Guaranteed Guess: A Language Modeling Approach for CISC-to-RISC Transpilation with Testing Guarantees

Anonymous ACL submission

Abstract

The hardware ecosystem is rapidly evolving, with increasing interest in translating low-level programs across different *instruction set architectures* (ISAs) in a quick, flexible, and correct way to enhance the portability and longevity of existing code. A particularly challenging class of this transpilation [1] problem is translating between complex- (CISC) and reduced- (RISC) hardware architectures, due to fundamental differences in instruction complexity, memory models, and execution paradigms. In this work, we introduce GG (Guaranteed Guess), an ISA-centric transpilation pipeline that combines the translation power of pre-trained large language models (LLMs) with the rigor of established software testing constructs. Our method generates candidate translations using an LLM from one ISA to another, and embeds such translations within a software-testing framework to build quantifiable confidence in the translation. We evaluate our GG approach over two diverse datasets, enforce high code coverage (>98%) across unit tests, and achieve functional/semantic correctness of 99% on HumanEval programs and 49% on BringupBench programs, respectively. Further, we compare our approach to the state-of-the-art Rosetta 2 framework on Apple Silicon, showcasing 1.73× faster runtime performance, 1.47× better energy efficiency, and 2.41× better memory usage for our transpiled code, demonstrating the effectiveness of GG for real-world CISC-to-RISC translation tasks. We will open-source our codes, data, models, and benchmarks to establish a common foundation for ISA-level code translation research.

1 Introduction

The modern hardware landscape is undergoing a fundamental transformation. As Moore's Law slows and Dennard scaling ends (Dennard et al., 1974; Connatser, 2023), the demand for energy-efficient, high-performance architectures has accelerated, particularly with the rise of machine learning (ML) applications (Horowitz, 2014; Jouppi et al., 2017). Hyperscalers are increasingly constrained by power and thermal limits (Patterson et al., 2021; Gupta et al., 2021), prompting a reevaluation of datacenter infrastructure.

A major outcome of this shift is the growing adoption of ARM-based processors. Historically dominant in mobile and edge devices due to their RISC-based, low-power design, ARM CPUs were largely absent from datacenters because of their performance gap with x86 (a CISC architecture) (Blem et al., 2013). However, this gap has narrowed significantly: ARM-based chips now match x86 on many benchmarks (CloudPanel, 2023) and deliver superior energy efficiency (IONOS, 2024). In 2024, x86 designs dominated over 80% of data center servers (Reuters, 2025), but ARM predicts that its share will reach 50% by the end of 2025 (Maruccia, 2025). Industry adoption supports this trend, with ARM-based systems like NVIDIA's Grace CPU (NVIDIA Corporation, 2024), Amazon's Graviton (Morgan, 2022), and Microsoft's ARM-compatible OS stack (Verma, 2024) accelerating deployment.

This rapid hardware transition introduces a significant software gap. Legacy binaries compiled for x86 often lack source code and cannot be recompiled for ARM. While solutions like Apple's Rosetta 2 (Apple Inc., 2020) and QEMU's emulation service (Bellard, 2005) provide runtime virtualization, they introduce memory and performance overheads. Compilers struggle to retarget opaque binaries (He et al., 2018), and decompilation-based approaches are fragile or legally restricted (Wang et al., 2024). A scalable, accurate, and architecture-aware binary-to-binary translation solution remains elusive.

In this work, we introduce *Guaranteed Guess* (GG), an assembly-to-assembly transpiler that trans-

[1] *We use "transpilation"* to describe the task of translating code between assembly languages.

lates x86 binaries (CISC) into efficient ARM or RISC-V (RISC) equivalents using a custom-trained large language model (LLM). Our approach is *open-source*, avoids the *virtualization tax* by generating native ARM/RISC-V assembly, and directly supports legacy binaries *without decompilation*.

Transpiling across ISAs is non-trivial. CISC and RISC architectures differ in register-memory semantics, instruction complexity, and binary length, x86 instructions are fewer but more expressive, while RISC requires longer, register-centric code sequences. These differences must be learned implicitly by the model, which we achieve by incorporating hardware-informed design, tokenizer extensions, and context-aware training.

Our approach builds high-accuracy LLM-based transpilers by incorporating hardware-aware insights into the training process, enabling the model to better capture the CISC-specific patterns of x86 and generate semantically valid RISC targets such as ARM. However, unlike high-level language tasks, conventional NLP correctness proxies (e.g., BLEU, perplexity) fall short for binary translation where functional correctness is paramount. Therefore, we embed our predictions within rigorous software testing infrastructure to provide test-driven guarantees of correctness. Holistically, our paper makes the following key contributions:

1. The first CISC-to-RISC transpiler, coined **GG**, built via a custom-trained, architecture-aware LM achieving a test accuracy of 99.39% on ARMv8 and 89.93% on RISC-V64.

