中国科学院计算技术研究所
INSTITUTE OF COMPUTING TECHNOLOGY, CHINESE ACADEMY OF SCIENCES

SKLP
处理器芯片全国重点实验室
State Key Lab of Processors, ICT, CAS

QiMeng: Automated Hardware and Software Design for Processor Chip

Di Huang

Institute of Computing Technology
Chinese Academy of Sciences

The Problem of Hardware and Software Design Automation

The automatic design of hardware and software has long been a pursuit of researchers

CHESS MACHINE of the 18th century was actually run by man inside.

A Chess-Playing Machine

Electronic computers can be set up to play a fairly strong game, raising the question of whether they can "think"

by Claude E. Shannon

FOR CENTURIES philosophers and scientists have speculated about whether or not the human brain is essentially a machine. Could a machine be designed that would be capable of "thinking"? During the past decade several large-scale electronic computing machines have been constructed which are capable of something very close to the reasoning process. These new computers were designed primarily to carry out purely numerical calculations. They perform automatically a long sequence of additions, multiplications and other

it was undertaken with a serious purpose in mind. The investigation of the chess-playing problem is intended to develop techniques that can be used for more practical applications.

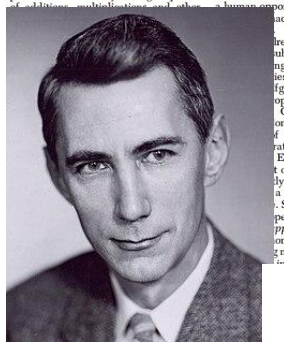
The chess machine is an ideal one to start with for several reasons. The problem is sharply defined, both in the allowed operations (the moves of chess) and in the ultimate goal (checkmate). It is neither so simple as to be trivial nor too difficult for satisfactory solution. And such a machine could be pitted against a human opponent, giving a clear measure of the machine's ability in this type

plicit set of rules can be given for making satisfactory moves in such an end game, the problem is relatively simple, but the idea was quite advanced for that period.

AN electronic computer can be set up to play a complete game. In order to explain the actual setup of a chess machine, it may be best to start with a general picture of a computer and its operation.

A general-purpose electronic computer is an extremely complicated device containing several thousand vacuum tubes, relays and other elements. The basic principles involved, however, are quite simple. The machine has four main parts: 1) an "arithmetic organ," 2) a control element, 3) a numerical memory and 4) a program memory. (In some designs the two memory functions are carried out in the same physical apparatus.) The manner of operation is exactly analogous to a human computer carrying out a series of numerical calculations with an ordinary desk computing machine. The arithmetic organ corresponds to the desk computing machine, the control element to the human operator, the numerical memory to the work sheet on which intermediate and final results are recorded, and the program memory to the computing routine describing the series of operations to be performed.

In an electronic computing machine, the numerical memory consists of a large number of "boxes," each capable of holding a number. To set up a problem on the computer, it is necessary to assign box numbers to all numerical quantities

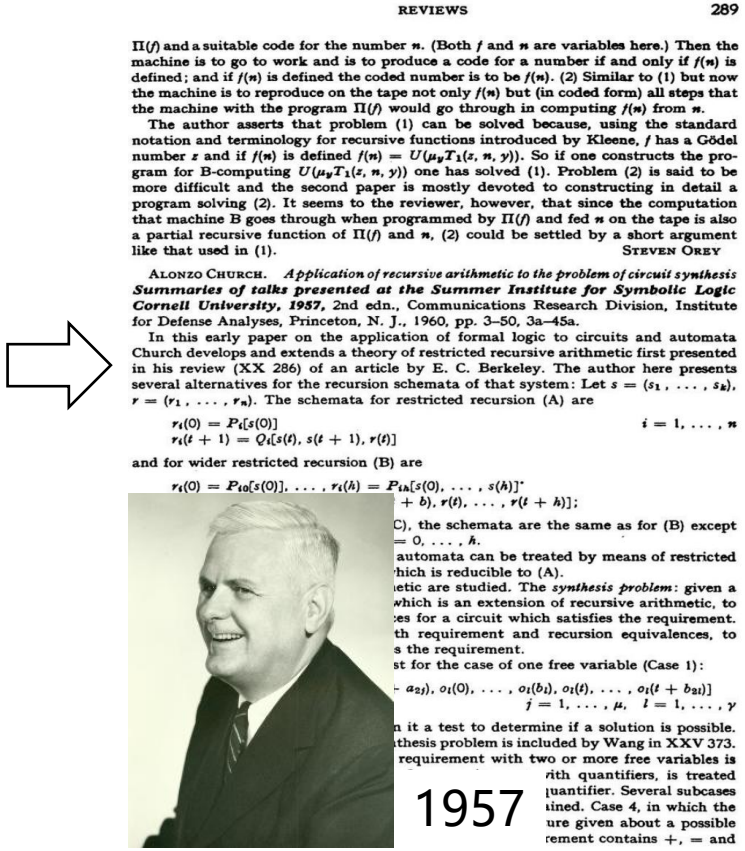


1950

... playing a fair game of chess ...

... translating from one language to another ...

... designing electrical filters and relay circuits ...



REVIEWS 289

$\Pi(f)$ and a suitable code for the number n. (Both f and n are variables here.) Then the machine is to go to work and is to produce a code for a number if and only if $f(n)$ is defined; and if $f(n)$ is defined the coded number is to be $f(n)$. (2) Similar to (1) but now the machine is to reproduce on the tape not only $f(n)$ but (in coded form) all steps that the machine with the program $\Pi(f)$ would go through in computing $f(n)$ from n.

The author asserts that problem (1) can be solved because, using the standard notation and terminology for recursive functions introduced by Kleene, f has a Gödel number z and if $f(n)$ is defined $f(n) = U(\mu_y T_1(z, n, y))$. So if one constructs the program for B-computing $U(\mu_y T_1(z, n, y))$ one has solved (1). Problem (2) is said to be more difficult and the second paper is mostly devoted to constructing in detail a program solving (2). It seems to the reviewer, however, that since the computation that machine B goes through when programmed by $\Pi(f)$ and fed n on the tape is also a partial recursive function of $\Pi(f)$ and n, (2) could be settled by a short argument like that used in (1). STEVEN OREY

ALONZO CHURCH. *Application of recursive arithmetic to the problem of circuit synthesis* ***Summaries of talks presented at the Summer Institute for Symbolic Logic Cornell University, 1957,*** 2nd edn., Communications Research Division, Institute for Defense Analyses, Princeton, N. J., 1960, pp. 3–50, 3a–45a.

In this early paper on the application of formal logic to circuits and automata Church develops and extends a theory of restricted recursive arithmetic first presented in his review (XX 286) of an article by E. C. Berkeley. The author here presents several alternatives for the recursion schemata of that system: Let $s = (s_1, \ldots, s_k)$, $r = (r_1, \ldots, r_n)$. The schemata for restricted recursion (A) are

$$r_i(0) = P_i[s(0)] \qquad i = 1, \ldots, n$$
$$r_i(t+1) = Q_i[s(t), s(t+1), r(t)]$$

and for wider restricted recursion (B) are

$$r_1(0) = P_{10}[s(0)], \ldots, r_1(h) = P_{1h}[s(0), \ldots, s(h)]$$

+ b), r(t), ..., r(t + h)];

C), the schemata are the same as for (B) except
= 0, ..., h.
automata can be treated by means of restricted
hich is reducible to (A).
etic are studied. The *synthesis problem*: given a
which is an extension of recursive arithmetic, to
es for a circuit which satisfies the requirement.
th requirement and recursion equivalences, to
s the requirement.
st for the case of one free variable (Case 1):

$$- a_{2j}), o_i(0), \ldots, o_i(b_i), o_i(t), \ldots, o_i(t + b_{2i})]$$
$$j = 1, \ldots, \mu, \quad i = 1, \ldots, \gamma$$

n it a test to determine if a solution is possible.
thesis problem is included by Wang in XXV 373.
requirement with two or more free variables is
rith quantifiers, is treated
quantifier. Several subcases
ained. Case 4, in which the
ure given about a possible
rement contains +, = and

1957

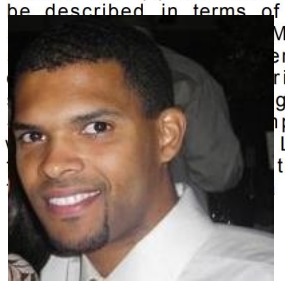
Given input string alpha and output strings beta, is it possible to automatically construct a circuit that satisfies the alpha and beta constraints?

VI. Automatic Programming

A. Introduction

The automatic writing, checking, and debugging of computer programs are problems of great interest both for their independent importance and as useful tools for intelligent machines. This section shows how a theorem prover can be used to solve certain automatic programming problems. The formalization given here will be used to precisely state and solve the problem of automatic generation of programs, including recursive programs, along with concurrent generation of proofs of the correctness of these programs. Thus any programs automatically written by this method have no errors.

We shall take LISP "' as our example of a programming language. In the LISP language, a function is described by two entities: (1) its value, and (2) its side effect. Side effects can be described in terms of their effect upon the Methods for describing erations, as well as methriting of programs in a guage, were presented in plicity, in this section LISP, in which a LISP the standard notion of a value but no side effect.



1969

The automatic writing , checking, and debugging of computer programs are problems of great interest both for their independent importance and as useful tools for intelligent machines.

A Chess Playing Machine. 1950.

Application of recursive arithmetic to the problem of circuit systems. Summaries of the Summer Institute of Symbolic Logic. 1957.

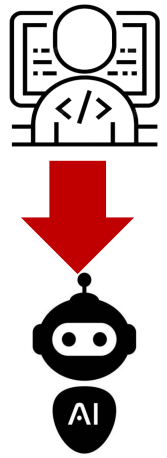
Application of Theorem Proving to Problem Solving. 1969.

Problem Formulation

Problem Statement

Description I/O Verification

Program Design



Computation

Boolean Circuits / Programs

Define the set X of functions and design constraints described in natural language, and the verification condition $R(x, y)$.

Function Description Truth Tables

Construct a mapping $F: X \rightarrow Y$, s.t. $\forall x \in X$, $R(x, F(x))$ $holds$

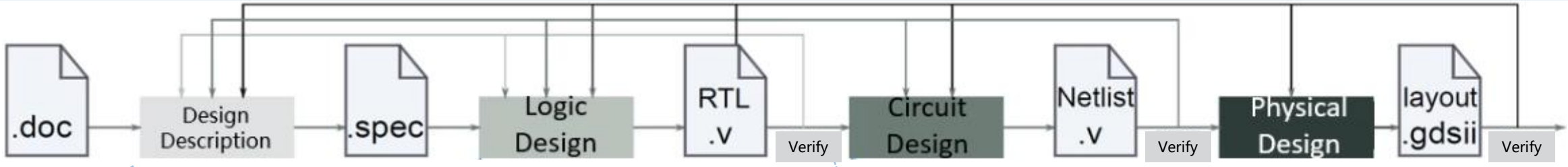
Verilog Chisel

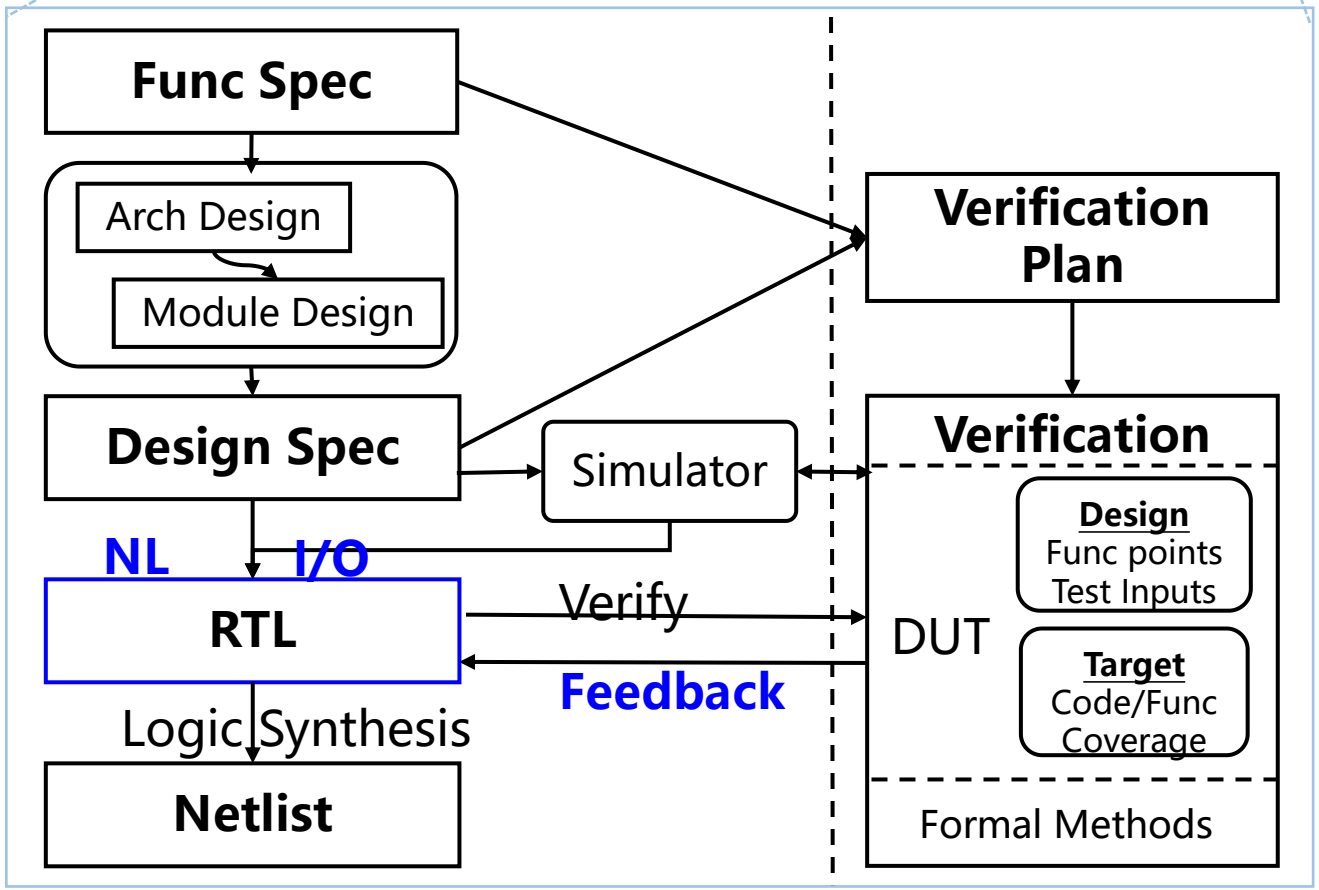
The set of executable operations Y

Basic Boolean Circuits

AND OR NOT

Example: Existing Processor Design Flow



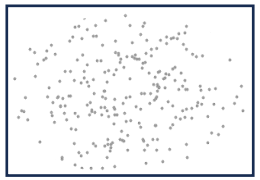


The current processor design flow is overly lengthy and complex. Existing EDA tools can only handle a **single stage** from formal input to formal output, and a large amount of manual involvement is still required in the design process.

