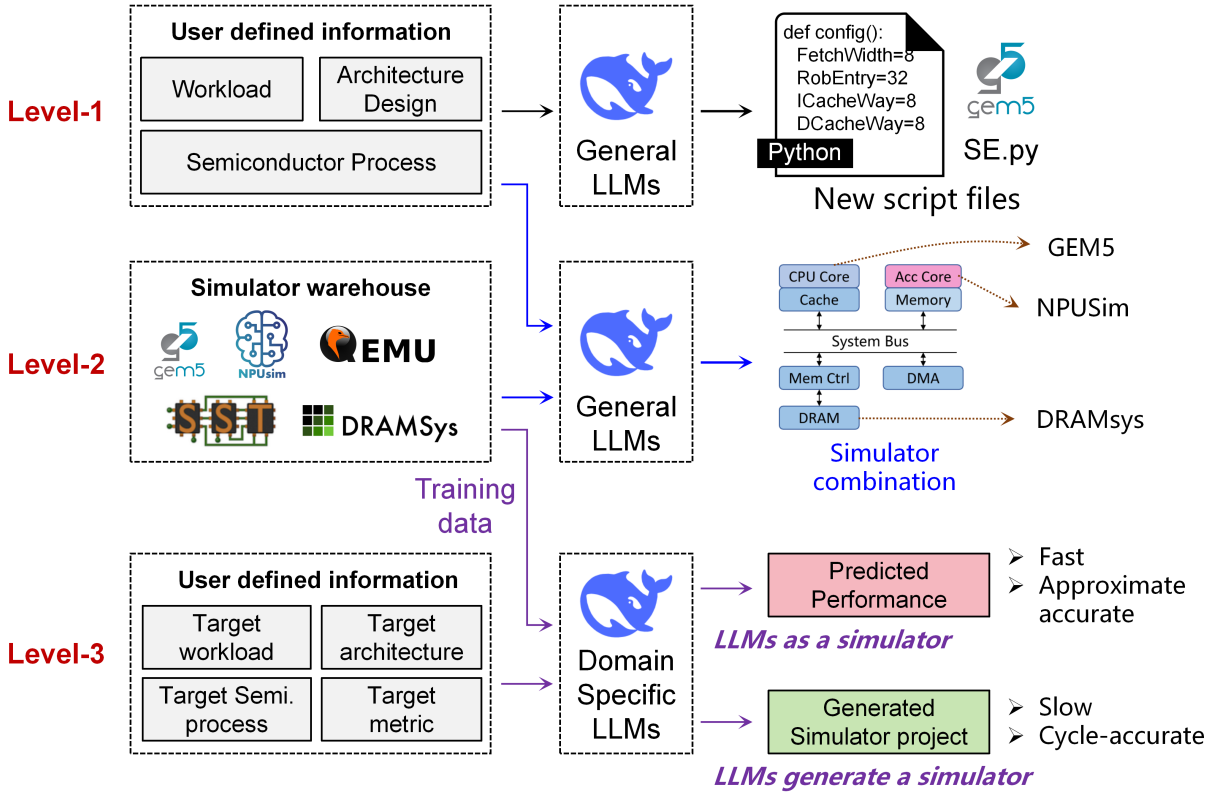


**Figure 3** The three levels of design automation for LLM-driven computer system architecture simulators.

This progress has the potential to transform the design approach of computer system simulators, moving from a human-centered methodology to complete process automation.

We have developed and implemented an intelligent computer system architecture simulator configuration solution leveraging the capabilities of a general-purpose LLMs, successfully accomplishing the core objectives of the L1 stage. In our experiments within the CPU domain, we selected the DeepSeek-V3 [46] as the foundational model to configure GEM5 [42] simulator. By processing user-defined parameter configuration requirements, we effectively generated customized *SE.py* simulation script files that meets user requirements, utilizing the unmodified *SE.py* file as the starting template. Meanwhile, the generated scripts successfully passed the workload tests for 3D Gaussian Splatting [47] (3DGS), confirming their validity and accuracy. For the NPU domain experiments, we utilized the DeepSeek-V3 model, without any fine-tuning, and employed partial systolic array modules from the Gem5-Aladdin [48] simulator as configuration templates. With minimal human feedback for adjustments, we successfully created an NPU simulator specifically optimized for the 3DGS workload, based on the configuration parameters provided by the user. After integrating the generated code segments into the Gem5-Aladdin project, the simulator successfully passed the GEMV and GEMM tests within the workload, further establishing its performance and reliability. In our end-to-end 3DGS workload simulation experiments, we conducted comprehensive simulations of the 3DGS algorithm using the generated CPU and NPU modules. For the CPU module, we automated the complete parameter search process by integrating it with a design space exploration module, evaluating eight different parameter configurations before identifying the optimal set. Similarly, for the NPU module, we conducted an in-depth exploration of six key parameters, including systolic array dimensions and SRAM size, through the design space exploration module, ultimately determining the best configuration. Subsequently, the generated modules are integrated into the Gem5-Aladdin framework, facilitating inter-module communication through shared memory to support the accelerated execution of the 3DGS algorithm. The experiments contrasted CPU-only mode with CPU-NPU mode for simulating 3DGS. The results revealed that the CPU-NPU simulator achieved over a 20% improvement in end-to-end performance compared to CPU-only execution for the 3DGS algorithm. This outcome not only confirms the effectiveness of our solution in automating simulator configuration but also highlights its considerable potential for enhancing algorithm execution performance.

In future work, we will further explore using LLMs to achieve Level 2 and Level 3 automated computer architecture simulator generation. Specifically, to achieve efficient integration of L2 level simulators, we will first develop a standardized interface protocol for simulator interconnection. This will enable plug-and-play connectivity among simulators using different simulation mechanisms by defining a unified data exchange format and communication specifications, thereby avoiding the technical bottleneck associated with extensive code refactoring in traditional integration approaches. Secondly, we will establish a comprehensive knowledge base of computer system architecture. This knowledge base will include two main types of data: the first encompasses simulator engineering data, systematically featuring the code bases of prominent architectures such as GEM5 [42] and Booksim [44]; the second comprises knowledge on simulator interconnection and design methodologies. By leveraging the capabilities of LLMs alongside this knowledge base, the system will be equipped to semantically parse user requirements, identify the optimal simulators from the knowledge base, and seamlessly facilitate efficient interconnection among different simulators through the use of standardized interface protocols.

To achieve the goal of automating the design and generation of architecture simulators without human intervention at the level 3, we leverage the impressive capabilities of large models in code generation and performance prediction. Our approach consists of two distinct technical routes, focusing on the core dimensions of simulator accuracy and speed.

(1) **LLM as an simulator**. By gathering workload code, hardware architecture details, and performance data from code executed on various architectures, we engage in domain-specific LLMs for computer architecture system simulator. This model is capable of accurately analyzing different workloads and hardware design schemes, simulating the execution process of workloads on various hardware architectures, and generating precise user required performance evaluation results. Essentially, the LLMs evolve into a simulator that incorporates performance prediction capabilities, making it suitable for architecture simulation and performance evaluation.

(2) **LLM generates an simulator**. The distinctive strengths of LLMs in code generation offer us a remarkable opportunity to translate researchers' design requirements into computer architecture system engineering code. In this way, researchers need only to articulate their design specifications, targeted performance metrics, and other critical information in natural language. The LLMs can then automatically interpret these requirements, transforming them into precise and standardized computer architecture system engineering code through their advanced semantic understanding and coding capabilities.

## 5 HW/SW partition meets LLM

### 5.1 Motivation

As heterogeneous computing systems grow increasingly complex, hardware-software partitioning has become an indispensable step in the development process. Hardware-software partitioning is essential for rapid validation and iteration. At the early stages of system development, partitioning helps designers quickly establish system models, estimate performance, and analyze bottlenecks, providing clear guidance for subsequent hardware design and software development. By effectively distributing tasks between software and hardware modules, it not only optimizes performance and power consumption but also maximizes resource utilization. However, traditional partitioning methods face the following major challenges:

1. **Low development efficiency and lengthy optimization cycles**: Hardware-software partitioning involves complex task decomposition, hardware resource mapping, and interaction logic analysis. Manual tuning is time-consuming and struggles to meet the demands of rapid iteration.

2. **High dependency on expertise**: Partitioning requires developers to possess deep expertise in both hardware architecture and software design, increasing human resource costs and potentially leading to suboptimal results when expertise is lacking.

3. **Underutilized optimization opportunities**: Without systematic learning from historical design patterns and large-scale data, manual partitioning often fails to uncover hidden performance optimization potential.

To address these challenges, a highly efficient framework leveraging LLMs for hardware-software partitioning automation can be designed to support end-to-end optimization, from requirement specification to partitioning scheme generation. The ideal framework's input and output paradigms are as follows:

**Input**: The framework should support diverse input formats, including natural language descriptions of requirements and original task C/C++ code. To enable flexible customization, it should also allow users to specify hardware resource parameters and partitioning constraints via structured formats such as JSON.

**Output**: (1) Partitioning scheme: The framework generates detailed partitioning schemes tailored to specific hardware architectures, specifying the allocation of each task to software or hardware. (2) Performance prediction report: Outputs key performance metrics such as power consumption, performance, and latency to help users quickly evaluate the scheme's quality.

With this design, the framework can significantly reduce development cycles and optimization time while uncovering hidden performance improvement opportunities through large-scale data learning. This dual enhancement of development efficiency and design quality represents a transformative advancement in the field of hardware-software co-design.

### 5.2 Proposed framework design

Based on the performance and involvement of LLM we proposed above, there are three progress challenging design approaches:

- **Level 1: Human-guided HW/SW Partitioning**. At this level, human engineers remain the primary decision-makers, while LLMs serve as intelligent assistants. Leveraging their capabilities in natural language understanding and code analysis, LLMs can provide concise summaries, interpret complex modules, and analyze memory access patterns. These insights help engineers make informed design decisions more efficiently, although the final partitioning choices are still made by humans.
- **Level 2: Agent-Aissisted HW/SW Partitioning**. At this level, LLMs take on a more proactive role. Acting as intelligent agents, they can parse source code, generate task graphs, and suggest preliminary partitioning strategies based on computational characteristics and platform constraints. Through iterative feedback from users, these agents can refine their recommendations, enabling a co-creative workflow between humans and machines.
- **Level 3: LLM-Driven Autonomous HW/SW Partitioning & Optimization**. At this level, LLMs are deeply integrated with compilers, performance models, and learning frameworks such as graph neural networks. They are capable of executing the entire pipeline: analyzing code, constructing task graphs, estimating computational and memory profiles, and conducting multi-objective design space exploration to identify Pareto-optimal partitioning schemes. Moreover, LLMs can automatically generate the necessary software-hardware interface code, facilitating seamless deployment.