2. A methodology to measure and build confidence into transpilation output via software testing approaches ("guaranteeing" the guess) (§3), including detailed analysis of correctness, errors, and hallucinations (§4)

3. An in-depth analysis into the inner workings of our transpiler, including hardware-informed design decisions to best train an accurate LLM model for assembly transpilation (§3, §5).

4. We perform a case-study using our transpiler in a real-world setting, by comparing it directly to Apple Rosetta's x86 to ARM virtualization engine. Results show that **GG**'s generated assembly achieves 1.73x runtime speedup while delivering 1.47x better energy efficiency and 2.41x memory efficiency (§5).

2 Background and Related Work

Virtualization and Emulation Emulation and assembly-level virtualization enable the execution of one ISA's binary on a host machine for which it was not originally compiled. QEMU (Bellard, 2005), an open-source emulator, uses dynamic binary translation (Sites et al., 1993) to translate machine code on-the-fly, offering flexibility but with performance overhead. Supported emulation currently includes x86 to ARM, amongst other ISAs. Rosetta 2 (Apple Inc., 2020), Apple's virtualization layer for macOS, combines ahead-of-time (AOT) and just-in-time (JIT) translation, providing better performance within the Apple ecosystem.

These approaches face challenges in achieving native-level performance and ensuring broad compatibility, due to the dynamic nature of execution. A transpiler approach, directly converting x86 to ARM assembly, could supplant these solutions by eliminating runtime translation overhead with a one-time translation into the host ISA. This method could address the limitations of current emulation and virtualization techniques, particularly in performance-critical scenarios, or where pre-processing is feasible, or when source code is not available (due to proprietary IP).

Coding with LLMs Language modeling approaches for code have primarily focused on understanding, generating, and translating high-level programming languages such as C++, Java, and Python (Lachaux et al., 2020; Feng et al., 2020; Wang et al., 2021; Roziere et al., 2023; Liu et al., 2024). These models demonstrate increasingly sophisticated code manipulation capabilities through self-supervised learning on vast code repositories. Models further trained with reinforcement learning have shown remarkable performance in rules-based reasoning tasks, including code (et al., 2025). However, the resulting models struggle when applied to languages under-represented in their training sets, in particular when used to write assembly-level code, where the semantics and structure differ significantly from their high-level counterparts.

Neural Low-Level Programming Recent research demonstrates the potential of adapting LLMs to various tasks related to low-level code analysis and transformation: decompilation, binary similarity analysis, and compiler optimization. LLM4Decompile (Tan et al., 2024) introduced specialized language models for direct binary-to-source

translation and decompiler output refinement. DeGPT (Hu et al., 2024) further explored decompiler enhancement through semantic-preserving transformations. SLaDe (Armengol-Estapé et al., 2024) combines a 200M-parameter sequence-to-sequence Transformer with type inference techniques to create a hybrid decompiler capable of translating both x86 and ARM assembly code into readable and accurate C code, effectively handling various optimization levels (-O0 and -O3). Language models have also been adapted to optimization tasks, with LLM Compiler (Cummins et al., 2024) introducing a foundation model that supports zero-shot optimization flag prediction, bidirectional assembly-IR translation, and compiler behavior emulation. Binary similarity analysis has similarly benefited from language model adaptations. DiEmph (Xu et al., 2023) addressed compiler-induced biases in transformer models, while jTrans (Wang et al., 2022) incorporated control flow information into the transformer architecture. Yu et al. (Yu et al., 2020) combined BERT-based semantic analysis with graph neural networks to capture both semantic and structural properties of binary code. While these applications have shown promising results, the use of LLMs to port efficient machine code from one machine to another, while maintaining efficiency, remains underexplored and largely unsolved. Assembly languages present unique challenges due to their under-representation in training datasets, lack of human readability, extensive length, and fundamental differences in execution models across architectures.

Guess & Sketch (Lee et al., 2024) introduced a neurosymbolic approach combining language models with symbolic reasoning for translating assembly code between ARMv8 and RISC-V architectures. In our work, we extend the neural transpiliation direction with a focus on leveraging the existing efficiency in x86 programs to transpile into efficient ARM binaries, bridging architectural differences in ISA complexity and execution models. Further, instead of fixing transpilations with symbolic approaches, as done in Guess & Sketch, we focus on upfront data design and modeling methods to flexibly handle the increased scale and complexity of CISC-to-RISC transpilation.

3 Guaranteed Guess

In this section, we explore the two primary components of building our GG transpiler: data generation and model training.