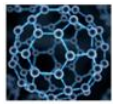
Especially in the process from design documents to RTL code and code fixes based on verification feedback, the work is almost entirely done manually—writing, debugging, and modifying—with little support from EDA tools.

Challenge 1: Large Solution Space

~10^{80} particles



“half universe”

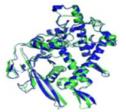
Materials  10^{48}

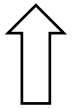
Challenge 1: Large Solution Space

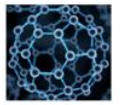
~10^{80} particles

32bit CPU  $10^{10^{540}}$

Drugs  10^{300}

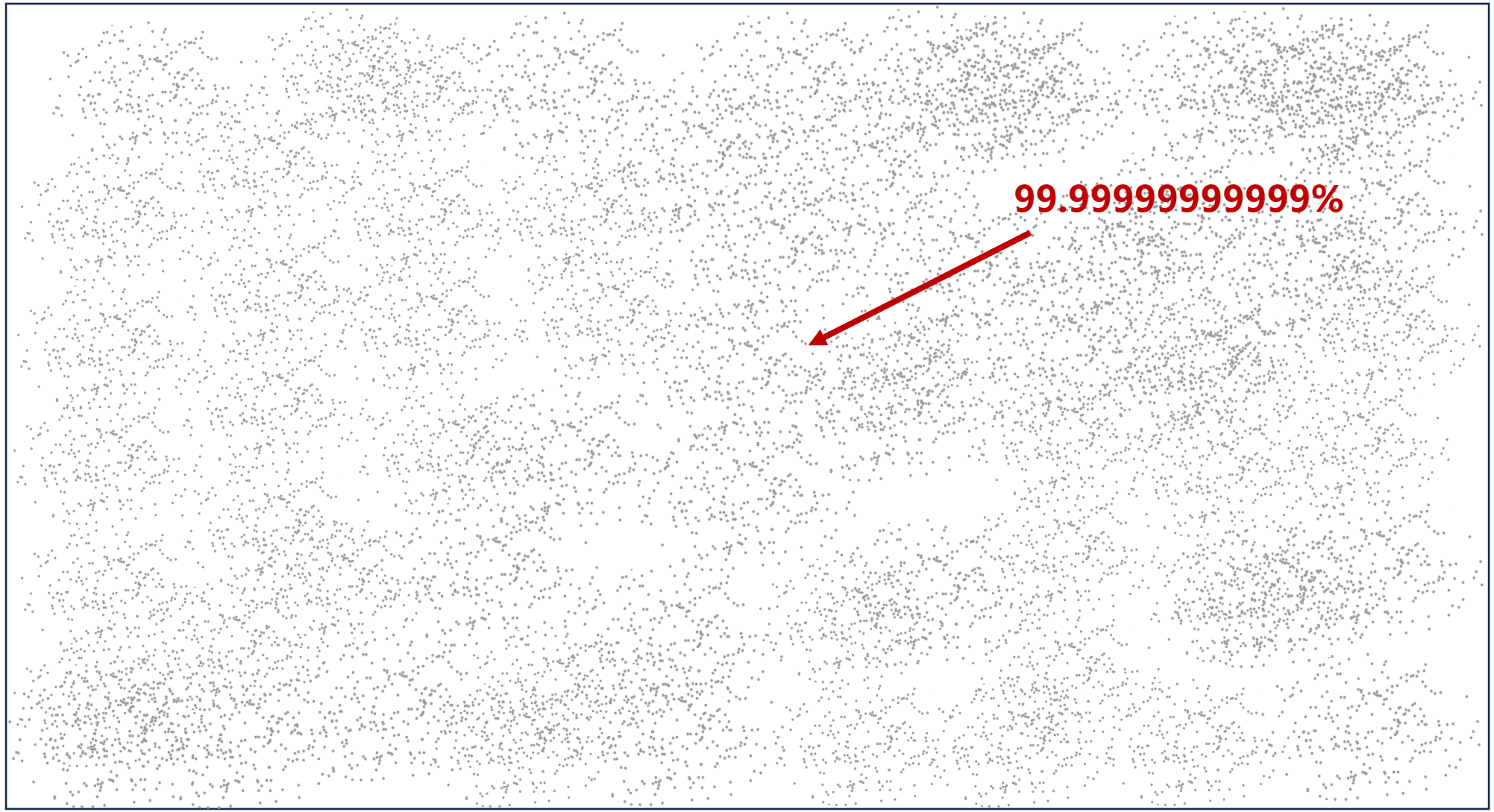
Proteins  10^{125}



Materials  10^{48}

$10^{10^{540}}$ >>  $\sim 10^{10^{82}}$ "universe universe"

Challenge 2: High Accuracy Requirement

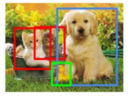


One billion tests (each with kilos instructions) to ensure Intel P4 CPU with a **99.99999999999%** accuracy

VS

CAT

Image classification
~90%



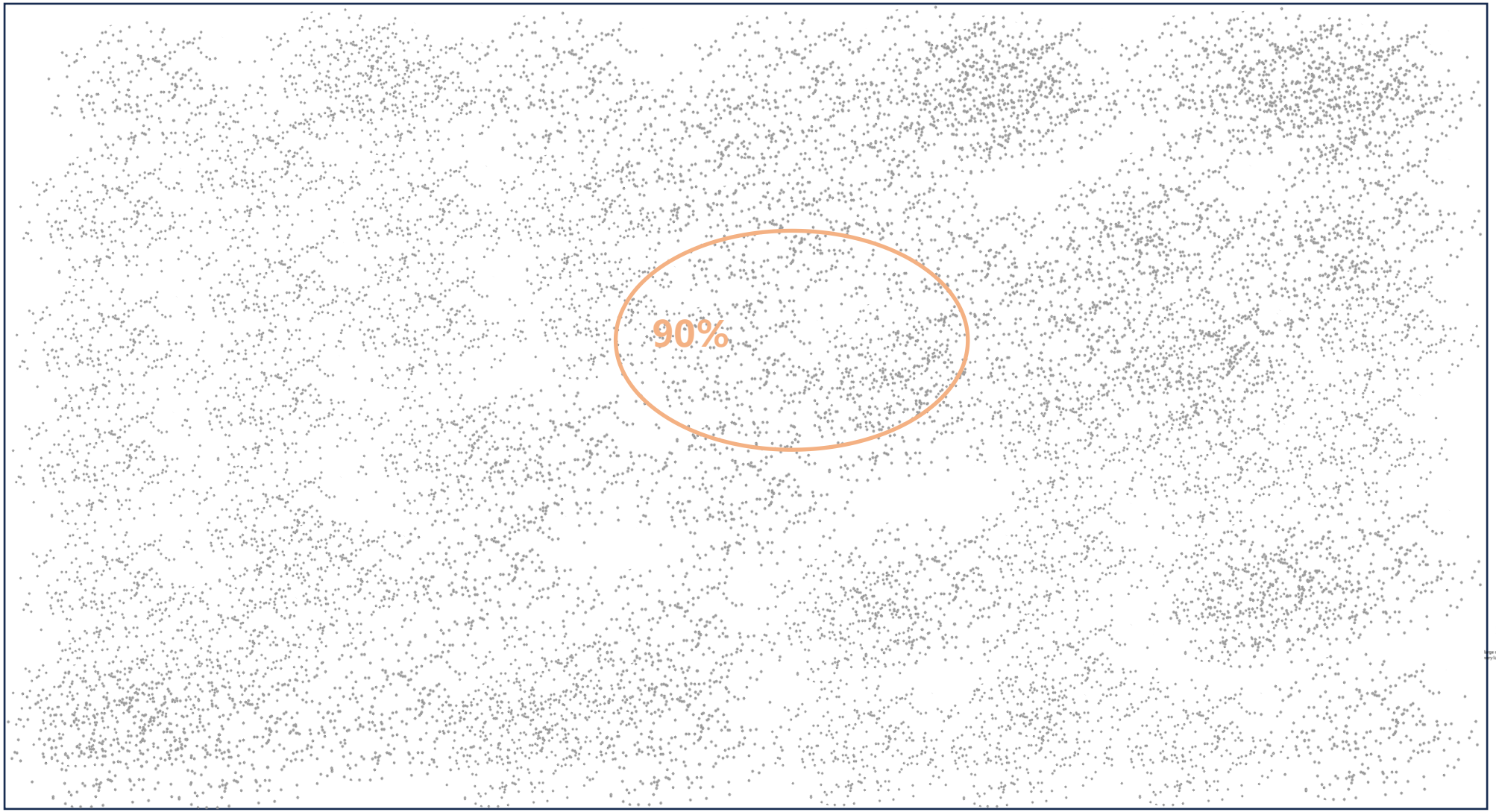
Object detection
~80%

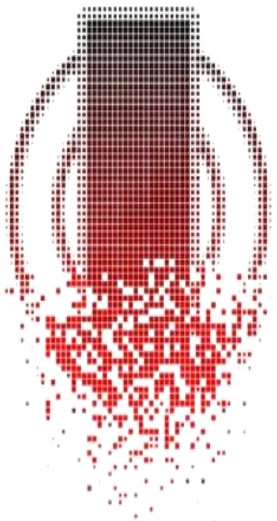
Voice recognition
~90%

QA agent
~90%

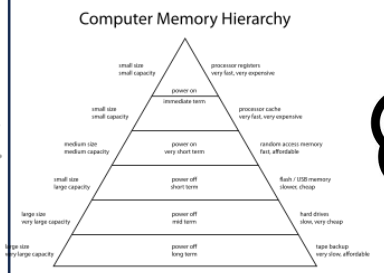
"Find the particle in  "

Our Attempt: Top-down Prior





Top-down Prior

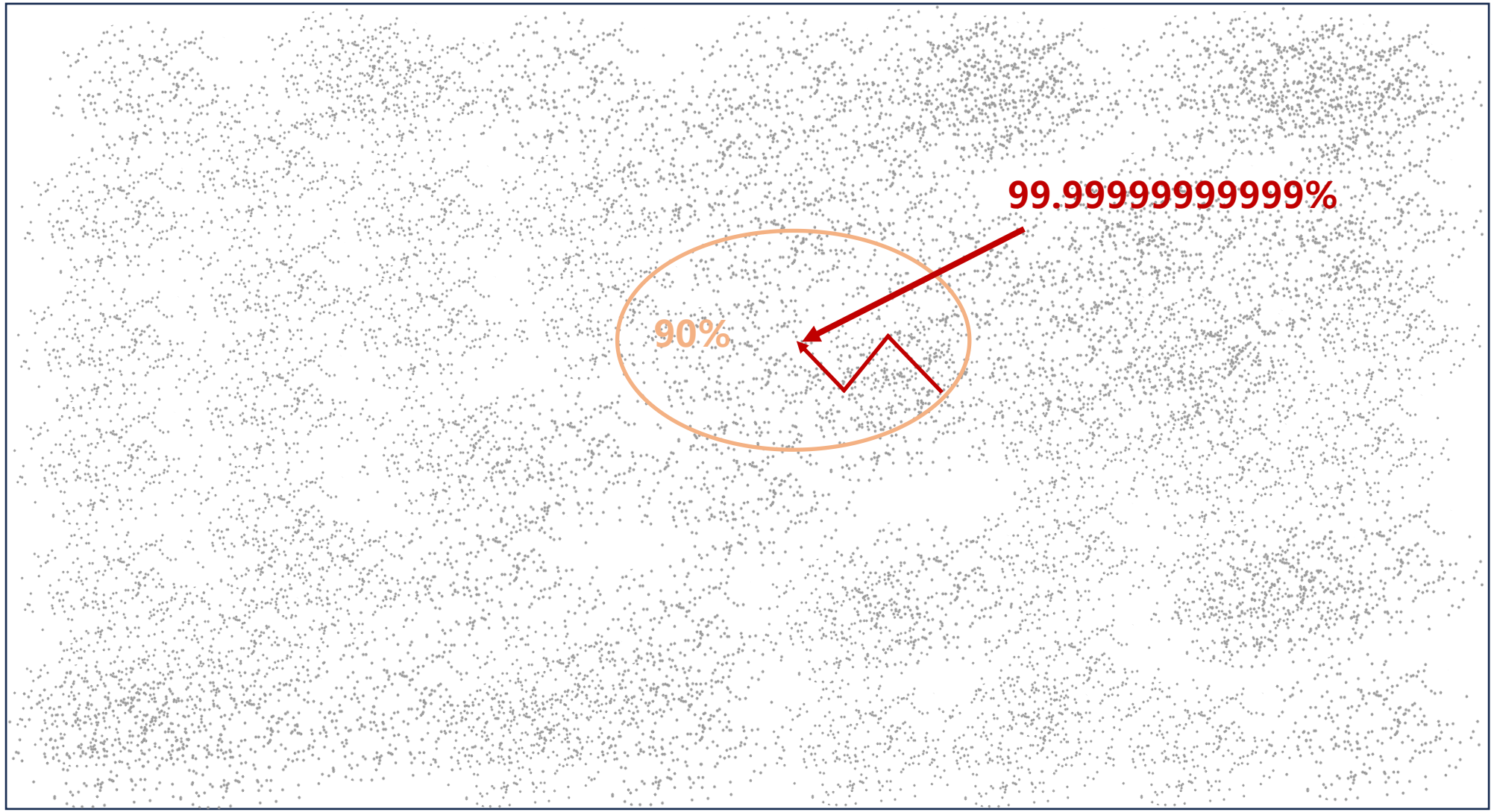


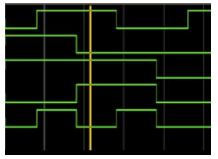
Design Prior

Semantic Prior

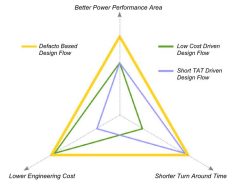
"First, circle a **meaningful** range"

Our Attempt: Bottom-up Feedback





Accuracy



Performance

Bottom-up Feedback

"Then, get the target through **feedback** and **monotonic iteration**."