Most current approaches to HW/SW partitioning still rely on traditional methods, with limited involvement of LLMs. These conventional techniques typically depend on combinatorial optimization, heuristic algorithms [49] [50], and performance estimation models to determine task allocation. Heuristic-based methods—such as genetic algorithms [51] [52], simulated annealing [53], and hill climbing [54]—aim to iteratively explore the solution space in search of optimal or near-optimal partitioning strategies under constraints like performance, power consumption, and hardware resource utilization.

According to the input and output requirements outlined in Section 1.1, the LLM-based automated hardware-software partitioning framework must fulfill two core functions: (1) Decomposition of Complex Tasks: Automatically analyze and decompose the input program into multiple simpler subtasks and generate the corresponding task dependency graph. (2) Rapid Task Performance Estimation and Architecture Mapping: Quickly estimate the performance and resource consumption of individual nodes and the entire task graph, enabling predictions of overall PPA (Power, Performance, and Area) under different hardware-software partitioning schemes.

Based on the considerations above, we propose a method combining LLMs and graph learning models to achieve these key functionalities. The overall workflow of the proposed automated hardware-software partitioning framework is shown in Figure 4. The framework primarily consists of two components: (1) **LLM-Driven Task Graph Generation**: This component automates the decomposition of input programs and generates the corresponding task dependency graph. The goal of decomposition is to transform the input program from a complex and tightly coupled structure into a more transparent and interpretable task graph data structure. In the task dependency graph, nodes represent different architectural tasks,

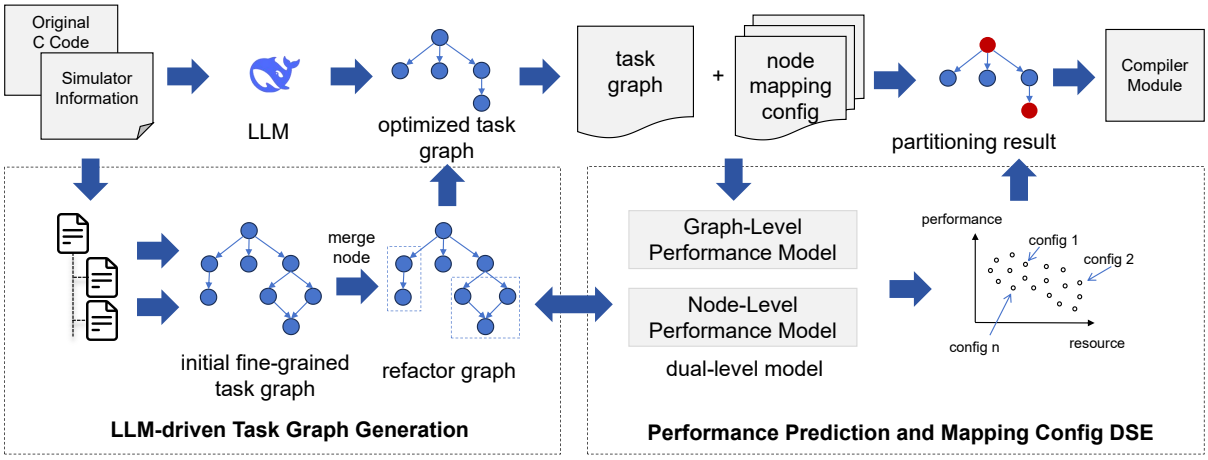


**Figure 4** Automated hardware-software partitioning framework

while the dependencies between these tasks are captured by the edges of the graph. This approach reformulates the hardware-software partitioning problem into a problem of managing dependencies between nodes in the task dependency graph, enabling more efficient and actionable task scheduling and partitioning in subsequent steps. (2) **Graph Learning Model-Based Task Graph Performance Prediction and Partitioning**: This component employs a dual-layer performance prediction framework powered by graph learning models to rapidly assess the performance and resource consumption of individual nodes in the task dependency graph, as well as the overall performance of the task graph under different mapping configurations. This framework facilitates efficient exploration of the design space, enabling the identification of the optimal hardware-software partitioning scheme under given architectural and resource constraints, thereby optimizing system performance and resource utilization.

### 5.2.1 *LLM-Driven Task Graph Generation*

As the first process of HW/SW partitioning, this step leverages LLM to generate an appropriate task dependency graph from the original C/C++ source code provided by the user. It begins by using an LLM to produce an initial task graph with the finest granularity. In this initial graph, each function in the input C/C++ code is treated as a separate task node, and edges are constructed based on function calls and data dependencies. During this phase, the LLM analyzes the semantics and structural relationships of the source code to generate nodes and infer the initial dependency edges. Next, LLM performs graph optimization through node merging. It iteratively examines all directly connected node pairs and evaluates the benefit of merging each pair, aiming to balance execution time and communication overhead. The merging benefit is estimated based on the predicted performance characteristics of the nodes and the volume of data exchanged between them. After evaluating all candidate pairs, LLM selects the one with the highest gain for merging, updates the task graph accordingly. This process repeats iteratively until no further beneficial merges can be performed. Through the above methodology, LLM generates a task dependency graph that balances both execution time and communication overhead. The final output is a structured JSON representation of the task graph, produced by the LLM.

### 5.2.2 *Graph Learning Model-Based Task Graph Performance Prediction and Partitioning*

This work is based on a dual-layer performance prediction model utilizing graph learning techniques. The goal is to efficiently estimate the performance and resource consumption of each node in the task dependency graph while predicting the overall performance of the entire task graph under different node-to-architecture mapping configurations rapidly. This framework enables effective design space exploration, allowing the identification of optimal software-hardware partitioning solutions within given architectural and resource constraints, ultimately optimizing system performance and resource utilization. The workflow of performance prediction is shown in Figure 6.

The performance prediction model consists of two layers: a node-level performance prediction model and a graph-level performance prediction model. In the node-level model, each node in the task dependency graph is treated as an independent subtask. The model takes as input the C/C++ code

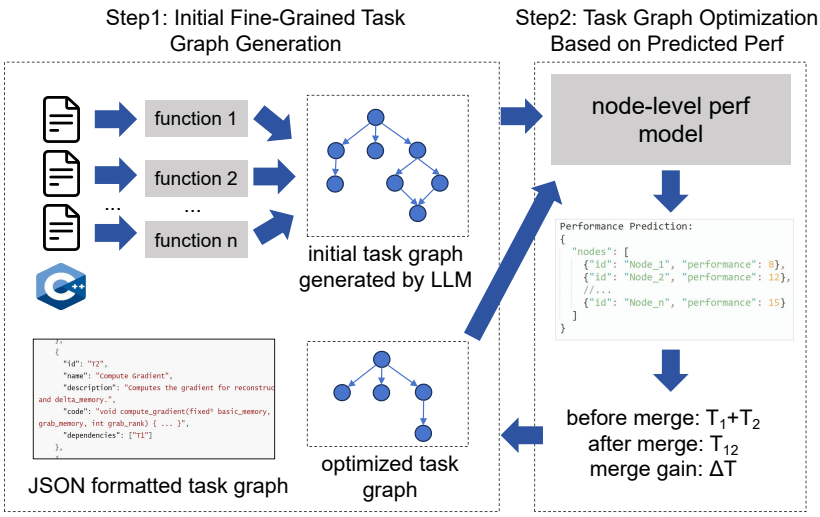


**Figure 5** Workflow of task graph generation

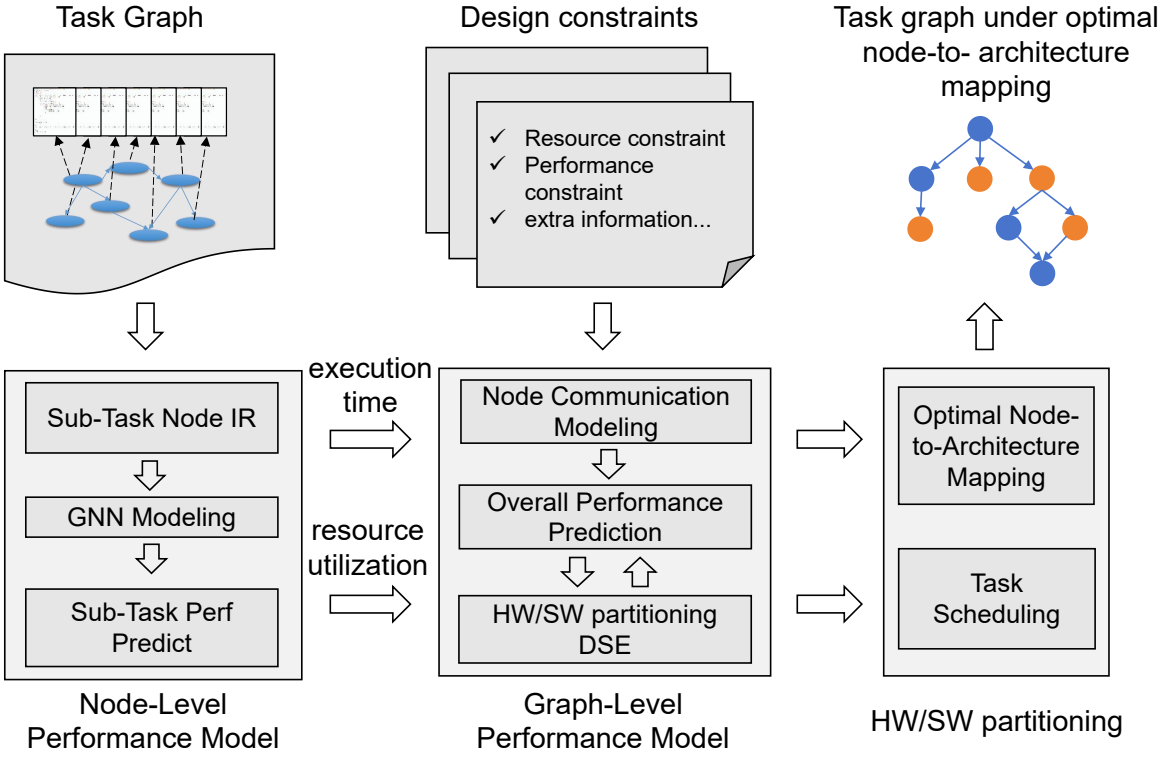


**Figure 6** Dual-layer performance prediction.

corresponding to each subtask and outputs predictions for the subtask's performance and resource consumption on different platforms (CPU and NPU). Specifically, the subtask code is first transformed into an intermediate representation (IR) using high-level synthesis tools. This IR effectively abstracts the underlying computational and memory access characteristics of the code, facilitating subsequent modeling. Next, a graph neural network (GNN) is employed to model the performance and resource consumption of each node. The GNN's ability to capture local dependencies within the IR structure enhances the accuracy of the predictions by leveraging both the features of the task nodes and the structural relationships in the IR.

The graph-level performance prediction model is designed to rapidly evaluate the overall performance of the task graph under specific node-to-architecture mapping configurations. The model's input includes the entire task dependency graph, the architecture mapping configuration of the nodes, and performance data for each subtask as predicted by the node-level model, such as execution time and resource consumption. This model accounts for critical factors such as inter-node communication data, task scheduling, and resource contention within the task graph to construct a comprehensive performance model.

Based on the output of the graph-level model, we can quickly evaluate the performance of the task dependency graph across all feasible node-to-architecture mapping configurations. By combining this with predefined resource constraints, the framework enables efficient design space exploration. This process ultimately identifies a reasonable software-hardware partitioning scheme and task scheduling strategy that ensures optimal overall system design.

## 6 Design Space Exploration meets LLM

### 6.1 Motivation

Design Space Exploration (DSE) plays a vital role as a strategic intermediary between algorithmic requirements and hardware architecture, enabling the co-design of CPUs and co-processors under stringent performance, power, and area (PPA) constraints. As computational demands continue to diversify, CPUs must adapt through architectural refinement to maintain versatility, while co-processors leverage specialization to accelerate targeted workloads. Together, they push the boundaries of system-level performance. However, this architectural flexibility results in exponentially growing design spaces and increasingly complex parameter interactions.

Traditional manual DSE processes are typically slow, requiring months or even years of iterative optimization, and have become a significant bottleneck in the hardware development lifecycle. This challenge has led researchers to pursue intelligent DSE techniques tailored for both CPUs and co-processors, aiming to rapidly identify optimal design points across vast configuration spaces. By automating this process, such methods promise to significantly reduce development time and foster faster hardware-software co-evolution.

However, achieving automated DSE comes with several key challenges. (1) Complex Integration within the System Architecture Flow: The DSE module must ensure consistency, compatibility, and accurate information exchange across various modules within the system architecture design workflow. Currently, significant manual intervention is still required to maintain coherence and correct functionality throughout the flow. (2) Offline Learning and Limited Adaptability: As new architectures, technologies, and design methodologies continue to emerge, DSE modules must be frequently updated to remain effective. However, enabling robust online learning while avoiding catastrophic forgetting remains a significant challenge, limiting the system's ability to adapt continuously and autonomously.

The integration of LLM into the DSE module can effectively addresses above challenges. LLM acts as a seamless coordinator across different modules, automatically understanding and transmitting design requirements and constraints, thereby reducing manual intervention and improving system consistency and efficiency. Additionally, LLM's robust knowledge generalization capability enables it to adapt to new architectures and technologies without the need for frequent retraining. By supporting incremental learning, LLM avoids catastrophic forgetting, ensuring continuous evolution. Through interactive optimization with designers, LLM fosters a dynamic, adaptive, and intelligent design process, significantly enhancing the automation and intelligence of the DSE module.

To address these challenges, we design a DSE module powered by LLMs. This module plays a foundational role in navigating the vast design space of modern processors. Within the LPCM framework, the

primary objective is to autonomously generate and refine hardware designs that meet diverse functional, performance, and cost requirements—where both the CPU and co-processor are critical system components. The DSE module, in this context, refers to the structured and intelligent process of exploring and evaluating a wide range of CPU and co-processor configurations and architectures. Its goal is to identify the most optimal hardware design that satisfies the multi-dimensional constraints and objectives defined by LPCM. The role of DSE module in the LPCM can be outlined as follows.

• **Automated Design Exploration.** DSE module automates the process of exploring numerous configurations of CPUs and co-processors, including cores, cache hierarchies, instruction sets, coprocessor architectures, extended instructions, cache coherence, and power/area/performance trade-offs. By systematically adjusting these parameters, DSE module identifies the configuration that best meets performance, power, area, and cost objectives.

• **Rapid Iteration and Optimization.** By leveraging performance predictions, different configurations are rapidly evaluated in an iterative manner, where feedback from each evaluation cycle informs subsequent design adjustments. This continuous refinement process significantly shortens development cycles and reduces reliance on costly and time-consuming trial-and-error methods.

• **Holistic System Optimization.** DSE module does not operate in isolation. Instead, it is part of a broader system-wide optimization process. The design of the CPU and co-processor is considered in conjunction with other system components such as compilers, software/hardware partitions, and memory subsystems. This interconnected approach ensures that the CPU and co-processor are optimized for efficient operation within the larger system, creating a fully integrated and optimized architecture.

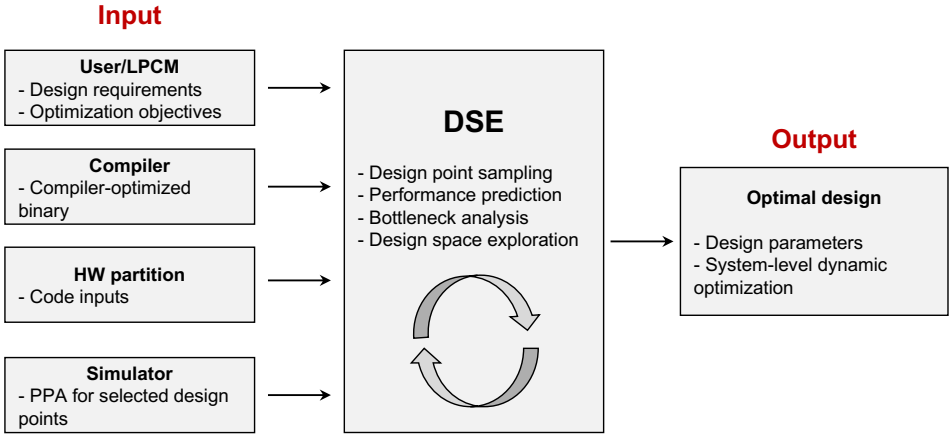


**Figure 7** Workflow of CPU DSE module on LPCM.

The DSE module operates as illustrated in Figure 7. In the **Input**, it gathers essential inputs including design requirements and optimization objectives from the top-level LPCM, workload characterization code from the hardware-software partitioning module, compiler-optimized binaries for performance modeling, and PPA results of selected design points from the simulation module for training and validation purposes. During the exploration, the DSE module performs autonomous design point sampling with adaptive density control, predicts PPA metrics for unexplored configurations without requiring external simulation, and conducts bottleneck analysis to identify architectural inefficiencies—such as suboptimal cache hierarchy or pipeline depth—informing microarchitectural refinements. These capabilities are integrated to guide efficient and intelligent exploration of the design space toward optimal architecture configurations. Finally, in the **Output**, the DSE module delivers the optimized microarchitecture parameters—including core topology, execution units, and cache structures—to the RTL code generation backend. Simultaneously, LPCM leverages insights from the DSE process to perform system-level dynamic optimization, adjusting OS settings, compiler strategies, and hardware acceleration schemes, thus establishing a closed-loop co-optimization flow across the stack.

### 6.2 The Overview of DSE Module

To leverage the capabilities of LLMs and realize fully automated design space exploration without human intervention, we propose the LLM DSE, an advanced exploration framework powered by LLMs, specifi-

cally designed for prediction and optimization tasks within LPCM. However, achieving the goal of fully autonomous hardware design is a long-term endeavor that requires progressive technological advances. To support this objective, we define three levels based on the development trends and varying capabilities of LLMs, as illustrated in Figure 8.

• **Level 1: Human-Guided and LLM-Assisted DSE**. At this level, LLMs function as intelligent assistants to streamline repetitive tasks such as organizing input data, linking evaluation tools, and generating basic configurations. While the core decision-making remains with human experts, LLMs contribute by chaining sub-modules and facilitating design flow integration. The system remains highly reliant on expert guidance for simulation execution, performance evaluation, and iterative refinement. LLMs support input preprocessing and workflow scripting, but do not participate in active exploration or optimization.

• **Level 2: LLM-Driven Semi-Automated DSE**. At this level, LLMs assume a more proactive role by autonomously generating candidate configurations, analyzing performance feedback, and iteratively refining design proposals under human supervision. Human experts define high-level objectives and constraints, while LLMs explore the design space by leveraging pre-trained knowledge and retrieval-augmented generation (RAG) techniques. The interaction becomes bidirectional—LLMs propose adjustments based on previous evaluation results, and experts respond by refining constraints or redefining targets. This level enables partial autonomy while retaining expert control over critical decision points within the design workflow. To effectively support diverse application scenarios, a general-purpose co-processor DSE framework is required. This framework is typically built upon heuristic algorithms such as genetic algorithms, with LLMs embedded into key components to enhance adaptability and decision-making efficiency.

• **Level 3: LLM-Governed Fully Autonomous DSE**. At this level, LLMs govern the entire design space exploration process with minimal to no human intervention. The system autonomously synthesizes design inputs, generates and evaluates candidate architectures, and selects optimal configurations through internal reasoning and feedback-driven learning. It performs holistic, cross-layer exploration and supports multi-objective optimization across key metrics such as performance, power, and area. LLMs at this stage act as central decision-makers, orchestrating the full loop of configuration generation, simulation execution, result analysis, and iterative refinement, thereby achieving a high degree of autonomy in hardware design.

Although recent efforts do not directly employ LLMs for DSE, they can serve as submodules within the DSE module. These submodules include design samplers, performance models, and exploration methods. The design samplers encompass techniques such as random sampling [55], orthogonal design [56,57], and Pareto-aware sampling [58]. The performance models include linear regression [59], spline models [60], neural networks [55, 56, 61, 62], Gaussian processes [63], XGBoost regressors [64], and GBRT regressors [57, 58, 65]. Prediction accuracy can be further enhanced by leveraging transfer learning and meta learning techniques from known workloads [66–70]. The exploration method includes heuristic search and acquisition functions. The heuristic search generates candidate design points using predefined rules, including random descent [71], genetic algorithm (GA) [72–76], and simulated annealing [77, 78]. The acquisition function relies on the statistical characteristics, including uncertainty [56], expected improvement [79, 80], and Pareto charateristics [57, 58, 63, 65].

Moreover, the design space of co-processor architectures is highly complex, and as task loads continue to diversify, it is crucial to thoroughly address the challenges of generality in co-processor design space exploration. The exploration of this design space can be classified into three primary levels based on the degree of generality in the methodologies employed. The first level focuses on optimization for single-task execution on specific co-processor architectures, with studies at this level typically involving the design of custom DNN accelerators [81, 82]. The second level deals with the optimization for general tasks on specific co-processor architectures, exemplified by automated design methods for systolic arrays [83, 84]. The third and most advanced level concerns the automated selection and optimization of co-processor architectures for general tasks. Although research in this area is still limited, studies like [85] have introduced methods for designing general-purpose co-processors, offering valuable insights for future co-processor design space exploration.