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

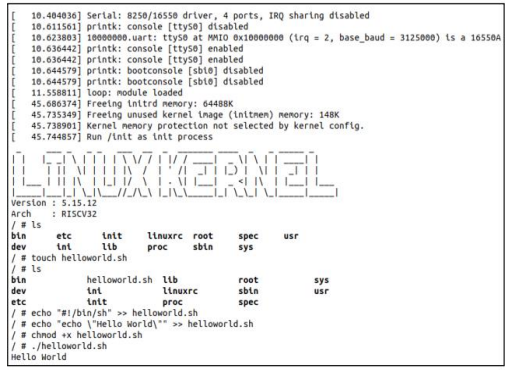
Library	Automated High-Performance Library Generation
Compiler	Automated Tensor Program Transcompiler
	Automated Compiler Tool-Chain Design
OS	Automated OS Configuration Optimization
Processor	Automated HDL Generation
	Automated Front-End Design

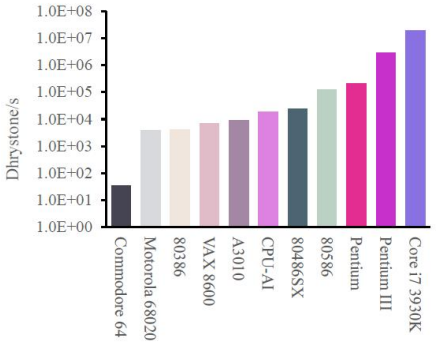
QiMeng-TensorOp [IJCAI 25], QiMeng-Attention [ACL 25]
QiMeng-GEMM [AAAI 25], QiMeng-Xpiler [OSDI 25]
BabelTower [ICML 22], AutoOS [ICML 24]
CodeV , QiMeng-CPU [IJCAI 24, 25]

https://qimeng-ict.github.io

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

Library	Automated High-Performance Library Generation
Compiler	Automated Tensor Program Transcompiler
	Automated Compiler Tool-Chain Design
OS	Automated OS Configuration Optimization
Processor	Automated HDL Generation
	Automated Front-End Design



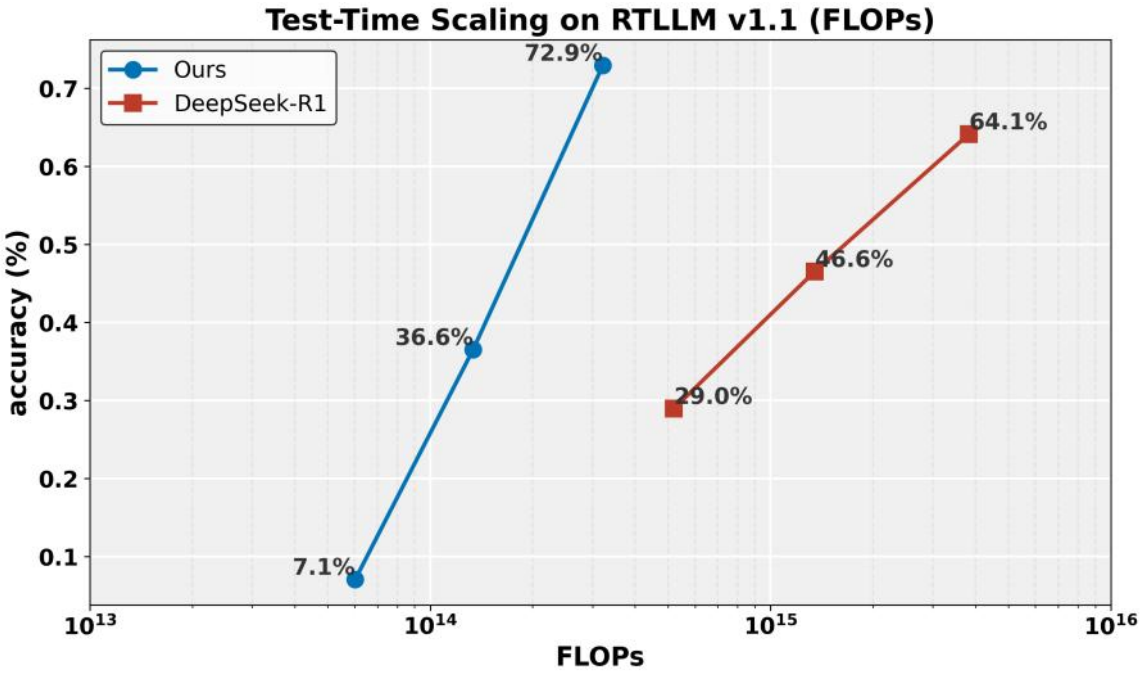


QiMeng-TensorOp [IJCAI 25], QiMeng-Attention [ACL 25]
QiMeng-GEMM [AAAI 25], QiMeng-Xpiler [OSDI 25]
BabelTower [ICML 22], AutoOS [ICML 24]
CodeV , QiMeng-CPU [IJCAI 24, 25]

https://qimeng-ict.github.io

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

Library	Automated High-Performance Library Generation
Compiler	Automated Tensor Program Transcompiler
	Automated Compiler Tool-Chain Design
OS	Automated OS Configuration Optimization
Processor	Automated HDL Generation
	Automated Front-End Design



NL → HDL

QiMeng-TensorOp [IJCAI 25], QiMeng-Attention [ACL 25]
QiMeng-GEMM [AAAI 25], QiMeng-Xpiler [OSDI 25]
BabelTower [ICML 22], AutoOS [ICML 24]
CodeV , QiMeng-CPU [IJCAI 24, 25]

https://qimeng-ict.github.io

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

Library	Automated High-Performance Library Generation
Compiler	Automated Tensor Program Transcompiler
	Automated Compiler Tool-Chain Design
OS	Automated OS Configuration Optimization
Processor	Automated HDL Generation
	Automated Front-End Design

usr: I want to improve the speed of creating processes, how do I modify the linux kernel's configuration options? Please provide specific options directly and give the recommend setting.

chatgpt: Modifying the kernel's configuration options to improve the speed of creating Linux image processes can be a complex task, and it largely depends on the specific use case and requirements ....Keep in mind that changing kernel configurations should be done with caution and with a good understanding of the potential impacts on your system.

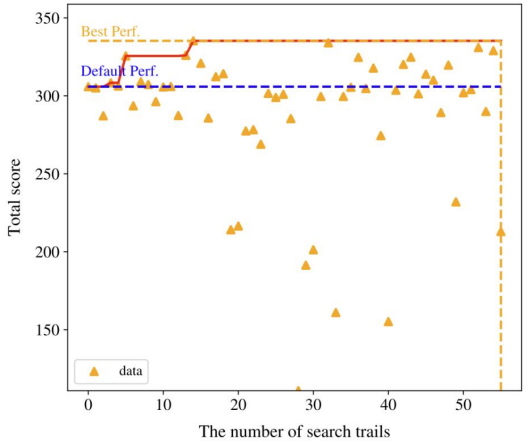
1 CONFIG_BASE_SMALL:
Recommended Setting: y
description: ...
2 CONFIG_PREEMPT:
Recommended Setting: y (for desktop systems), n (for servers)
Description:...
...

7 CONFIG_HZ:
Recommended Setting: 1000
Description:...

```
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=100301
CONFIG_CLANG_VERSION=0
CONFIG_AS_IS_GNU=y
CONFIG_AS_VERSION=23500
CONFIG_LD_IS_BFD=y
CONFIG_LD_VERSION=23500
CONFIG_LLD_VERSION=0
CONFIG_CC_CAN_LINK=y

...

# CONFIG_TEST_MEMINIT is not set
# CONFIG_TEST_FREE_PAGES is not set
CONFIG_ARCH_USE_MEMTEST=y
```

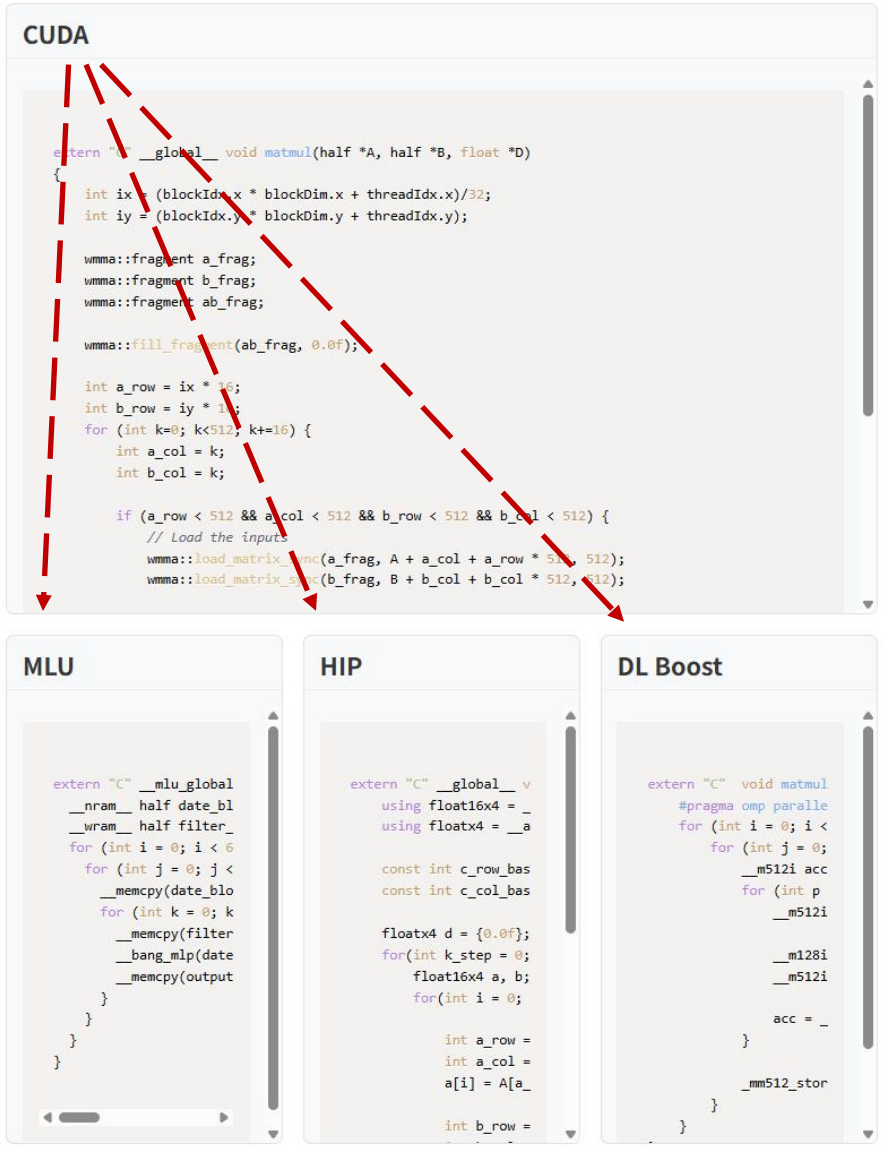


QiMeng-TensorOp [IJCAI 25], QiMeng-Attention [ACL 25]
QiMeng-GEMM [AAAI 25], QiMeng-Xpiler [OSDI 25]
BabelTower [ICML 22], AutoOS [ICML 24]
CodeV , QiMeng-CPU [IJCAI 24, 25]

https://qimeng-ict.github.io

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

Library	Automated High-Performance Library Generation
Compiler	Automated Tensor Program Transcompiler
	Automated Compiler Tool-Chain Design
OS	Automated OS Configuration Optimization
Processor	Automated HDL Generation
	Automated Front-End Design