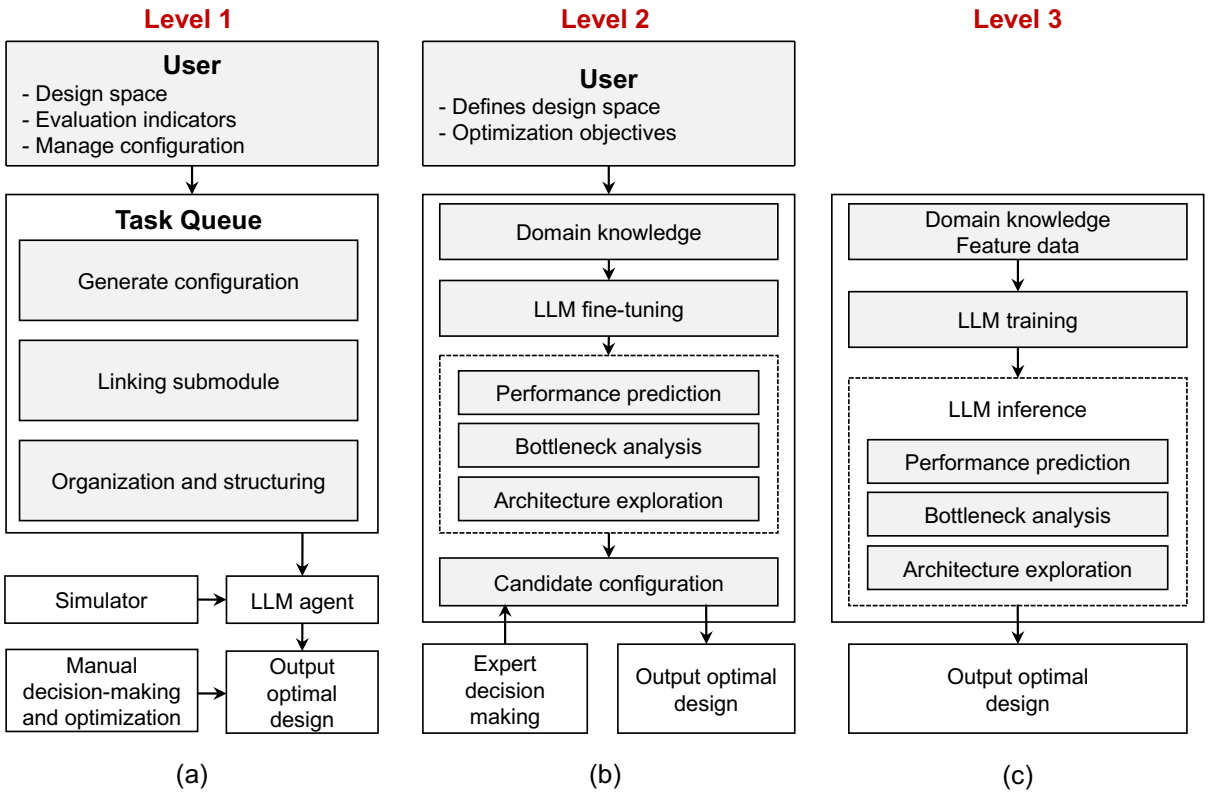


**Figure 8** DSE module at (a) Level 1, (b) Level 2, and (c) Level 3.

### 6.2.1 *Key Future Designs*

To achieve higher levels of automation, adaptability, and intelligence in processor design space exploration, future advancements must go beyond static decision-making pipelines. In this section, we present two key enablers that underpin the transition from human-guided design to fully autonomous systems: (1) a modularized reasoning and adaptive decision-making framework that empowers LLMs with structured, interpretable, and feedback-driven capabilities, and (2) a unified knowledge graph with online learning mechanisms to ensure continuous evolution and system-wide knowledge integration. Together, these foundational designs lay the groundwork for scalable, explainable, and continuously improving DSE processes across all levels of automation.

**Modularized Reasoning and Adaptive Decision-Making Framework.** To support fully autonomous and reliable design space exploration at advanced automation levels, we propose a modular reasoning and adaptive decision-making framework that empowers LLMs with structured, verifiable, and context-aware inference capabilities. By decomposing the DSE process into modular sub-decisions—such as architectural parameter tuning, pipeline depth configuration, and power gating strategy selection—the LLM can apply localized heuristics and domain rules to each decision unit. This modular structure simplifies learning, supports hierarchical optimization, and improves reasoning interpretability. In parallel, the framework integrates adaptive reasoning mechanisms that dynamically adjust exploration strategies based on feedback signals such as constraint violations, PPA degradation, or convergence stagnation. Reasoning chains are established to trace the causal relationships between design choices and performance outcomes, enabling explainable inference and reduced decision uncertainty. In the context of L1 to L3 levels of DSE module, we propose a modularized reasoning and adaptive decision-making framework to facilitate increasing autonomy and effectiveness at each level.

At L1, the modular reasoning framework plays a supporting role. LLMs assist human experts by organizing and structuring design parameters into modular units. For example, sub-decisions like architectural parameter tuning and pipeline configuration can be structured into modules, enabling the LLM to propose initial configurations. The modular approach simplifies the management of multiple design parameters, but human experts still provide final decisions. Adaptive decision-making here primarily supports experts by flagging potential issues (e.g., performance degradation or constraint violations) based

on the modular structure, without taking full control.

At L2, the LLM can autonomously apply modular reasoning to evaluate and optimize design configurations. The framework will allow the LLM to perform more sophisticated analyses, such as dynamically adjusting strategies based on the feedback from previous iterations. For example, if a configuration violates constraints or results in PPA degradation, the LLM can adjust its exploration strategies by altering specific modules (e.g., adjusting power gating strategies or reconfiguring pipeline depth). The integration of adaptive reasoning ensures that the LLM can autonomously adjust the design space exploration process and make iterative improvements based on the feedback loop from simulation results.

At L3, the modular reasoning and adaptive decision-making framework allows the LLM to take complete control of the design space exploration process. The LLM independently generates and optimizes designs by making decisions across multiple modules and adapting its exploration strategy based on continuous feedback. Reasoning chains will enable the LLM to link design choices to performance outcomes, facilitating self-correction and improving design decisions autonomously. This results in fully optimized configurations and performance metrics without the need for human intervention.

**Online Learning with Unified Knowledge Graph Integration.** To realize continuous adaptation and system-wide design intelligence, we introduce a unified knowledge graph framework coupled with online learning capabilities. The knowledge graph integrates heterogeneous data sources across abstraction layers, including microarchitectural configurations, RTL component libraries, EDA constraints, and empirical performance logs. Through a RAG-enabled interface, the LLM can retrieve relevant subgraphs in real time to inform design decisions, enabling contextual awareness and accurate cross-domain reasoning. Online learning pipelines are incorporated to incrementally refine the LLM's internal representation using newly acquired simulation results, verification outcomes, and emerging design patterns. This setup allows the system to autonomously evolve its design knowledge, adapt to novel hardware requirements, and remain aligned with the state of the art in CPU architecture and optimization techniques. In line with the increasing autonomy at each level of DSE, we introduce an online learning framework that leverages a unified knowledge graph to enable continuous adaptation of the LLM.

At L1, the knowledge graph serves as a central repository for organizing and referencing domain-specific information, such as component libraries and design constraints. The LLM can query the graph to suggest candidate solutions, but the expert remains in control of integrating new data. While online learning is limited at this stage, the system can assist in refining the decision-making process by providing relevant information and drawing from historical data for the experts.

At L2, the knowledge graph becomes more interactive, with the LLM retrieving real-time data from the graph to refine its design decisions. The integration of online learning allows the system to update its knowledge continuously based on simulation results and verification outcomes. As the LLM autonomously explores design configurations, it can incorporate new performance data, thereby improving its decision-making accuracy over time. The system becomes increasingly efficient in suggesting configurations that meet design goals based on updated knowledge and feedback.

At L3, the knowledge graph is fully integrated into the LLM's decision-making process. The LLM can autonomously query the graph for relevant data and adjust its optimization strategy based on newly acquired insights. The online learning pipeline allows the system to evolve continuously, adapting to emerging design patterns and technological advancements. The LLM uses the updated knowledge base to drive its optimization process, making decisions based on the most current and relevant data available, ensuring that the design process stays aligned with cutting-edge developments.

## 7 HDL generation module

### 7.1 Motivation

The Hardware Description Language (HDL) generation module automatically creates executable HDL code based on the optimal design generated by the above Design Space Exploration Agent. It processes diverse inputs such as natural language specifications, C code, and control flow graphs to interpret design parameters and perform system-level dynamic optimization using a pre-trained model. The module includes an input interface, a multimodal HDL generator, and a PPA (Power, Performance, Area) prediction feedback mechanism to ensure functional correctness and performance optimization.

As chip design complexity grows, integrating HDL generation into large processor chip models becomes

essential—not only to enhance efficiency and reduce human errors, but also to achieve global optimization. This approach overcomes the limitations of traditional manual HDL writing while enabling cross-layer optimization, resulting in a more efficient and accurate design process. Below are key reasons why HDL generation should be embedded within such models.

• Collaborative Optimization for Global Optimality

Cross-layer collaborative optimization integrates HDL generation into system-level modeling frameworks, enabling seamless coordination between different design layers. By establishing closed-loop feedback among compilation optimizations, simulation tools, and design space exploration, this approach not only enhances design correctness and performance but also ensures component compatibility and consistency, driving toward a globally optimal solution.

Consider the case of a complex microprocessor design. If HDL coding is done in isolation, it might result in under-use of available resources or suboptimal trade-offs between power consumption and performance due to ineffective communication with other design layers. However, by adopting a multimodal modeling approach at the system level, designers can make more informed decisions based on comprehensive data analysis, leading to an optimized overall design.

• Overcoming Limitations of Existing RTL Generation

Traditional RTL code generation methods often suffer from incomplete information and insufficient validation feedback, making it difficult to guarantee the accuracy of the generated code. Without robust verification mechanisms, errors can propagate unchecked. In contrast, modern large-scale processor chip models incorporate end-to-end support, from compilation optimization to simulation and design space exploration, establishing a closed-loop feedback system that significantly improves design reliability and correctness.

Handling highly complex hardware design requirements, such as architectural intricacies and resource constraints, poses significant challenges for traditional approaches, often leading to impractical designs or overly complex training and inference processes. Advanced large processor chip models offer a more effective solution by streamlining these complexities, enabling efficient automation of HDL generation while ensuring scalability and adaptability to diverse design constraints.

## 7.2 Overview of HDL Generation Module

### 7.2.1 *Development Stages*

We describe HDL generation across the following three levels. Currently, we are at level 3.

• **Level 1: Prompt-centric Copilot**. In this initial stage, general LLMs are used to generate HDL code. Debugging prompts play a crucial role here. Previous works directly prompt LLMs to generate HDL code or explore prompt optimization using feedback from EDA tools.

• **Level 2: Fine-tuning Submodel**. In the fine-tuning stage, the HDL model is one of the submodels of LPCM. It has two key characteristics. Large models fine-tuned on domain-specific datasets are used to generate HDL code. Although these training datasets are limited and do not yet encompass all multimodal data, significant progress has been made. Agents are employed to optimize models for HDL code generation, though their applications remain somewhat narrow and lack comprehensive coverage.

• **Level 3: MLM-governed Submodel**. In this stage, the MLM-governed submodel of LPCM leverages fully trained Multimodal Large Models (MLMs) to autonomously generate HDL code during the chip design process. These MLMs operate independently, invoking external tools without human intervention.

**Table 1** Comparison of Different Levels of HDL Generation Systems

| Level | Does the Model Require Training? | How EDA Tools Are Used? |
|---|---|---|
| **Level 1** | No training: General-purpose LLMs generate HDL code directly. | Passive verification: EDA tools verify HDL code without participating in generation. |
| **Level 2** | Fine-tuned: Large models adapt to domain-specific datasets for accurate HDL generation. | Active optimization: EDA tools act as agents, refining HDL code via feedback. |
| **Level 3** | Full training: Multimodal LLMs (MLMs) are trained end-to-end for autonomous HDL design. | Full integration: EDA tools and MLM co-execute autonomous verification and optimization. |

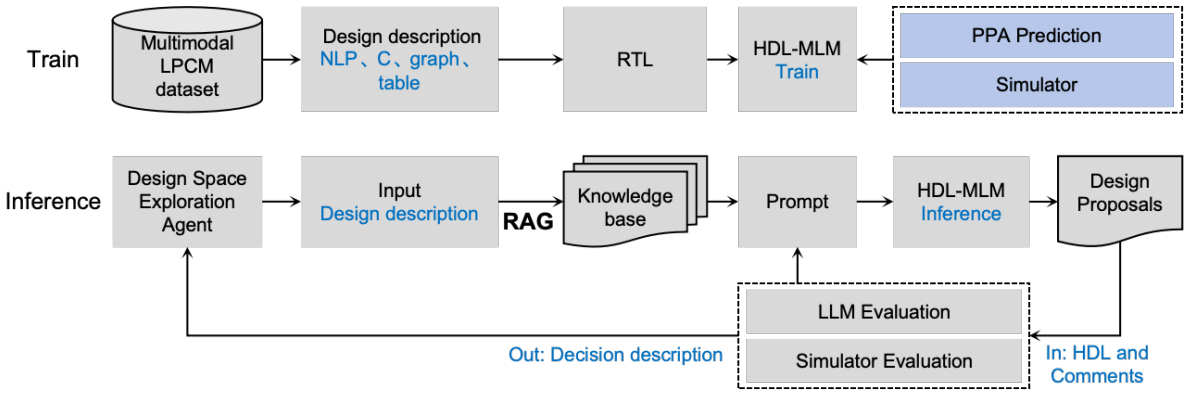


**Figure 9** HDL Level 3 workflow: MLM-Governed

#### 7.2.2 *Related work*

Direct Prompting and Feedback-Driven Optimization. Prior research has explored the direct use of LLMs to generate HDL code. For instance, Chip-chat [86] designed an 8-bit accumulator-based microprocessor using commercial LLMs. Other studies [87, 88] investigate prompt optimization through feedback from EDA tools. AutoChip [87] leverages error reports from compilers and simulators to help LLMs correct faulty code, while MEMV [88] employs a Monte Carlo tree-search algorithm to improve the correctness and PPA efficiency of generated code based on feedback from compilers and synthesis tools. OriGen [89] uses a self-reflection mechanism guided by compiler feedback to fix syntactic errors in the generated code.

Fine-Tuning Strategies. VeriGen [90] performs continual pre-training by predicting the next token on corpora from open-source code and textbooks. VerilogEval [91] applies supervised fine-tuning (SFT) using synthetic instruction-code pairs and releases an open-source evaluation dataset containing 156 questions with golden solutions. MEV-LLM [92] fine-tunes LLMs for hardware designs of varying complexity and integrates them into a unified framework. ChipNeMo [21] customizes LLMs for applications, including chatbots, EDA script generation, and bug summarization. RTLCoder [22] introduces a fine-tuning algorithm that evaluates code quality on candidate samples generated by pre-trained LLMs. BetterV [23] optimizes Verilog code generation using generative discriminators, while CodeV [93] constructs a fine-tuning dataset by generating multi-level Verilog code summaries with LLMs.

Agent-based approaches improve HDL generation by integrating simulation feedback throughout the code generation pipeline, covering planning, verification, and iterative refinement. RTLLM [94] evaluates syntax correctness, functional accuracy, and design quality, while ITERTL [95] iteratively optimizes training data using RTL tool feedback. Multi-agent systems [96–98] enhance reliability through specialized agents, such as those handling RTL coding, testbench generation, validation, and debugging, collaborating in a recursive framework to produce optimized implementations.

### 7.3 Proposed MLM-governed HDL module design

The MLM-HDL model is trained using chip design data generated by upstream modules. The training data consists of natural language descriptions, circuit diagrams, tables, and C code, paired with the corresponding HDL code. The architecture simulator provides supervisory signals to fine-tune the multimodal model, enabling it to incorporate the physical meaning of chip designs.

During inference, the model accepts design parameters from the DSE Agent and generates optimized HDL code. This output undergoes verification through EDA tools, with simulation-validated feedback enabling continuous design refinement. The system achieves complete automation from high-level specifications to production-ready HDL implementation.

Key capabilities include:

• Automated Design Process. The system achieves a fully autonomous design loop where large models can generate complete HDL code from high-level descriptions including functional requirements and performance metrics. This enables end-to-end implementation from initial concept to final physical design without human intervention. The models perform cross-layer optimization by integrating simulation tools, verification platforms, and synthesis tools to ensure the generated code meets specific design constraints

such as power consumption, performance, and area requirements. Through hardware-software co-design, they maintain consistency across design layers while achieving overall performance optimization.

• Efficient Quality Assurance Based on RTL-Level PPA Prediction. We propose a multimodal PPA prediction model for RTL-stage design, integrating both LLMs and graph neural networks (GNNs). First, we leverage fine-tuned LLMs to encode high-level functional and structural information directly from RTL code, focusing in particular on register and critical logic descriptions. Meanwhile, we map the synthesized netlist into a standard-cell graph, allowing a GNN to capture granular structural and timing dependencies. Through methods such as adaptive aggregation and two-phase propagation, the GNN efficiently models local circuit behaviors, while global functionality insights come from the LLM, achieving more accurate PPA predictions even as circuit size grows. On top of this multimodal fusion, our training framework adopts knowledge distillation from a layout-aware "teacher" model to guide the RTL-level "student" model toward near sign-off precision on key metrics like arrival time and power. By aligning node-, subgraph-, and global-level features, the student progressively assimilates timing-critical and layout-specific knowledge. This strategy not only narrows the accuracy gap between RTL and post-layout predictions but also preserves efficiency through a lightweight GNN-based student, making it highly practical for early-stage design exploration.

• Precision Control and Optimization. A feedback-based learning mechanism enables continuous improvement by incorporating EDA toolchain feedback. When initial designs fail to meet PPA standards, the models dynamically adjust parameters or architectural choices to regenerate optimized HDL code. The system also generates innovative design proposals by incorporating cutting-edge research, such as creating HDL implementations for emerging memory technologies or quantum computing architectures.

• Efficient Collaboration Under Human Guidance. The process maintains goal-oriented design where human designers set high-level objectives like "design a low-power IoT processor" while the model handles implementation details. This approach supports rapid iteration and prototype development, allowing designers to quickly test concepts and refine versions, significantly accelerating time-to-market while enhancing design quality and innovation potential.

In conclusion, HDL-MLMs are transforming HDL generation and optimization through domain expertise integration, workflow automation, and feedback-driven learning, revolutionizing traditional design methodologies.

## 8 Put Them All Together

As shown in Figure 10, the framework accepts two input types: binary applications or high-level design specifications, both of which generate an intermediate target code representation.

The SW/HW Partitioning Agent analyzes this representation to determine optimal hardware-software boundaries while respecting the area and power constraints. This agent works alongside the Compiler Agent, which generates appropriate compiler implementations and processes code according to the specified ISA. The rewritten code then enters the design space exploration phase conducted by parallel CPU and Co-Processor DSE Agents. These agents work in tandem with the Simulator Agent, continuously refining architectural proposals through iterative evaluation.

Once an optimal SoC design emerges, the HDL Generation Agent produces RTL code, which undergoes further refinement by the PPA Prediction & Code Optimization Agent. The framework ultimately delivers four key outputs: ISA specifications, compiler implementations, firmware, and the physical SoC.

### 8.1 Gradual LLM Integration

The LPCM framework implements a graduated approach to integrating LLMs into the system architecture design process, progressing through three distinct levels of automation and integration. Each level represents an advancement in the role, capabilities, and autonomy of LLMs within the design workflow.