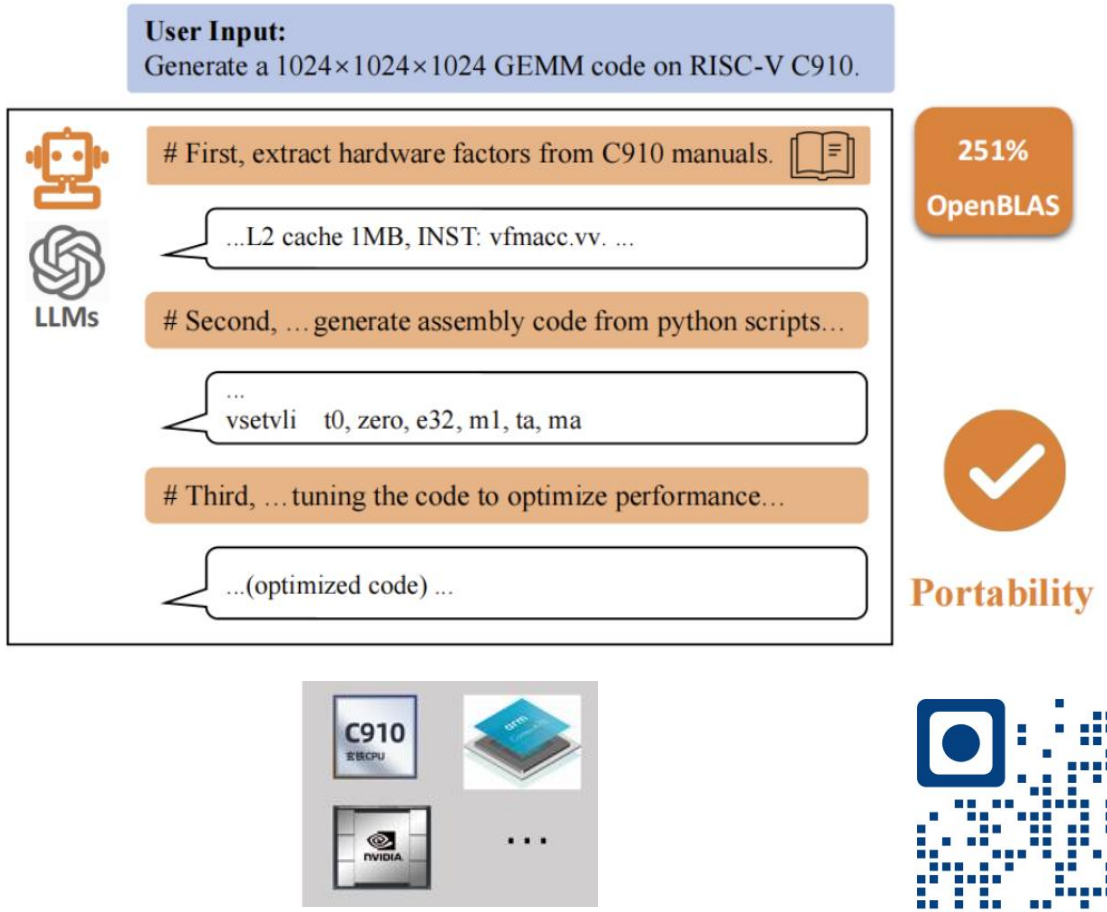


QiMeng-TensorOp [IJCAI 25], QiMeng-Attention [ACL 25]
QiMeng-GEMM [AAAI 25], QiMeng-Xpiler [OSDI 25]
BabelTower [ICML 22], AutoOS [ICML 24]
CodeV , QiMeng-CPU [IJCAI 24, 25]

https://qimeng-ict.github.io

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

Library	Automated High-Performance Library Generation
Compiler	Automated Tensor Program Transcompiler
	Automated Compiler Tool-Chain Design
OS	Automated OS Configuration Optimization
Processor	Automated HDL Generation
	Automated Front-End Design



QiMeng-TensorOp [IJCAI 25], QiMeng-Attention [ACL 25]
QiMeng-GEMM [AAAI 25], QiMeng-Xpiler [OSDI 25]
BabelTower [ICML 22], AutoOS [ICML 24]
CodeV , QiMeng-CPU [IJCAI 24, 25]

https://qimeng-ict.github.io

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

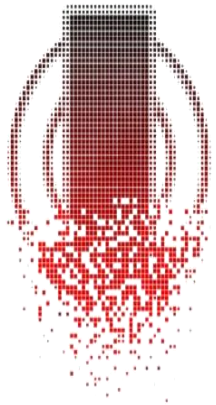
CPU

CodeV

NL → HDL

Xpiler

Cuda → BangC
Cuda → HIP
Cuda → DL Boost



QiMeng: Fully Automated Hardware and Software Design for Processor Chip

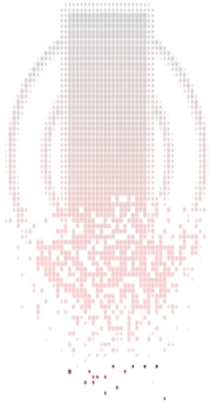
CPU

CodeV

NL → HDL

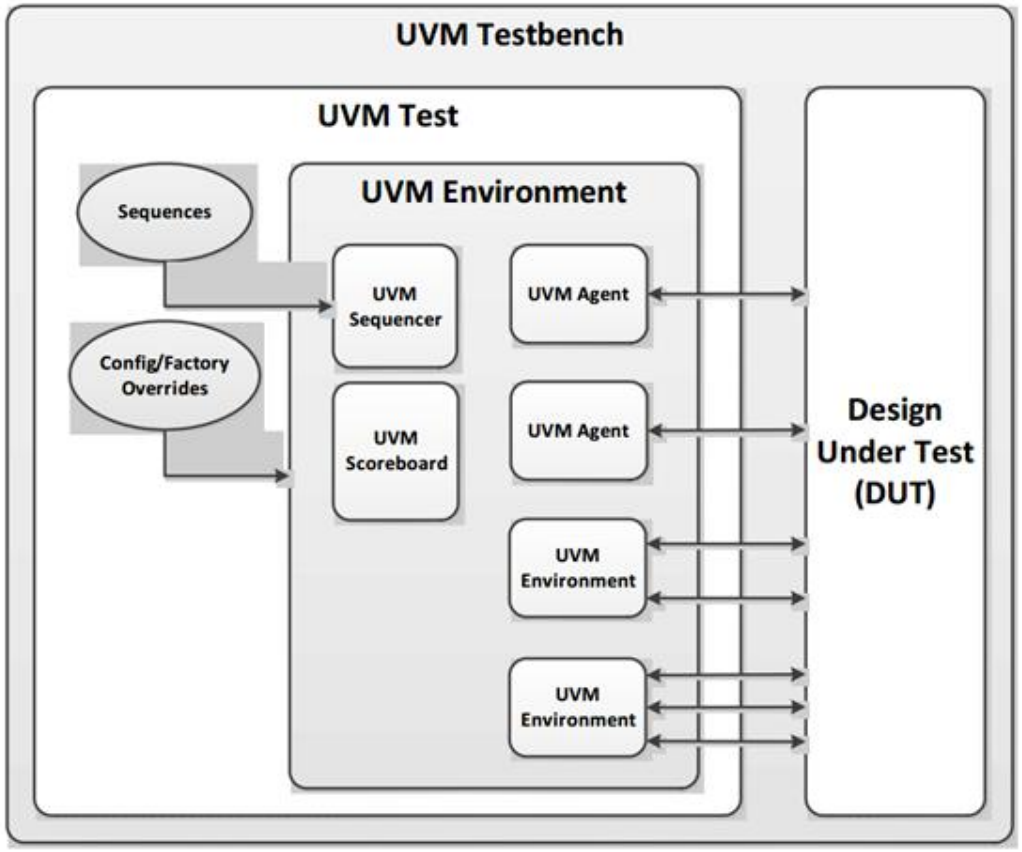
Xpiler

Cuda → BangC
Cuda → HIP
Cuda → DL Boost

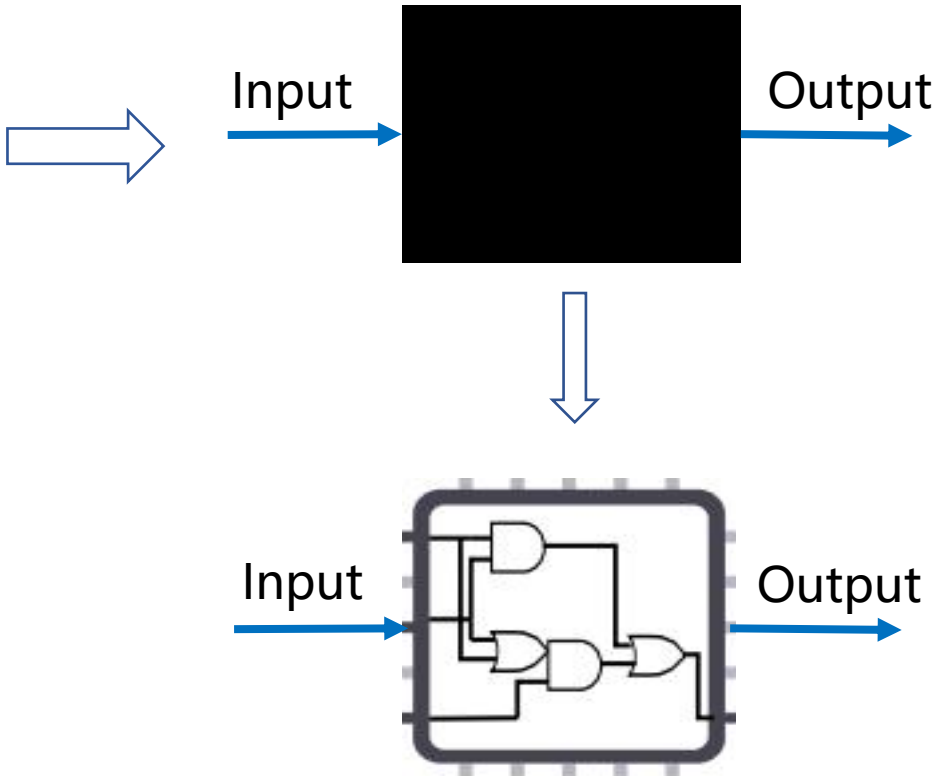


Automatically Generate the CPU from a Finite Set of IOs

Test case for verification

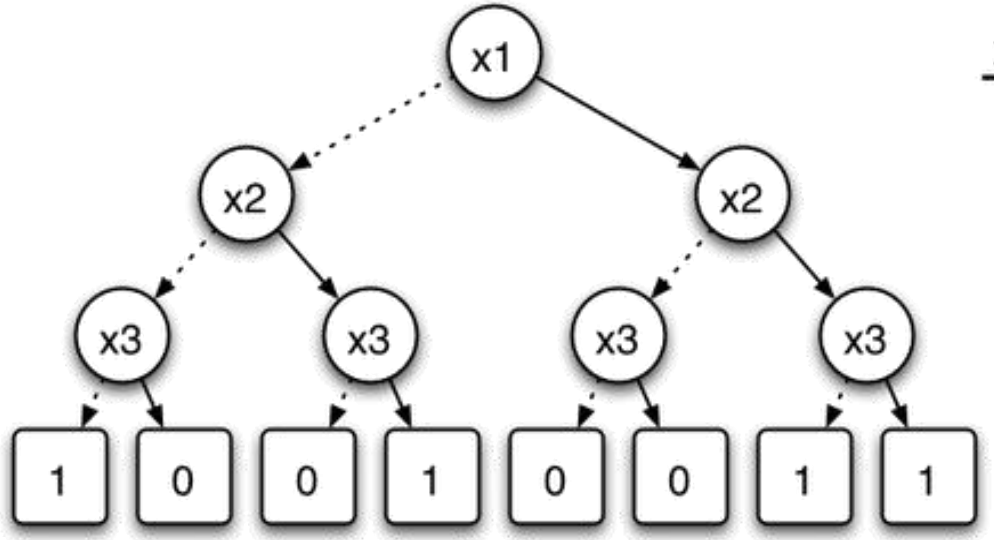


Get I/O spec

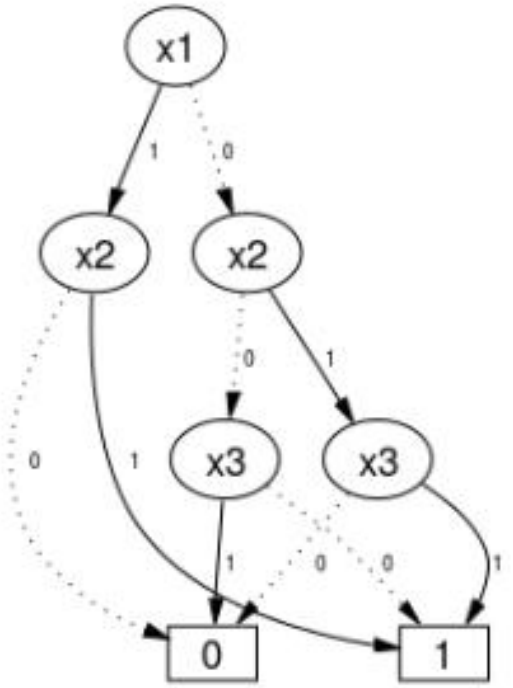


Problem: How to generate **a CPU** given **a finite set of IOs** (truth tables).

Binary Decision Diagram



x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



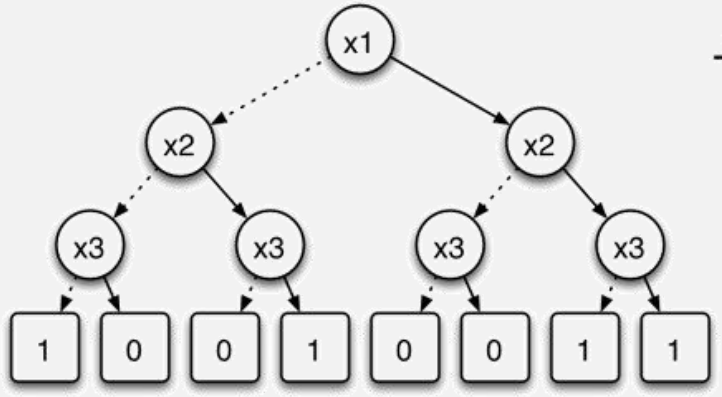
Binary Decision Diagram (BDD):

- A Boolean function can be represented as a rooted, directed, acyclic graph, which consists of several (decision) nodes and two terminal nodes.
- The two terminal nodes are labeled 0 (FALSE) and 1 (TRUE).
- Each (decision) node u is labeled by a Boolean variable and has two child nodes called low child and high child. The edge from node u to a low (or high) child represents an assignment of the value FALSE (or TRUE, respectively) to variable xi.

Binary ~~Decision~~ Speculation Diagram

Problem: How to generate **large-scale and accurate** CPU given **a limited set of IOs**.

BDD (Binary Decision Diagram)

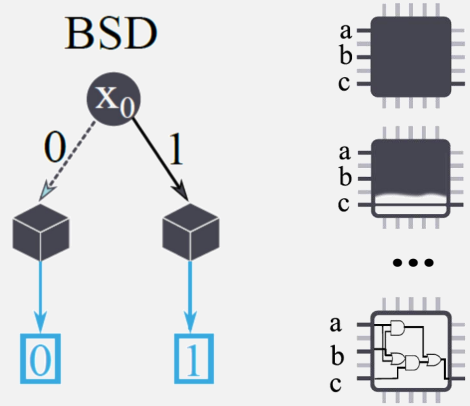


x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

All of truth table, leaf-nodes assigned

VS

BSD (Binary Speculation Diagram)



Partial truth table, leaf-nodes speculated

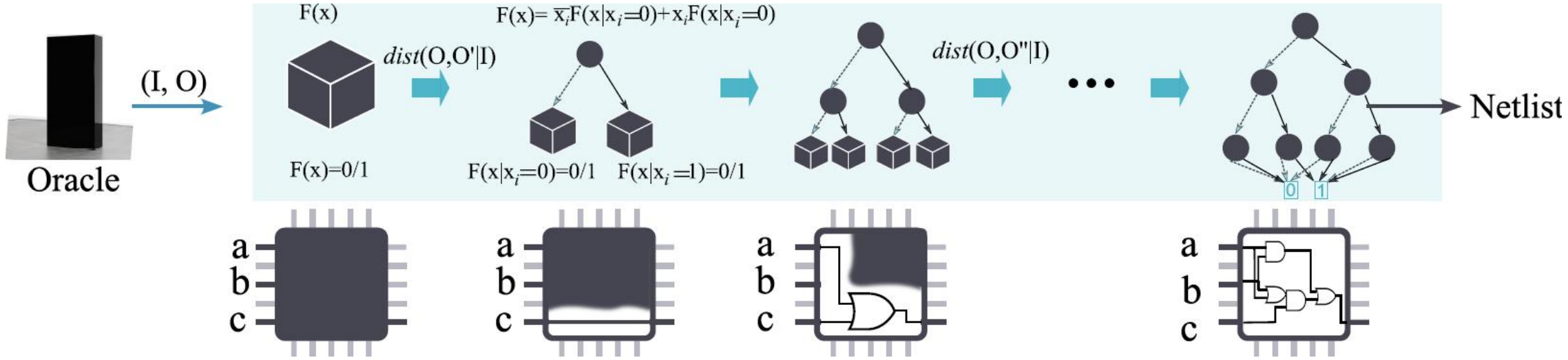
Difficulty: Fitting an almost **infinitely large** truth table.

32-bit RISC-V CPU

(~1800-bit input/output)

$$10^{10^{540}}$$

Reduce the Search Space with Bool Distance



- Boolean Distance: Monte Carlo method to approximately calculate

$$Dist(f,g) = C_\Omega(f) + C_\Omega(g) - C_\Omega(\tau)$$

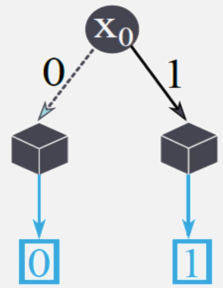
- **Merging the nodes with 0 Boolean Distance to narrow down the space**

$$10^{10^{540}} \Rightarrow 10^6$$

Speculate, Compute, and Verify

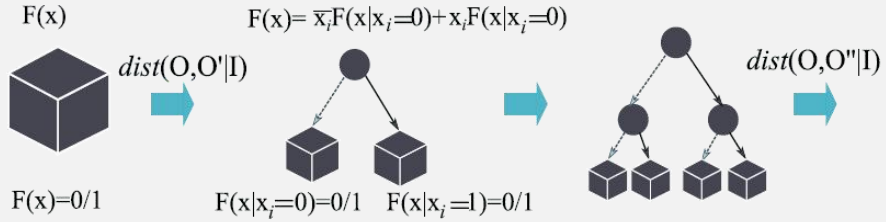
Speculate Circuit Diagram Generation

BSD **(Binary Speculation Diagram)**



MCTS-based leaf nodes speculation

Compute and Verify Accurately pinpoint errors: if one occurs, identify the leaf node and continue sampling and expanding it.



Iteration: Continuously correct errors, and a rigorous theoretical proof demonstrates that by iteratively expanding the guessed leaf nodes, more accurate generation results can be achieved.

Theorem 1 (The accuracy of BSD boosts after expansion). *After expanding the generated BSD $\mathcal{F}_k$ by any input bit x_i to $\mathcal{F}_{k+1}$, the accuracy of expansion ended with $\mathcal{F}_k$ will be no larger than the accuracy of expansion ended with $\mathcal{F}_{k+1}$, that is,*

$$Acc(\mathcal{F}_k) \leq Acc(\mathcal{F}_{k+1}). \quad (3)$$

2021：QiMeng-CPU-v1

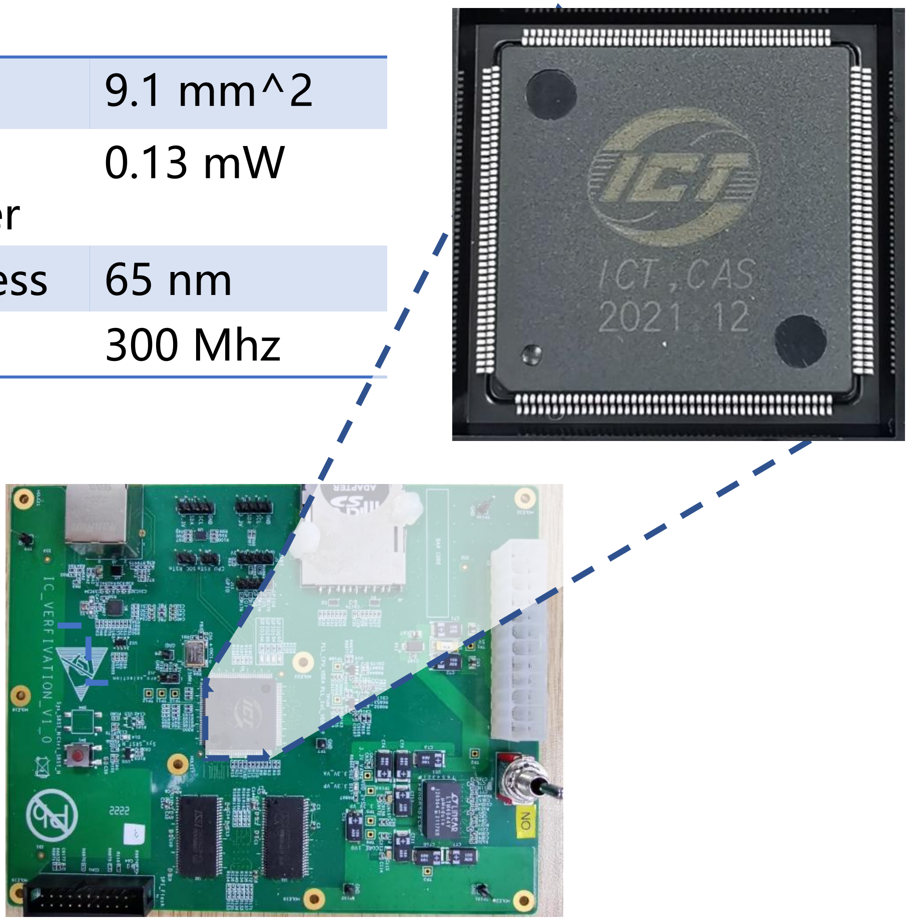
Object	Output	Gates	Team	Method
Adder	Circuit logic	~200 Gates	Nvidia [Roy et al. 2021]	Deep RL
Simple ALU	Circuit logic	~1000 Gates	Univ. TW [Chen et al. 2020]	Decision Tree
Simple ALU	Circuit logic	~2500 Gates	Univ. Tokyo [Rai et al. 2021]	Assemble Learning
8- bit CPU	Circuit logic	~1000 Gates	Univ. NY [Blocklove et al. 2023]	Deep Learning
32- bit CPU	**2021**	**~4,000,000 Gates**	**ICT, CAS**	**Binary Speculation Diagram**

The design scale is increased by thousands of times

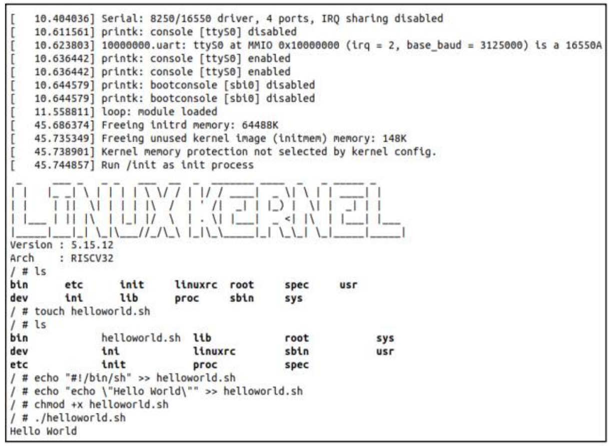
2021: QiMeng-CPU-v1

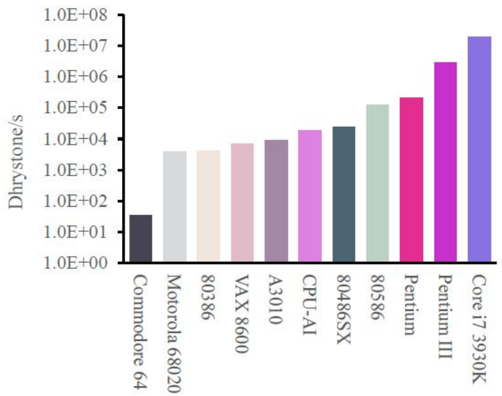
Frontend design of a 32-bit RISC-V CPU (4 million gates) in 5 Hours

Area	9.1 mm^2
Core Power	0.13 mW
Process	65 nm
Freq	300 Mhz



- 2021.12: Tapout of QiMeng-CPU-v1
- 2022.05: testing, running Linux OS and SPEC CPU2000 successfully, on par with Intel 486

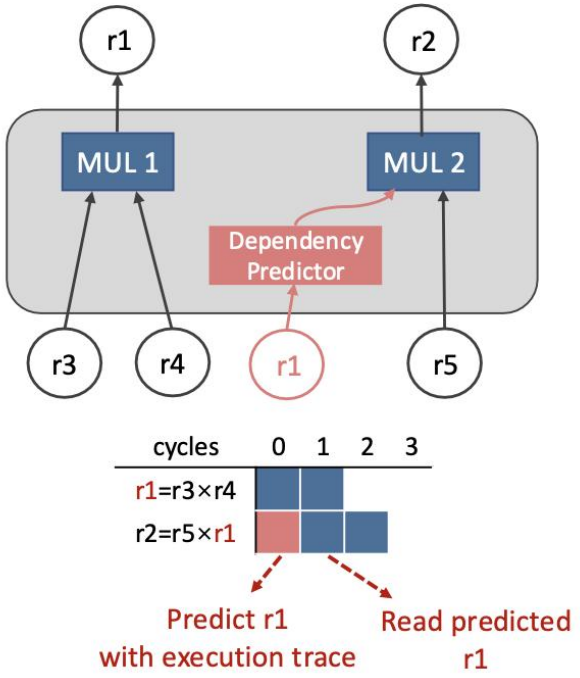




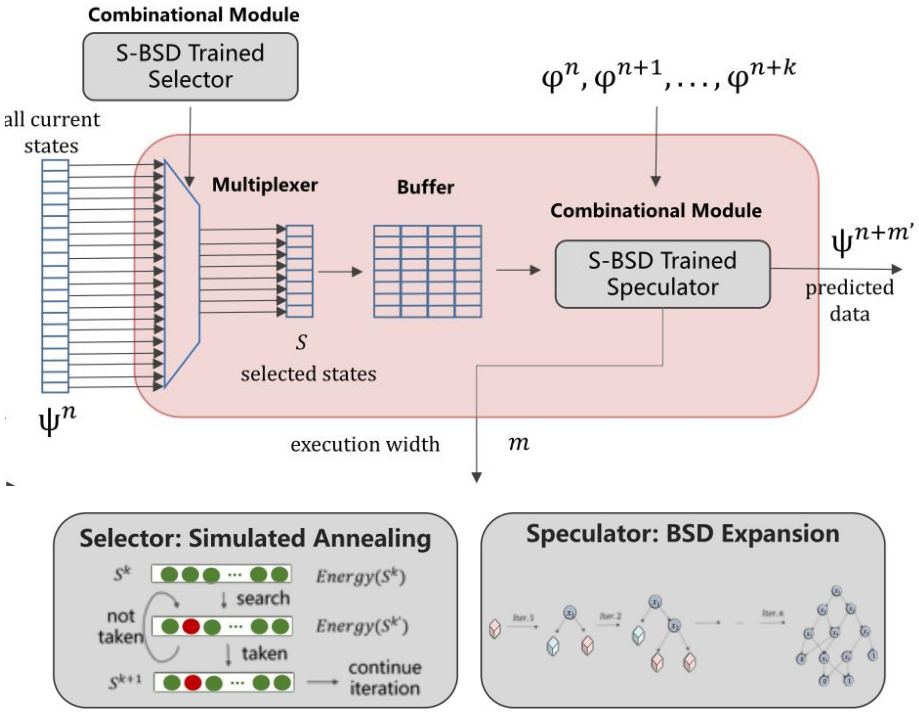
2024: QiMeng-CPU-v2

QiMeng-CPU-v2, an automatically designed **superscalar** CPU core, achieving performance comparable to the **ARM Cortex-A53**.