• **Level 1: Human-Centric with LLM Assistance.** At this level, human designers remain the primary decision-makers, with LLMs functioning as assistive tools. Experts define design requirements, make architectural decisions, and execute core design tasks, while LLMs provide supplementary support such as code generation, documentation assistance, and reference information retrieval. The workflow begins with human designers establishing clear specifications and constraints. LLMs then assist by generating code snippets, suggesting optimization approaches, or retrieving relevant examples from prior

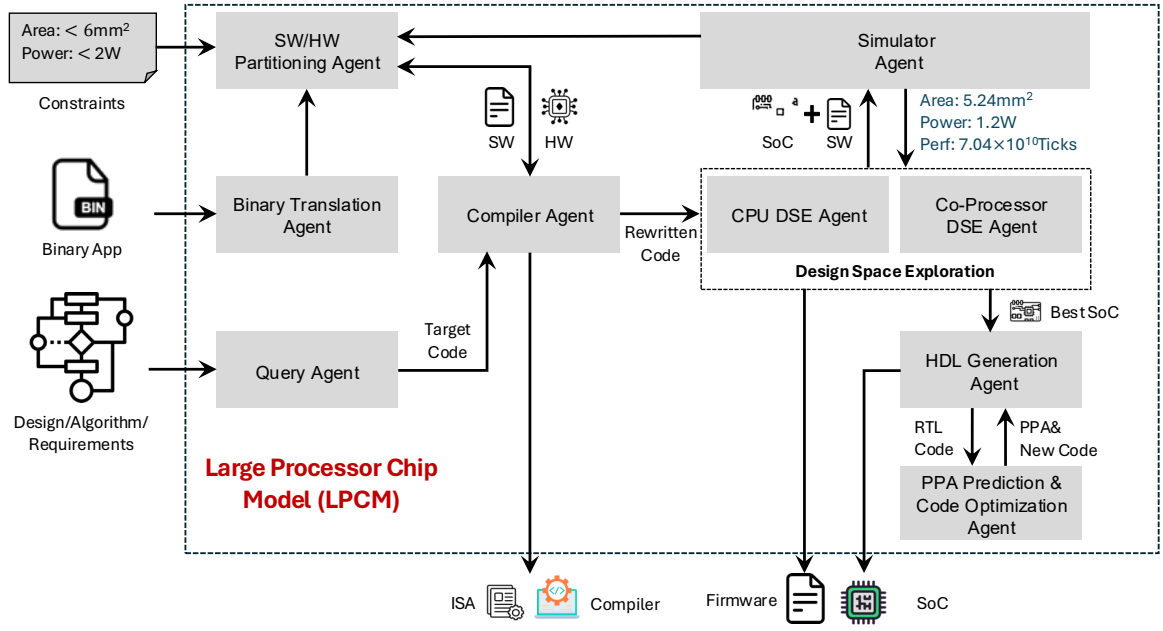


**Figure 10** The end-to-end LPCM flow

designs. Human experts carefully review, modify, and integrate all LLM-generated content into the overall design. Verification and validation processes remain entirely under human control, with LLMs contributing only to specific subtasks within well-defined boundaries. LLMs focus on enhancing designer productivity rather than making autonomous decisions.

• **Level 2: Agent-Orchestrated with Multi-submodule Cooperation.** This level represents a significant advancement in automation, where LLM-based agents coordinate activities across multiple design modules with reduced human intervention. These agents possess greater domain-specific knowledge and can handle routine decision-making tasks autonomously while still operating within frameworks established by human designers. Each module incorporates specialized agent capabilities tailored to its domain. For example, in the Compiler module, agents can autonomously perform code analysis, optimization selection, and transformation application. In the Hardware/Software Partitioning module, agents can identify code segments amenable to hardware acceleration and propose partitioning strategies. These module-specific agents coordinate through standardized interfaces, exchanging structured information and collaborating to solve cross-domain challenges. The interaction becomes bidirectional—LLMs analyze performance feedback and propose design adjustments, while experts refine constraints or objectives as needed.

• **Level 3: LLM-Governed Autonomous Generation.** This level represents a transformative shift where LLMs assume primary control over the entire design process. Humans provide only high-level objectives and constraints, while LLMs autonomously execute the complete design workflow from requirements analysis through implementation. The LLM-governed system exhibits advanced reasoning capabilities, domain expertise across the entire system stack, and the ability to make sophisticated trade-offs between competing design objectives. The workflow begins with minimal human input, typically a concise description of functional requirements and design constraints. The system then autonomously decomposes these requirements into specific tasks, coordinates execution across modules, evaluates multiple design alternatives, and progressively refines solutions based on performance feedback. It can identify and resolve conflicts between requirements, explore unconventional design approaches, and provide detailed explanations for its design decisions, achieving a high degree of autonomy in hardware design.

## 8.2 Case Study: 3D Gaussian Splatting

This section demonstrates the effectiveness of LPCM through a practical application to 3D Gaussian Splatting (3DGS), an emerging rendering technique that has garnered significant attention for its ability to deliver high-quality visual experiences with improved performance. 3DGS represents scenes using a collection of 3D Gaussian primitives, providing a balance between quality and rendering speed that is

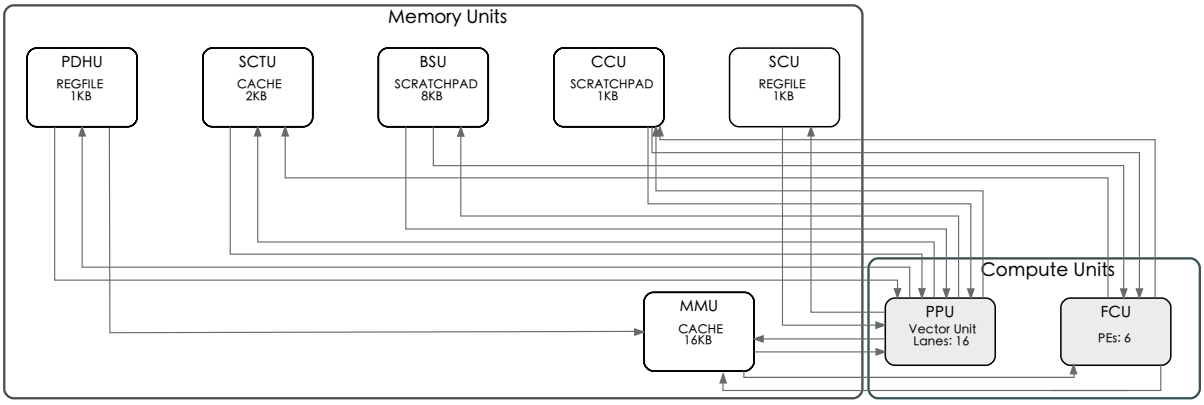


**Figure 11** The 3DGS accelerator designed by the LPCM flow

crucial for applications in virtual reality, scientific visualization, and computer graphics.

The workflow begins with two parallel paths of input processing. While the Binary Translation Agent can process executable applications, in this case study, we focus on the path through the Query Agent. The Query Agent conducts a comprehensive search across multiple GitHub repositories to identify various implementations of the 3DGS rendering algorithm. During this process, the agent evaluates multiple candidate codebases, comparing their quality metrics such as implementation completeness, code efficiency, and documentation quality. After thorough analysis and comparison, the agent selected the implementation from the repository cited in [99] as the optimal choice due to its alignment with the design objectives. This particular implementation stood out for its comprehensive feature set and platform versatility, offering a complete implementation that could run efficiently on CPU architecture rather than being highly platform-dependent.

Simultaneously, the framework receives design constraints specifying requirements of area less than $6\text{mm}^2$ and power consumption under 2W, which directly inform the SW/HW Partitioning Agent's decision-making process.

The Target Code from the Query Agent is then passed to the Compiler Agent, which compiles it for the target CPU platform, generating an executable binary with comprehensive debug information. This debug-enabled binary preserves the algorithmic essence while enabling detailed runtime analysis. Then the SW/HW Partitioning Agent uses this instrumented executable to identify acceleration candidates, recognizing pixel coordinate transformations, conic term calculations, and power sum computations as prime hardware acceleration targets, while maintaining control logic in software. The resulting design specifies clear interfaces between software and hardware components.

Based on this partitioning, the Compiler Agent generates optimized code for both software execution and hardware implementation. This involves producing rewritten code that incorporates extended instructions to access hardware-accelerated components. The compiler specifically creates custom instruction extensions that allow the CPU to efficiently communicate with and offload computations to the specialized co-processor units. In addition, this stage also outputs detailed specifications for these extended instructions that define their functionality and interface, and a modified compiler toolchain that supports the new extension to the instruction set.

The Design Space Exploration Agent, comprising the Co-Processor DSE Agent and CPU DSE Agent, evaluates various architectural configurations to optimize performance under the given constraints. For 3DGS, this exploration focuses on finding the optimal balance between computational units, memory hierarchy, and the interconnect structure. The exploration process evaluates different configurations of CPU and co-processor choices and their parameters, and memory sizes to maximize rendering throughput while meeting the area and power constraints.

Following the identification of the optimal architecture, the HDL Generation Agent translates the architectural specification into RTL code that describes the hardware implementation. For the 3DGS accelerator, this includes generating descriptions for specialized units such as the Pixel Processing Unit with its 16-lane vector unit, the Feature Computation Unit with its 6 processing elements, and memory management units with their respective storage capacities, as shown in Figure 11.

Finally, the PPA Prediction & Code Optimization Agent refines the implementation to ensure it meets the target constraints. For our 3DGS case, the final design achieves an area of $5.24\text{mm}^2$ and power

consumption of 1.2W, comfortably within the specified constraints. Performance analysis shows a 1.41× speedup over a high-performance NVIDIA A100 GPU while requiring only a fraction of the chip area.

### 8.3 Challenges and Future Planning

As we advance our Large Processor Chip Model toward higher levels of automation and integration, several significant challenges must be addressed. This section outlines these challenges and our strategies for overcoming them.

#### 8.3.1 *Multi-modal Knowledge Integration and Transfer*

A significant challenge lies in integrating and transferring knowledge across diverse modalities and domains within the LPCM framework. Each module operates on different data representations—including source code, compiler intermediate representations, task graphs, architectural specifications, and hardware description languages—making knowledge sharing inherently difficult. Furthermore, optimization techniques that prove effective in one domain may not directly translate to others without substantial adaptation.

To address this challenge, we are developing advanced multi-modal learning frameworks capable of extracting, representing, and transferring knowledge across domains. Our approach leverages techniques from transfer learning and multi-task learning to identify generalizable patterns and principles that apply across abstraction levels. By training models on paired examples from different domains, such as software implementations and corresponding hardware accelerators.

We are also establishing comprehensive knowledge repositories that capture design patterns, optimization strategies, and architectural templates across domains. These repositories employ structured knowledge representations that facilitate retrieval and application in new design contexts. By explicitly modeling the relationships between patterns in different domains, we can more effectively transfer successful approaches across abstraction boundaries. This knowledge-sharing infrastructure is complemented by continual learning mechanisms that progressively enhance the system's capabilities based on design experience, enabling it to recognize increasingly complex cross-domain opportunities.

#### 8.3.2 *Verification and Trustworthiness of Autonomous Design Decisions*

As the LPCM framework progresses toward higher levels of automation, ensuring the verification and trustworthiness of autonomously generated designs becomes increasingly critical. Traditional verification methodologies that rely heavily on human oversight become impractical as the system assumes greater responsibility for design decisions. Establishing confidence in the correctness, optimality, and robustness of automatically generated architectures presents significant technical and methodological challenges.

Our approach to addressing this challenge incorporates multiple complementary strategies. First, we are integrating formal verification techniques throughout the design flow. These techniques provide mathematical guarantees about specific properties of the generated designs, such as functional correctness, timing compliance, and security characteristics. By formally verifying critical aspects of the design, we can provide stronger assurances than possible with traditional simulation-based approaches. We are particularly focused on compositional verification methods that can scale to complex system architectures by verifying components individually and their interactions systematically.

Second, we are establishing comprehensive validation frameworks that test generated designs against diverse workloads and operating conditions. These frameworks employ systematic testing strategies to explore the design's behavior across nominal and edge cases, identifying potential weaknesses before implementation. By subjecting designs to rigorous validation, we can identify and address potential issues early in the development process, increasing confidence in the final implementation.

Finally, we are developing incremental automation approaches that gradually transfer responsibility from human designers to the autonomous system as confidence in its capabilities increases. This progressive autonomy enables the system to establish a track record of successful designs in limited contexts before tackling more complex challenges. By carefully managing this transition, we can build trust in the system's capabilities while minimizing the risks associated with autonomous design decisions.

#### 8.3.3 *Cross-Layer Optimization in Heterogeneous Design Spaces*

A fundamental challenge in developing an effective LPCM framework lies in achieving true cross-layer optimization across heterogeneous design spaces. Traditional design methodologies typically optimize each architectural layer independently—application, algorithm, compiler, instruction set, microarchitecture, and circuit implementation, leading to suboptimal global solutions. When design decisions at one layer constrain options at another without coordination, the resulting architectures often exhibit inefficiencies that could be avoided through holistic optimization. This challenge is exacerbated in domain-specific architectures, where the interdependencies between software and hardware design choices become increasingly complex and the potential performance gains from cross-layer optimization grow substantially.

The difficulty stems from several factors: first, the design spaces at different layers are inherently heterogeneous, with distinct representations and optimization metrics that complicate joint exploration. Second, the causal relationships between design decisions across layers are often complex and non-obvious, making it difficult to predict how changes at one layer will propagate through the system. Third, the sheer dimensionality of the combined design space makes exhaustive exploration infeasible, necessitating intelligent navigation strategies that can identify promising cross-layer optimization opportunities.

To address this challenge, we are pursuing several innovative approaches. One promising direction involves reinforcement learning finetuning techniques for cross-layer optimization. By formulating the design process as a sequential decision-making problem, we can train RL agents to make coordinated design choices across multiple layers simultaneously. These agents learn to navigate the complex cross-layer design space through experience, discovering optimization strategies that may not be apparent through conventional methods. The RL framework enables explicit modeling of the long-term performance implications of design decisions, allowing the system to make early-stage choices that facilitate more effective optimization at later stages.

Another innovative approach we are exploring is Critique Finetuning for knowledge alignment across architectural domains. This method leverages expert feedback to critically evaluate design decisions and their cross-layer implications, progressively refining the system's understanding of effective design patterns.

## 9 Conclusion

Research in computer system architecture has evolved significantly, spanning from quantum computing to large-scale data centers, characterized by an unprecedented rate of development, broad impact, and deep influence. Nevertheless, current approaches in this field continue to depend heavily on the conventional process of "research design-experimental verification-data analysis," which presents challenges such as inefficiency, variable quality, and difficulties in replicability. The rise of large language models (LLMs), noted for their sophisticated learning, reasoning, and planning capabilities, offers exciting opportunities for reshaping the paradigm of computer system architecture research. In this context, this paper presents a Large Processor Chip Model (LPCM). It organizes its development into three levels and explores various dimensions, including task complexity, model processing capabilities, and trends in technological evolution. This paper further explores the specific realization paths and technical details related to six key aspects: compiler design, binary translation, simulator design, hardware and software partitioning, design space exploration, and RTL design and generation, across various levels. Additionally, this paper utilizes 3D Gaussian Splatting (3DGS) as a representative workload to examine the LPCM design methodology and operational flow at Level 1, emphasizing the benefits of LPCM in improving design efficiency and quality. Finally, this paper provides an in-depth analysis of the challenges and potential solutions related to the development of LPCM. In conclusion, LPCM is poised to transform the current methodologies of computer system architecture design, steering the information infrastructure towards greater intelligence and automation, while paving the way for new advancements in information technology.

**References**


1 Hennessy J L, Patterson D A. A new golden age for computer architecture. Communications of the ACM, 2019, 62: 48–60
2 Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential logic synthesis and formal verification. URL https://people.eecs.berkeley.edu/~alanmi/abc/
3 Edwards T. Open circuit design. URL http://opencircuitdesign.com/
4 The OpenROAD Project. Opensta: A static timing analysis tool. URL https://github.com/The-OpenROAD-Project/OpenSTA
5 Icarus Verilog Development Team. Icarus verilog. URL https://steveicarus.github.io/iverilog/
6 Wolf C, Glaser J. Yosys - a free verilog synthesis suite. URL https://github.com/YosysHQ/yosys
7 Schafer B C, Takenaka T, Wakabayashi K. Adaptive simulated annealer for high level synthesis design space exploration. In: 2009 International Symposium on VLSI Design, Automation and Test, 2009. 106–109
8 Mahapatra A, Schafer B C. Machine-learning based simulated annealer method for high level synthesis design space exploration. In: Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn), 2014. 1–6
9 Brayton R K, Hachtel G D, McMullen C, et al. Logic minimization algorithms for VLSI synthesis, volume 2. 1984, 1984
10 Keutzer K. Dagon: Technology binding and local optimization by dag matching. In: Proceedings of the 24th ACM/IEEE Design Automation Conference, 1987. 341–347
11 Grosnit A, Malherbe C, Tutunov R, et al. Boils: Bayesian optimisation for logic synthesis. In: 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2022. 1193–1196
12 Cheng C K, Kahng A B, Kang I, et al. Replace: Advancing solution quality and routability validation in global placement. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018, 38: 1717–1730
13 Singh K J, Wang A R, Brayton R K, et al. Timing optimization of combinational logic. In: 1988 IEEE International Conference on Computer-Aided Design, 1988. 282–283
14 Ajayi T, Chhabria V A, Fogaça M, et al. Toward an open-source digital flow: First learnings from the openroad project. In: Proceedings of the 56th Annual Design Automation Conference 2019, 2019. 1–4
15 Carrion Schafer B, Wakabayashi K. Machine learning predictive modelling high-level synthesis design space exploration. IET computers & digital techniques, 2012, 6: 153–159
16 Zuluaga M, Krause A, Milder P, et al. " smart" design space sampling to predict pareto-optimal solutions. In: Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, 2012. 119–128
17 Ferianc M, Fan H, Chu R S, et al. Improving performance estimation for fpga-based accelerators for convolutional neural networks. In: Applied Reconfigurable Computing. Architectures, Tools, and Applications: 16th International Symposium, ARC 2020, Toledo, Spain, April 1–3, 2020, Proceedings 16, 2020. 3–13
18 Mirhoseini A, Goldie A, Yazgan M, et al. Chip placement with deep reinforcement learning. arXiv preprint arXiv:2004.10746, 2020
19 He Y, Bao F S. Circuit routing using monte carlo tree search and deep neural networks. arXiv preprint arXiv:2006.13607, 2020
20 Alawieh M B, Li W, Lin Y, et al. High-definition routing congestion prediction for large-scale FPGAs. In: Proceedings of the Asia and South Pacific Design Automation Conference, 2020. 26–31
21 Liu M, Ene T, Kirby R, et al. Chipnemo: Domain-adapted llms for chip design. CoRR, 2023, abs/2311.00176. URL https://doi.org/10.48550/arXiv.2311.00176
22 Liu S, Fang W, Lu Y, et al. Rtlcoder: Outperforming GPT-3.5 in design RTL generation with our open-source dataset and lightweight solution. CoRR, 2023, abs/2312.08617. URL https://doi.org/10.48550/arXiv.2312.08617
23 Pei Z, Zhen H, Yuan M, et al. Betterv: Controlled verilog generation with discriminative guidance. In: Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024, 2024. URL https://openreview.net/forum?id=jKnW7r7de1
24 Wu H, He Z, Zhang X, et al. Chateda: A large language model powered autonomous agent for eda. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2024
25 Chang K, Wang Y, Ren H, et al. Chipgpt: How far are we from natural language hardware design. CoRR, 2023, abs/2305.14019. URL https://doi.org/10.48550/arXiv.2305.14019
26 Fang W, Li M, Li M, et al. Assertllm: Generating and evaluating hardware verification assertions from design specifications via multi-llms. arXiv preprint arXiv:2402.00386, 2024
27 Tsai Y, Liu M, Ren H. Rtlfixer: Automatically fixing rtl syntax errors with large language model. In: Proceedings of the 61st ACM/IEEE Design Automation Conference, 2024. 1–6
28 Nguyen A T, Nguyen T T, Nguyen T N. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015. 585–596
29 Chen X, Liu C, Song D. Tree-to-tree neural networks for program translation. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018. 2552–2562
30 Feng Z, Guo D, Tang D, et al. Codebert: A pre-trained model for programming and natural languages, 2020
31 Wang Y, Wang W, Joty S, et al. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021
32 Zhong M, Lyu F, Wang L, et al. Comback: A versatile dataset for enhancing compiler backend development efficiency. In: The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track, 2024
33 Zhang S, Zhao J, Xia C, et al. Introducing compiler semantics into large language models as programming language translators: A case study of C to x86 assembly. 2024: 996–1011. URL https://aclanthology.org/2024.findings-emnlp.55/
34 Chen Y, Mendis C, Carbin M, et al. Vegen: a vectorizer generator for simd and beyond. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021. 902–914
35 Armengol-Estapé J, O'Boyle M F. Learning c to x86 translation: An experiment in neural compilation. arXiv preprint arXiv:2108.07639, 2021
36 Guo Z C, Moses W S. Enabling transformers to understand low-level programs. In: 2022 IEEE High Performance Extreme Computing Conference (HPEC), 2022. 1–9
37 facebookarchive. Bolt: Binary optimization and layout tool. https://github.com/facebookarchive/BOLT, 2023. Archived on Jul 1, 2023. Part of the LLVM project.
38 Panchenko M, Auler R, Sakka L, et al. Lightning bolt: powerful, fast, and scalable binary optimization. In: Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction, 2021. 119–130
39 Wong W K, Wang H, Li Z, et al. Refining decompiled c code with large language models. arXiv preprint arXiv:2310.06530, 2023
40 Armengol-Estapé J, Rocha R C, Woodruff J, et al. Forklift: An extensible neural lifter. arXiv preprint arXiv:2404.16041, 2024
41 Tan H, Luo Q, Li J, et al. Llm4decompile: Decompiling binary code with large language models. arXiv preprint