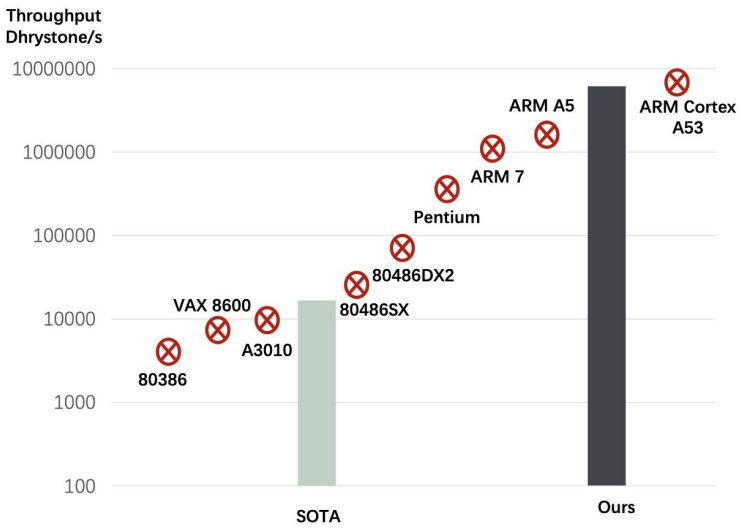
Automated instruction-level parallelism: By introducing internal processor state information, it predicts data dependencies between instructions to achieve parallel execution across instructions.



Decouple inter-instruction data dependencies



Predict dependent data



382× compared to QiMeng-CPU-v1

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

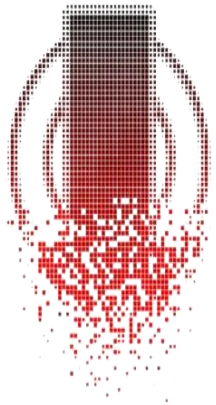
CPU

CodeV

NL → HDL

Xpiler

Cuda → BangC, HIP, DL Boost

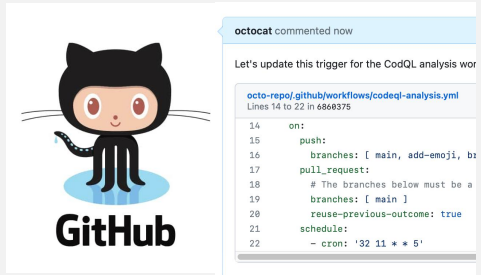


The Challenges of Constructing Prior with LLMs

Data Scarcity

Natural language, video, image
10^3ZB

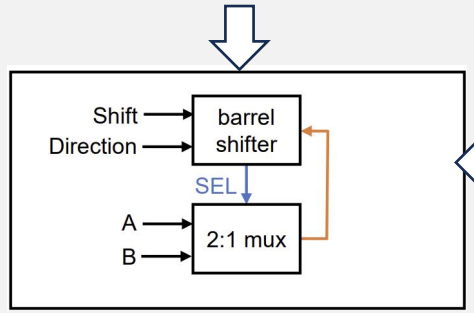
vs



Hardware and Software code
10-100TB

Large Semantic Gap

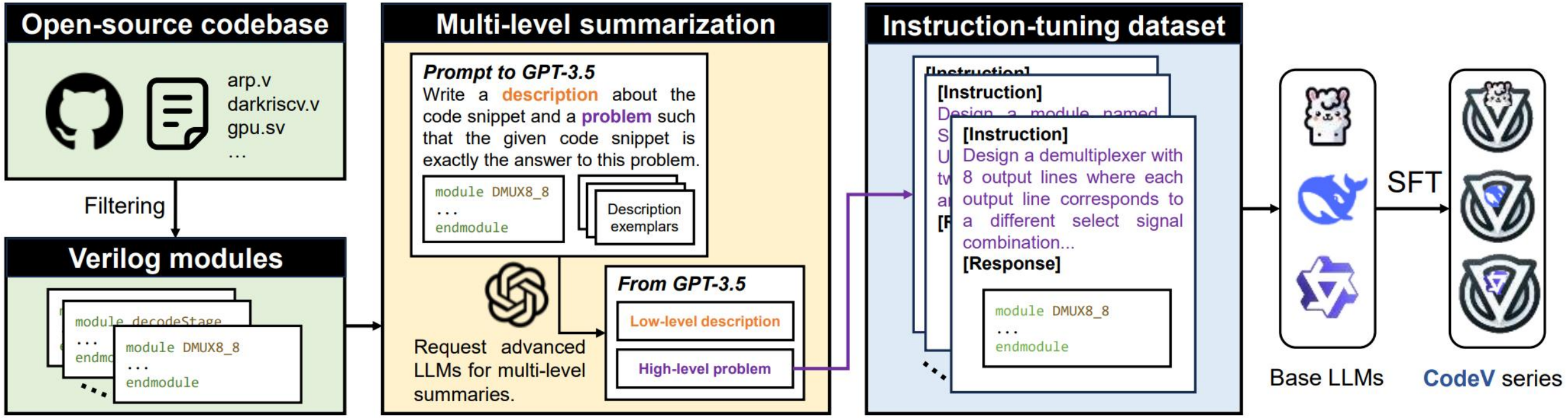
Design a system that combines the barrel shifter and the 2-to-1 multiplexer modules. The system should have four inputs (A, B, Shift, and Direction) and one output (Y). A and B are the two inputs to the multiplexer, while Shift and Direction are the inputs to the barrel shifter module. The system should select between the A and B inputs based on the value of SEL, which is the output of the barrel shifter module. The barrel shifter module should shift the selected input by the specified number of bit positions and in the selected direction.




```
module barrel_shifter (
    input [3:0] A,
    input [1:0] Shift,
    input Direction,
    output [3:0] Y
);
reg [3:0] shifted_value;
always @(*) begin
    case ({Direction, Shift})
        2'b00: shifted_value = A;
        2'b01: shifted_value = A << Shift;
        2'b10: shifted_value = A >> Shift;
        2'b11: shifted_value = {A[0], A[3:1]};
    endcase
end
assign Y = shifted_value;
endmodule

module mux_2to1 (
    input [3:0] A,
    input [3:0] B,
    input SEL,
    output [3:0] Y
);
assign Y = SEL ? B : A;
endmodule
```

Observations and Methods



- There is an **asymmetry** in the conversion between formal languages (code) and informal languages (natural language), where summarization is easier than generation.
- Multi-level summarization helps bridge the semantic gap between Verilog and natural language.

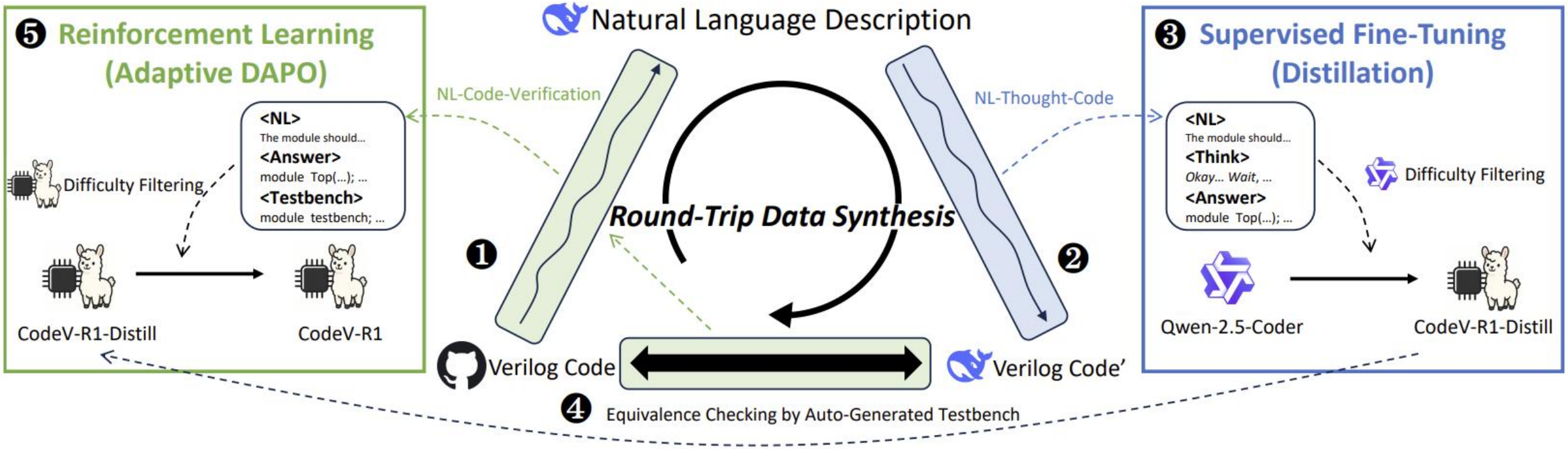
CodeV

TABLE II
COMPARISON OF OUR CODEV SERIES AGAINST VARIOUS BASELINE MODELS. RESULTS ARE CITED FROM THE ORIGINAL PAPER.

Type	Model	Model size	Open source	VerilogEval-Machine (%)			VerilogEval-Human (%)			RTLLM v1.1 (%)	
				pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	Syntax	Func.
Base LLMs	GPT-3.5	-	×	60.9	75.0	79.9	33.5	45.9	50.0	79.3	51.7
	GPT-4	-	×	60.0	70.6	73.5	43.5	55.8	58.9	**100.0**	65.5
	StarCoder [35]	15B	✓	46.8	54.5	59.6	18.1	26.1	30.4	93.1	27.6
	CodeLlama [36]	7B	✓	43.1	47.1	47.7	18.2	22.7	24.3	86.2	31.0
	DeepSeek-Coder [37]	6.7B	✓	52.2	55.4	56.8	30.2	33.9	34.9	93.1	44.8
	CodeQwen [38]	7B	✓	46.5	54.9	56.4	22.5	26.1	28.0	86.2	41.4
	Qwen2.5-Coder [39]	7B	✓	66.2	79.2	83.9	34.6	45.6	51.0	89.6	41.4
Fine-Tuned LLMs	ChipNeMo [40]	7B	×	43.4	-	-	22.4	-	-	-	-
	RTLCoder-Mistral [41]	7B	✓	62.5	72.2	76.6	36.7	45.5	49.2	96.6	48.3
	RTLCoder-DS [41]	6.7B	✓	61.2	76.5	81.8	41.6	50.1	53.4	93.1	48.3
	BetterV-CL [42]	7B	×	64.2	75.4	79.1	40.9	50.0	53.3	-	-
	BetterV-DS [42]	6.7B	×	67.8	79.1	84.0	45.9	53.3	57.6	-	-
	BetterV-CQ [42]	7B	×	68.1	79.4	84.5	46.1	53.7	58.2	-	-
	CraftRTL-CL [43]	7B	×	78.1	85.5	87.8	63.1	67.8	69.7	93.9	52.9
	CraftRTL-DS [43]	6.7B	×	77.8	85.5	88.1	65.4	70.0	72.1	84.3	58.8
CodeV-Verilog	CodeV-Verilog-CL	7B	✓	78.1	86.0	88.5	45.2	59.5	63.8	93.1	62.1
	CodeV-Verilog-DS	6.7B	✓	77.9	88.6	90.7	52.7	62.5	67.3	89.7	55.2
	CodeV-Verilog-CQ	7B	✓	77.6	88.2	90.7	53.2	65.1	68.5	93.1	55.2
	CodeV-Verilog-QC	7B	✓	80.1	87.9	90.5	59.2	65.8	69.1	96.6	51.7
CodeV-All	CodeV-All-CL	7B	✓	78.5	85.6	87.6	46.6	58.8	62.5	96.6	55.2
	CodeV-All-DS	6.7B	✓	79.8	86.0	86.7	53.0	63.3	67.2	96.6	51.7
	CodeV-All-CQ	7B	✓	79.9	88.3	91.1	54.1	65.1	68.6	93.1	58.6
	CodeV-All-QC	7B	✓	**81.9**	**89.9**	**92.0**	56.6	67.9	71.4	96.6	55.2

Achieves a leading position in VerilogEval and RTLLM benchmarks

CodeV-R1


❺ Reinforcement Learning (Adaptive DAPO)
<NL>
The module should...
<Answer>
module Top(...); ...
<Testbench>
module testbench; ...
Difficulty Filtering
CodeV-R1-Distill
CodeV-R1
Natural Language Description
NL-Code-Verification
NL-Thought-Code
Round-Trip Data Synthesis
❶
❷
Verilog Code
Verilog Code'
❹ Equivalence Checking by Auto-Generated Testbench
❸ Supervised Fine-Tuning (Distillation)
<NL>
The module should...
<Think>
Okay... Wait, ...
<Answer>
module Top(...); ...
Difficulty Filtering
Qwen-2.5-Coder
CodeV-R1-Distill

CodeV-R1

_different[0], neighbor is in[1]. But the example says that out_different[2] is in[2] XOR in[3]. So the code for out_different would be:

out_different[3] = 0 (no neighbor to the left)

out_different[2] = in[2] ^ in[3]

out_different[1] = in[1] ^ in[2]

out_different[0] = in[0] ^ in[1]

Wait, but the example says that in[3]'s neighbor to the left is in[0]. Wait, the example says: \"For this part, treat the vector as wrapping around, so in[3]'s neighbor to the left is in[0].\" Wait, that's conflicting. The example says that in[3]'s neighbor to the left is in[0], but according to the previous understanding, the neighbor to the left of in[3] is in[2], which is lower index. So this is a problem. Let me re-read the problem statement.