arXiv:2403.05286, 2024
42 Binkert N, Beckmann B, Black G, et al. The gem5 simulator. SIGARCH Comput. Archit. News, 2011, 39: 1–7. URL https://doi.org/10.1145/2024716.2024718
43 Lowe-Power J, Ahmad A M, Akram A, et al. The gem5 simulator: Version 20.0+, 2020. URL https://arxiv.org/abs/2007.03152
44 Jiang N, Becker D U, Michelogiannakis G, et al. A detailed and flexible cycle-accurate network-on-chip simulator. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2013. 86–96
45 Bellard F. Qemu, a fast and portable dynamic translator. 2005: 41
46 DeepSeek-AI, Liu A, Feng B, et al. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437
47 Kerbl B, Kopanas G, Leimkühler T, et al. 3d gaussian splatting for real-time radiance field rendering. ACM Transactions on Graphics, 2023, 42. URL https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/
48 Shao Y S, Xi S L, Srinivasan V, et al. Co-designing accelerators and soc interfaces using gem5-aladdin. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016. 1–12
49 Eles P, Peng Z, Kuchcinski K, et al. System level hardware/software partitioning based on simulated annealing and tabu search. Design automation for embedded systems, 1997, 2: 5–32
50 Lo V M. Heuristic algorithms for task assignment in distributed systems. IEEE Transactions on computers, 1988, 37: 1384–1397
51 Holland J H. Genetic algorithms. Scientific american, 1992, 267: 66–73
52 Zou Y, Zhuang Z, Chen H. Hw-sw partitioning based on genetic algorithm. In: Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753), 2004, volume 1. 628–633
53 Van Laarhoven P J, Aarts E H, van Laarhoven P J, et al. Simulated annealing. 1987, 1987
54 Selman B, Gomes C P. Hill-climbing search. Encyclopedia of cognitive science, 2006, 81: 10
55 Ipek E, McKee S A, Caruana R, et al. Efficiently exploring architectural design spaces via predictive modeling. In: J P Shen, M Martonosi, eds., Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, 2006. 195–206
56 Li D, Yao S, Liu Y, et al. Efficient design space exploration via statistical sampling and adaboost learning. In: DAC, Austin, TX, USA, June 5-9, 2016, 2016. 142:1–142:6
57 Wang D, Yan M, Liu X, et al. A High-accurate Multi-objective Exploration Framework for Design Space of CPU. In: DAC'23: 60th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 9 - 13, 2023, 2023
58 Wang D, Yan M, Teng Y, et al. MoDSE: A high-accurate multiobjective design space exploration framework for CPU microarchitectures. 43: 1525–1537
59 Joseph P J, Vaswani K, Thazhuthaveetil M J. Construction and use of linear regression models for processor performance analysis. In: 12th International Symposium on High-Performance Computer Architecture, HPCA-12 2006, Austin, Texas, USA, February 11-15, 2006, 2006. 99–108
60 Lee B C, Brooks D M. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In: J P Shen, M Martonosi, eds., Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, 2006. 185–194
61 Chen T, Guo Q, Tang K, et al. ArchRanker: A ranking approach to design space exploration. In: ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014, 2014. 85–96
62 Xue R, Wu H, Yan M, et al. Multi-objective optimization in cpu design space exploration: Attention is all you need. arXiv preprint arXiv:2410.18368, 2024
63 Bai C, Sun Q, Zhai J, et al. BOOM-Explorer: RISC-V BOOM Microarchitecture Design Space Exploration Framework. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021, 2021. 1–9
64 Zhai J, Bai C, Zhu B, et al. McPAT-Calib: A RISC-V BOOM Microarchitecture Power Modeling Framework. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2023, 42: 243–256
65 Wang D, Yan M, Teng Y, et al. A High-accurate Multi-objective Ensemble Exploration Framework for Design Space of CPU Microarchitecture. In: Proceedings of the Great Lakes Symposium on VLSI 2023, Knoxville, TN, USA, June 5–7, 2023. 379–383
66 Li D, Wang S, Yao S, et al. Efficient design space exploration by knowledge transfer. In: Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016, 2016. 12:1–12:10
67 Li D, Yao S, Wang S, et al. Cross-program design space exploration by ensemble transfer learning. In: S Parameswaran, ed., 2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017, 2017. 201–208
68 Ding Y, Mishra N, Hoffmann H. Generative and multi-phase learning for computer systems optimization. In: S B Manne, H C Hunter, E R Altman, eds., Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019, 2019. 39–52
69 Wang D, Yan M, Teng Y, et al. A transfer learning framework for high-accurate cross-workload design space exploration of cpu. In: 2023 IEEE/ACM International Conference On Computer Aided Design (ICCAD), 2023
70 Xue R, Wu H, Yan M, et al. Metadse: A few-shot meta-learning framework for cross-workload cpu design space exploration. In: DAC'25: 62nd ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 22 - 25, 2025, 2025
71 Eyerman S, Eeckhout L, De Bosschere K. Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors. In: Proceedings of the Design Automation & Test in Europe Conference, 2006, volume 1. 1–6
72 Mariani G, Palermo G, Silvano C, et al. Meta-model Assisted Optimization for Design Space Exploration of Multi-Processor Systems-on-Chip. In: A Núñez, P P Carballo, eds., 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2009, 27-29 August 2009, Patras, Greece, 2009. 383–389
73 Mariani G, Palermo G, Silvano C, et al. Multi-processor system-on-chip Design Space Exploration based on multi-level modeling techniques. In: W A Najjar, M J Schulte, eds., Proceedings of the 2009 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2009), Samos, Greece, July 20-23, 2009, 2009. 118–124
74 Mariani G, Palermo G, Zaccaria V, et al. Design-Space Exploration and Runtime Resource Management for Multicores. ACM Trans. Embed. Comput. Syst., 2013, 13
75 Wang H, Zhu Z, Shi J, et al. An accurate ACOSSO metamodeling technique for processor architecture design space exploration. In: The 20th Asia and South Pacific Design Automation Conference, 2015. 689–694
76 Zaccaria V, Palermo G, Castro F, et al. Multicube Explorer: An Open Source Framework for Design Space Exploration of Chip Multi-Processors. In: M Beigl, F J Cazorla-Almeida, eds., ARCS '10 - 23th International Conference on Architecture of Computing Systens 2010, Workshop Proceedings, February 22-23, 2010, Hannover, Germany, 2010. 325–331
77 Navada S, Choudhary N K, Rotenberg E. Criticality-driven superscalar design space exploration. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, 2010. 261–272
78 Mariani G, Palermo G, Silvano C, et al. An Efficient Design Space Exploration Methodology for Multi-Cluster VLIW

Architectures based on Artificial Neural Networks. In: Proc. IFIP International Conference on Very Large Scale Integration VLSI - SoC, 2008
79 Mariani G, Brankovic A, Palermo G, et al. A correlation-based design space exploration methodology for multi-processor systems-on-chip. In: S S Sapatnekar, ed., Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010, 2010. 120–125
80 Mariani G, Palermo G, Zaccaria V, et al. OSCAR: An Optimization Methodology Exploiting Spatial Correlation in Multicore Design Spaces. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst., 2012, 31: 740–753
81 Mei L, Houshmand P, Jain V, et al. Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators. IEEE Transactions on Computers, 2021, 70: 1160–1174
82 Hong C, Huang Q, Dinh G, et al. Dosa: Differentiable model-based one-loop search for dnn accelerators. In: Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, 2023. 209–224
83 Cong J, Wang J. Polysa: Polyhedral-based systolic array auto-compilation. In: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018. 1–8
84 Wang J, Guo L, Cong J. Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In: The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2021. 93–104
85 Shao Y S, Reagen B, Wei G Y, et al. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. ACM SIGARCH Computer Architecture News, 2014, 42: 97–108
86 Blocklove J, Garg S, Karri R, et al. Chip-chat: Challenges and opportunities in conversational hardware design. In: 5th ACM/IEEE Workshop on Machine Learning for CAD, MLCAD 2023, Snowbird, UT, USA, September 10-13, 2023, 2023. 1–6. URL https://doi.org/10.1109/MLCAD58807.2023.10299874
87 Thakur S, Blocklove J, Pearce H, et al. Autochip: Automating HDL generation using LLM feedback. CoRR, 2023, abs/2311.04887. URL https://doi.org/10.48550/arXiv.2311.04887
88 DeLorenzo M, Chowdhury A B, Gohil V, et al. Make every move count: Llm-based high-quality RTL code generation using MCTS. CoRR, 2024, abs/2402.03289. URL https://doi.org/10.48550/arXiv.2402.03289
89 Cui F, Yin C, Zhou K, et al. Origen:enhancing RTL code generation with code-to-code augmentation and self-reflection. CoRR, 2024, abs/2407.16237. URL https://doi.org/10.48550/arXiv.2407.16237
90 Thakur S, Ahmad B, Fan Z, et al. Benchmarking large language models for automated verilog RTL code generation. In: Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023, 2023. 1–6. URL https://doi.org/10.23919/DATE56975.2023.10137086
91 Liu M, Pinckney N R, Khailany B, et al. Verilogeval: Evaluating large language models for verilog code generation. In: IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023, 2023. 1–8. URL https://doi.org/10.1109/ICCAD57390.2023.10323812
92 Nadimi B, Zheng H. A multi-expert large language model architecture for verilog code generation. CoRR, 2024, abs/2404.08029. URL https://doi.org/10.48550/arXiv.2404.08029
93 Zhao Y, Huang D, Li C, et al. Codev: Empowering llms for verilog generation through multi-level summarization. CoRR, 2024, abs/2407.10424. URL https://doi.org/10.48550/arXiv.2407.10424
94 Lu Y, Liu S, Zhang Q, et al. RTLLM: an open-source benchmark for design RTL generation with large language model. In: Proceedings of the 29th Asia and South Pacific Design Automation Conference, ASPDAC 2024, Incheon, Korea, January 22-25, 2024, 2024. 722–727. URL https://doi.org/10.1109/ASP-DAC58780.2024.10473904
95 Wu P, Guo N, Xiao X, et al. ITERTL: an iterative framework for fine-tuning llms for RTL code generation. CoRR, 2024, abs/2407.12022. URL https://doi.org/10.48550/arXiv.2407.12022
96 ul Islam M, Sami H, Gaillardon P, et al. Aivril: Ai-driven RTL generation with verification in-the-loop. CoRR, 2024, abs/2409.11411. URL https://doi.org/10.48550/arXiv.2409.11411
97 Ho C, Ren H, Khailany B. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. In: T Walsh, J Shah, Z Kolter, eds., AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA, 2025. 300–307. URL https://doi.org/10.1609/aaai.v39i1.32007
98 Zhao Y, Zhang H, Huang H, et al. MAGE: A multi-agent engine for automated RTL code generation. CoRR, 2024, abs/2412.07822. URL https://doi.org/10.48550/arXiv.2412.07822
99 GitHub - mmt-at/diff-gaussian-rasterization: C Kernel — github.com. https://github.com/mmt-at/diff-gaussian-rasterization