Re-read the problem

Problem statement for out_different:

(3) out_different: Each bit of this output vector should indicate whether the corresponding input bit is different from its neighbor to the left. For example, out_different[2] should indicate if in[2] is different from in[3]. For this part, treat the vector as wrapping around, so in[3]'s neighbor to the left is in[0].

Reflect

Ah, so the neighbor to the left is the higher index. So for in[3], the neighbor to the left is in[2], not in[0]. But the example says that out_different[2] is in[2] different from in[3], which aligns with that. So perhaps the wrapping around is not the case here. The example says that in[3]'s neighbor to the left is in[0]. Wait, that's conflicting. Wait, the problem says: \"For this part, treat the vector as wrapping around, so in[3]'s neighbor to the left is in[0].\" So when the vector is wrapped around, the left neighbor of in[3] is in[0]. So the neighbor to the left is the higher index, but if there is none, then wrap around to the right. So for each bit i, the neighbor to the left is (i-1) mod 4. Wait, no. Let me think again.

Reflect

In the case of wrapping around, the neighbor to the left of in[3] is in[0], which is the rightmost bit. So the left neighbor is considered the higher index. So for in[3], there is no higher index, so the left neighbor wraps around to the right. So the left neighbor of in[3] is in[0]. So the code for out_different[i] is in[i] XOR in[(i-1) mod 4]. Wait, but how to handle that. Let's see:

For i=3: neighbor to the left is in[0] (wrapping around). So out_different[3] = in[3] ^ in[0]

i=2: neighbor to the left is in[3], so out_different[2] = in[2] ^ in[3]

i=1: neighbor to the left is in[2], so out_different[1] = in[1] ^ in[2]

i=0: neighbor to the left is in[1], so out_different[0] = in[0] ^ in[1]

Wait, but the example says that out_different[2] is in[2] different from in[3], which is correct. So the code for out_different is:

out_different[3] = in[3] ^ in[0]

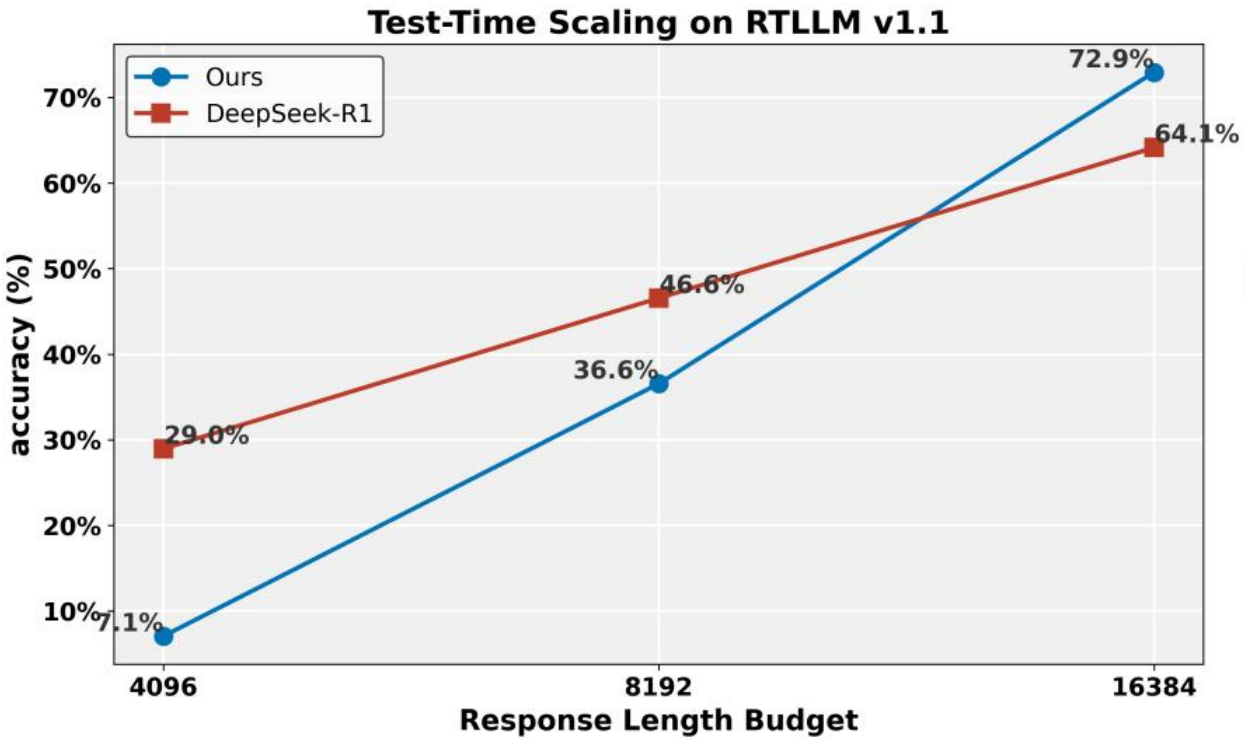
out_different[2] = in[2] ^ in[3]

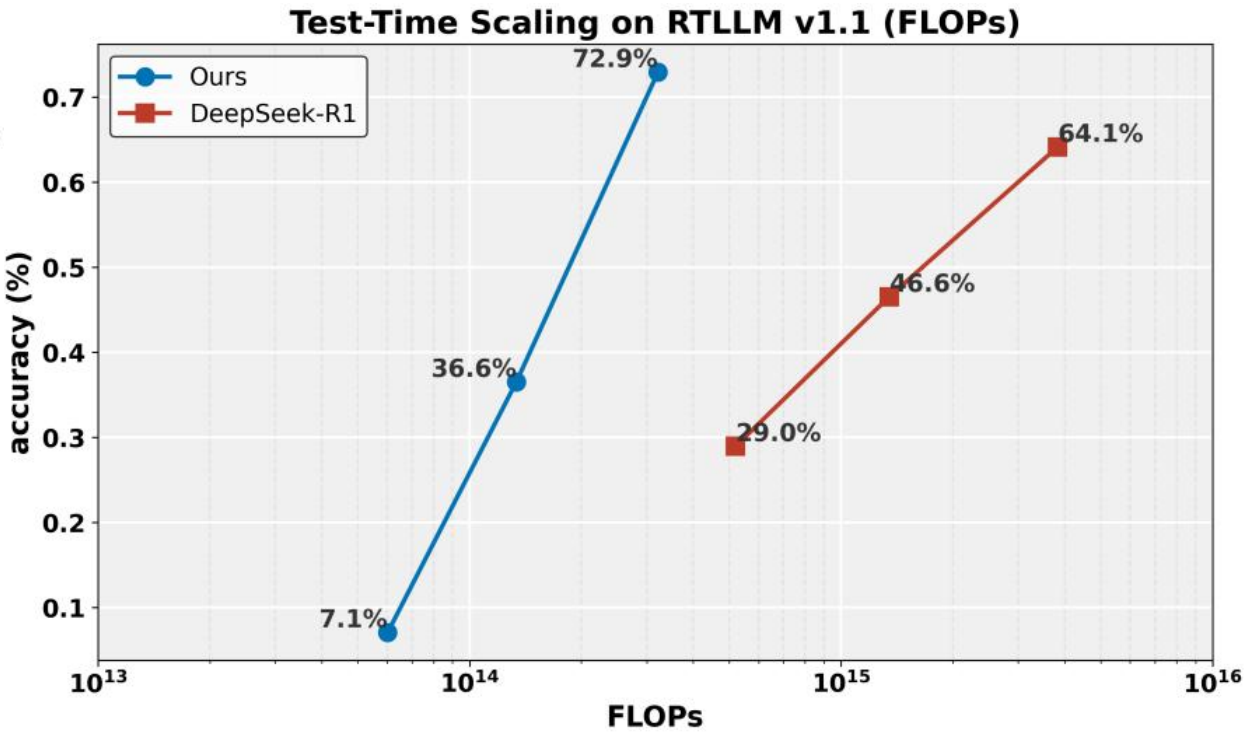
out_different[1] = in[1] ^ in[2]

out_different[0] = in[0] ^ in[1]

CodeV-R1 Reasoning

Tiny Model Size, Small Computation, Ultimate Performance





https://huggingface.co/zhuyaoyu/CodeV-R1-Qwen-7B
https://huggingface.co/collections/zhuyaoyu/codev-series-683f16115c357416e17b1bb5

QiMeng: Fully Automated Hardware and Software Design for Processor Chip

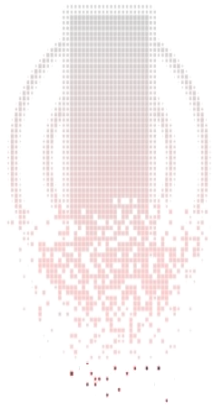
CPU

CodeV

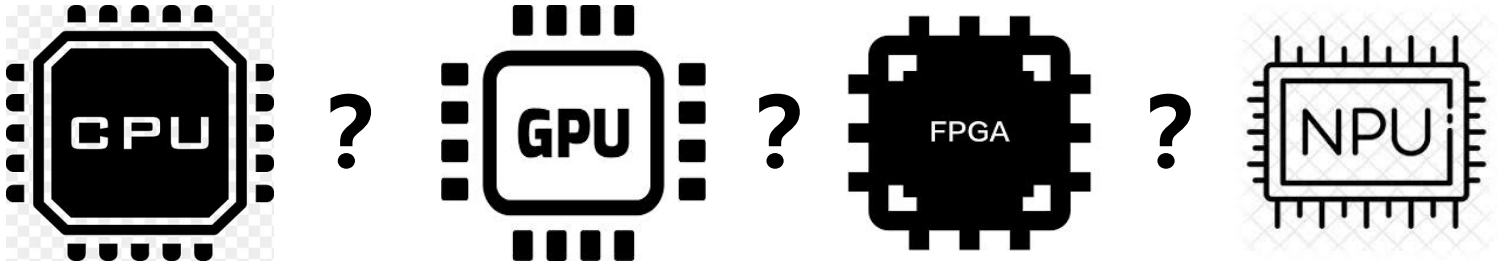
NL → HDL

Xpiler

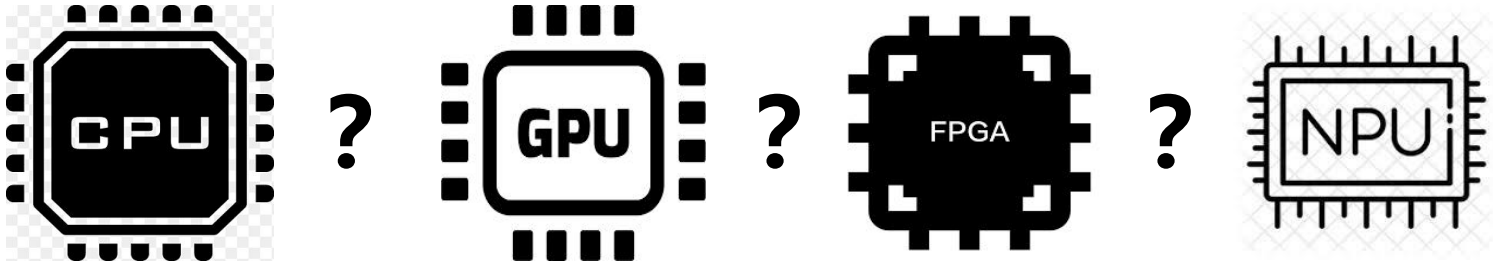
Cuda → BangC, HIP, DL Boost

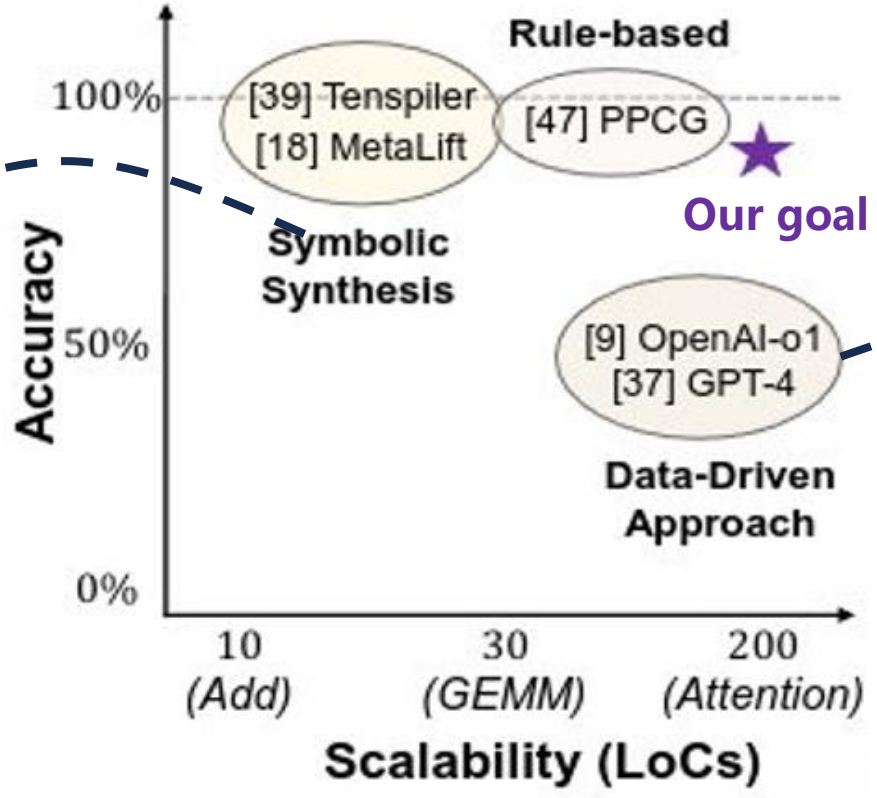


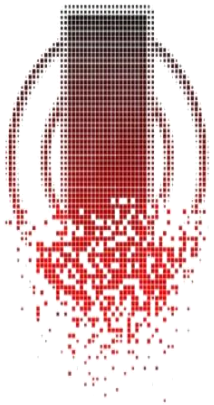
The Dilemma of Transcompilation



The Dilemma of Transcompilation







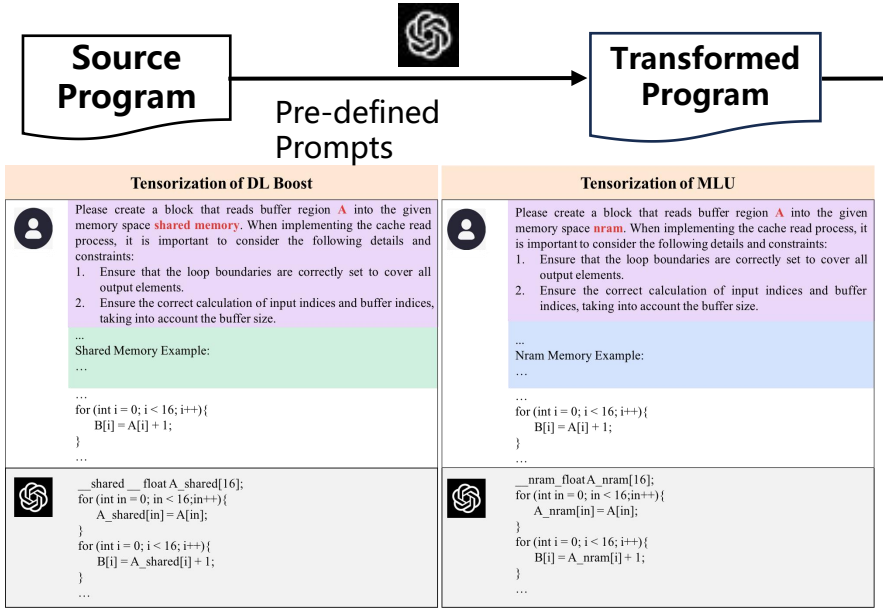
High accuracy, low scalability

High scalability, low accuracy

QiMeng-Xpiler

LLM-Based Program Transformation

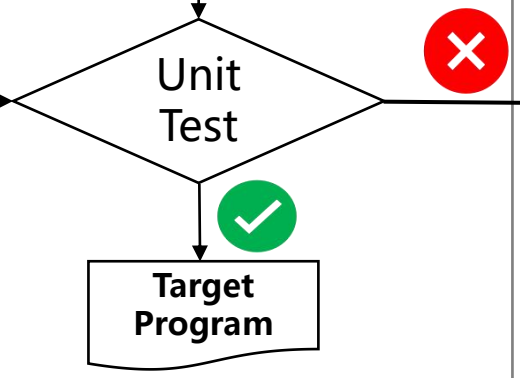
Leveraging LLMs' code comprehension and generation capabilities, we design general-purpose compilation prompts to enable automatic translation and enhance flexibility.



Code Guessing

Bug Localization

Identify faulty code segments using I/O pairs and execution traces through unit testing.

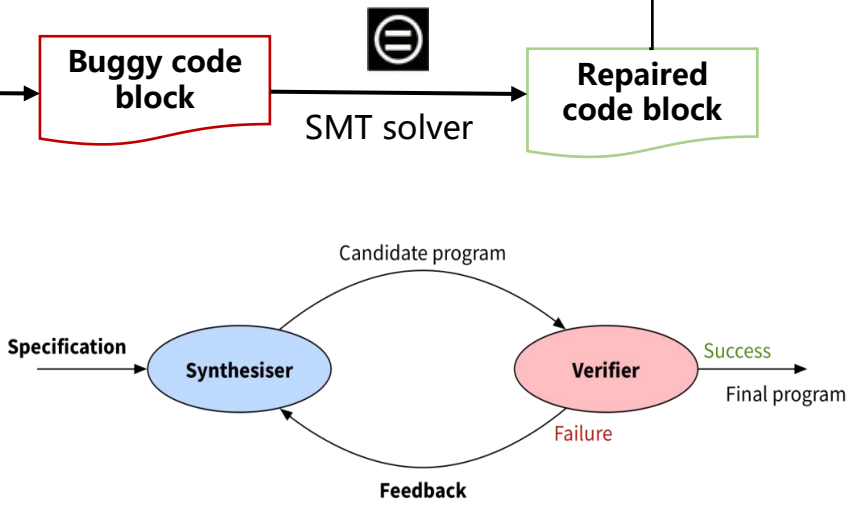


Auto Verification

SMT-based code repairing

Repair the incorrect low-level details through SMT-based symbolic synthesis with limited scale.

Index-related errors: Z3
tensor instruction-related errors: Tenspiler[1]



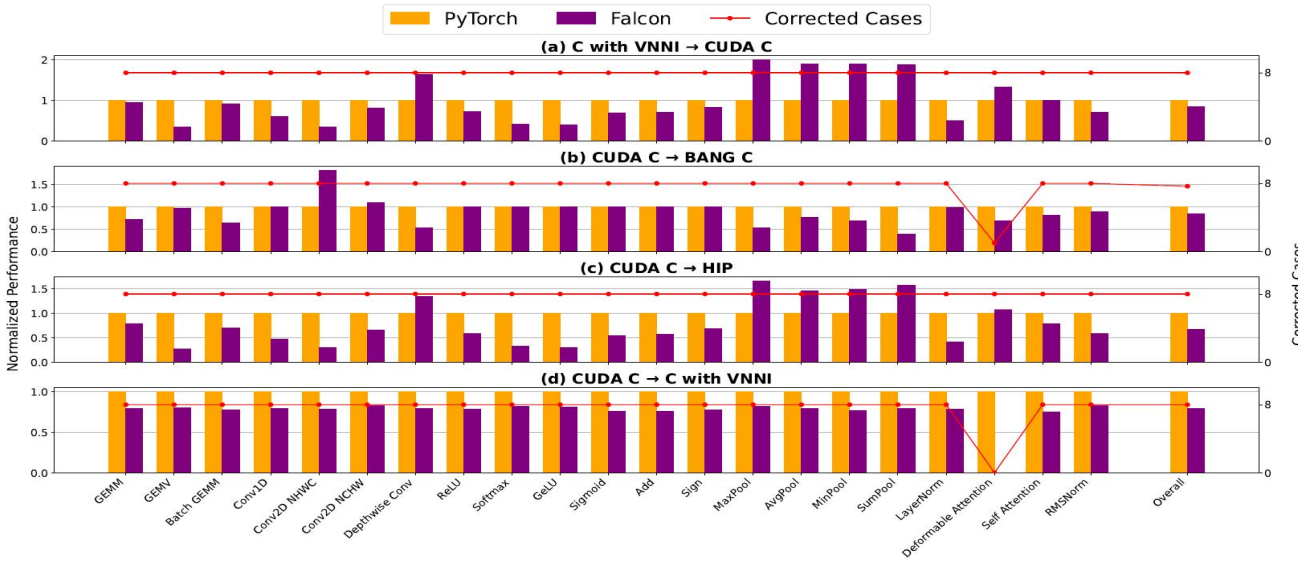
Auto Repair

[1] Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations

Transcompiler across Different Systems

Source	Method	Compilation Accuracy				Computation Accuracy			
		CUDA C	BANG C	Hip	C with VNNI	CUDA C	BANG C	Hip	C with VNNI
CUDA C	GPT-4 Zero-Shot	–	0	82.7	9.5	–	0	82.7	4.2
	OpenAI o1 Zero-Shot	–	0	85.7	61.9	–	0	82.7	60.7
	GPT-4 Few-Shot	–	50.6	97.0	84.5	–	7.7	96.4	30.4
	OpenAI o1 Few-Shot	–	51.8	98.2	85.1	–	48.2	98.2	55.4
	Falcon w/o SMT	–	82.7	98.2	88.1	–	54.2	98.2	58.3
	Falcon w/o SMT + Self-Debugging	–	87.5	98.8	89.3	–	84.5	98.2	58.9
	Falcon	–	**100**	**100**	**100**	–	**91.7**	**100**	**95.2**
BANG C	GPT-4 Zero-Shot	74.4	–	76.8	0	0	–	0	0
	OpenAI o1 Zero-Shot	27.4	–	97.0	9.5	0	–	0	4.2
	GPT-4 Few-Shot	69.0	–	66.1	23.8	6.5	–	6.5	13.1
	OpenAI o1 Few-Shot	71.4	–	97.0	41.7	10.1	–	7.7	23.2
	Falcon w/o SMT	85.1	–	84.5	47.6	77.4	–	78.6	41.1
	Falcon w/o SMT + Self-Debugging	88.1	–	88.7	50.6	77.4	–	78.6	41.1
	Falcon	**100**	–	**100**	**100**	**95.8**	–	**97.0**	**95.2**
Hip	GPT-4 Zero-Shot	97.0	0	–	23.8	97.0	0	–	5.4
	OpenAI o1 Zero-Shot	98.2	0	–	45.8	98.2	0	–	4.2
	GPT-4 Few-Shot	97.0	35.1	–	85.1	97.0	5.4	–	24.4
	OpenAI o1 Few-Shot	98.8	42.3	–	88.7	98.2	9.0	–	30.4
	Falcon w/o SMT	98.2	60.7	–	65.5	97.6	52.4	–	57.1
	Falcon w/o SMT + Self-Debugging	98.8	62.5	–	66.1	98.2	52.4	–	57.1
	Falcon	**100**	**100**	–	**100**	**100**	**86.9**	–	**96.4**
C with VNNI	GPT-4 Zero-Shot	57.1	0	60.1	–	8.3	0	8.9	–
	OpenAI o1 Zero-Shot	66.1	0	97.0	–	10.1	0	96.4	–
	GPT-4 Few-Shot	81.5	41.7	14.4	–	14.3	6.0	12.5	–
	OpenAI o1 Few-Shot	87.5	55.4	97.0	–	51.2	10.7	96.4	–
	Falcon w/o SMT	95.8	78.0	87.5	–	83.9	58.3	85.7	–
	Falcon w/o SMT + Self-Debugging	97.0	84.5	89.3	–	83.9	58.3	85.7	–
	Falcon	**100**	**99.4**	**100**	–	**98.2**	**88.7**	**99.4**	–

Experimental Results on Different Processors



Performance Comparison with Different PyTorch Backend Libraries

Deformable Attention (~ 200LoCs)		CUDA C-> BANG C		C with VNNI -> CUDA C	
		Costs	Performance	Costs	Performance
Senior Coder	Manual	~6 d	100%	~1 d	100%
	w/ Falcon	4.5 + 0.5 h	69.20%	2.1 h	132.50 %
	Time Saving	~28.8×		~11.4×	
Junior Coder	Manual	~30 d	49.85%	~3 d	75.76%
	w/ Falcon	4.5 + 3 h	65.17%	2.1 h	132.50 %
	Time Saving	**~96.0×**		**~34.3×**	

Productivity Improvement

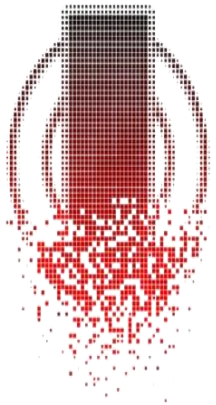
Enable rapid migration between software ecosystems:

- Automatic program translation across different processors (e.g., NVIDIA GPU, Cambricon MLU, AMD MI, Intel DLBoost) and programming models (e.g., SIMT, SIMD).
- Real-world C/CUDA-to-BANG program translation with a functional correctness rate exceeding 91.7%.
- Performance can reach up to 2× that of manually optimized vendor-provided code.
- Programming efficiency can be improved by up to 96× compared to manual development.

Conclusion: From Top-down and Bottom-up to Self-Evolution

Library	Automated High-Performance Library Generation
Compiler	Automated Tensor Program Transcompiler
	Automated Compiler Tool-Chain Design
OS	Automated OS Configuration Optimization
Processor	Automated HDL Generation
	Automated Front-End Design

Prior

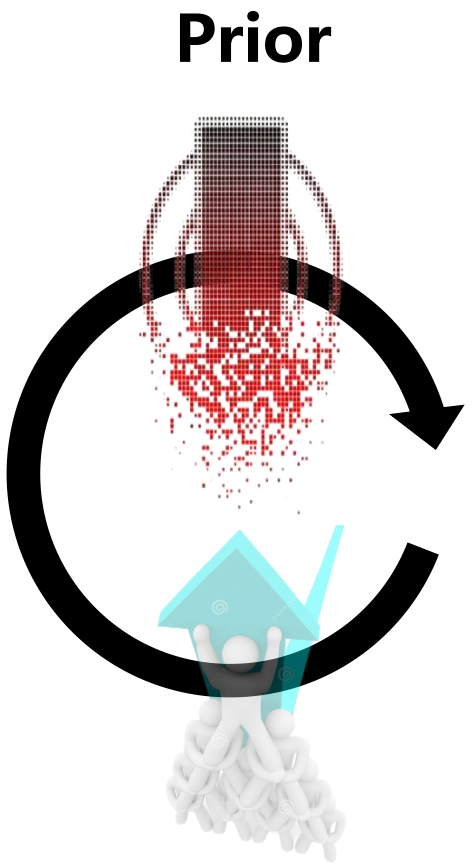


Feedback

https://qimeng-ict.github.io

Conclusion: From Top-down and Bottom-up to Self-Evolution

Library	Automated High-Performance Library Generation
Compiler	Automated Tensor Program Transcompiler
	Automated Compiler Tool-Chain Design
OS	Automated OS Configuration Optimization
Processor	Automated HDL Generation
	Automated Front-End Design



https://qimeng-ict.github.io