3.1 Data Collection

As shown in Figure 1, our training dataset is derived from AnghaBench(Da Silva et al., 2021) and The Stackv2(Kocetkov et al., 2022). AnghaBench is a comprehensive benchmark suite that contains 1 million compilable C/C++ programs extracted from major public C/C++ repositories on GitHub. The Stack is a 3.1TB dataset of permissively licensed code in 30 languages for training and evaluating code LLMs. From these datasets, we randomly sampled 1.01M programs (16.16B tokens) from AnghaBench and 306k programs (4.85B tokens) from the stack to form our training set, equivalent to 1.32M samples. After we collected the whole samples, we removed boilerplates, deduplicated the data, and choose file that were neither too short (<10 lines) nor too long (>16k lines). These programs were then compiled for x86 (CISC) $\leftrightarrow$ ARMv8/ARMv5/RISC-V (RISC).

Each program was compiled to both x86 (CISC) $\leftrightarrow$ ARMv8/ARMv5/RISC-V (RISC) targets under two optimization levels: `-O0` (no optimization) and `-O2` (aggressive optimization). These flags were selected to expose models to both raw, semantically transparent code (`-O0`) and real-world, performance-optimized binaries (`-O2`), enabling the model to learn both unoptimized and optimized ISA patterns. Compilation for ARMv5 and RISC-V64 was performed via cross-compilation on an Ubuntu 20.04 machine with a Ryzen 7 CPU, using `arm-linux-gnueabi-gcc` (Radcolor, n.d.) and `gcc-riscv64-linux-gnu` (Project, 2025), respectively. ARMv8 binaries were compiled natively on an Apple M2 Pro (macOS) using `clang` (Lattner, 2008), ensuring architectural fidelity for performance-critical ARM targets.

3.2 Training

All hyperparameter optimization experiments were conducted on a small 500k portion of AnghaBench. We tested various hyperparameter settings on this subset of our benchmark. After identifying the optimal configuration, we scaled up the training data to 1.31M samples. We trained three models: `DeepSeek-Coder1.3B` (Guo et al., 2024), `Qwen2.5-Coder` (1.5B and 0.5B) (Hui et al., 2024b). Given the dataset size of 1.3M million samples, with an average of 13k tokens per sample, we opted for smaller models. Training was done on A100 GPUs (40GB each). Training with 1.3M

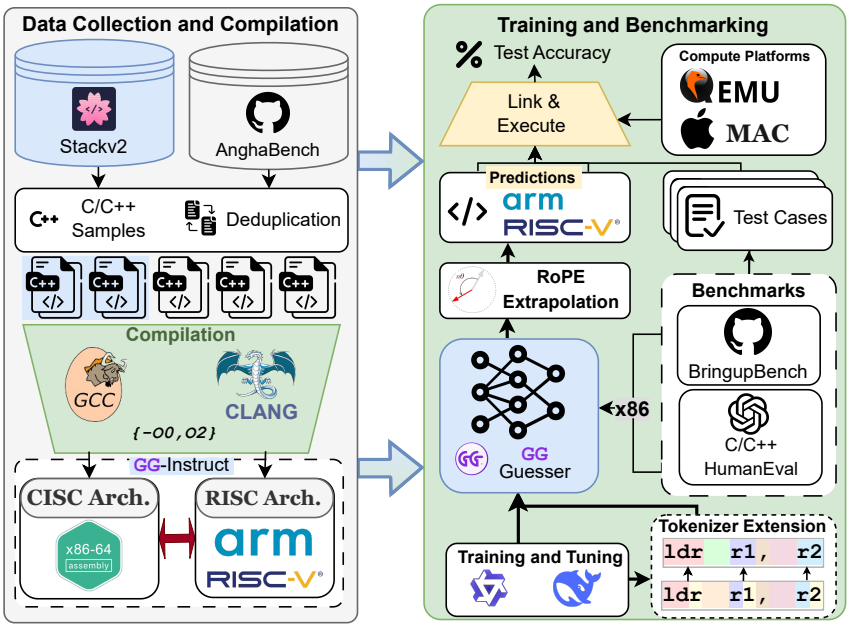


Figure 1: GG **System Overview**. A two-stage transpilation pipeline from x86 to ARM/RISC-V. Left: Data is sourced from Stackv2 and AnghaBench, deduplicated, and compiled using both GCC and Clang to generate paired assembly (x86 ↔ ARM) from C/C++. Right: A specialized LLM (GG Guesser), trained with tokenizer extension and inferenced with RoPE extrapolation, predicts target ISA code. Predictions are evaluated via unit tests and symbolic analysis on benchmarks like HumanEval and BringupBench. The system emphasizes functional correctness, architectural alignment, and near-native performance.

samples, a batch size of 24, and 2 epochs required three days. To conserve memory, mixed precision training with bfloat16 was employed. Given limited capacity for large batch sizes, we applied gradient accumulation with an effective batch size of 2. We used paged AdamW (Loshchilov, 2017) to avoid memory spikes, with a weight decay of 0.001. We chose a small learning rate of $2 \times 10^{-5}$with a cosine schedule, as experiments indicated this schedule performed best. We trained our model with a context window of 16k. In inference, we do RoPE (Su et al., 2024) extrapolation to increase the context window to 32.7k.

Input	`ldr r1, r2`
Tokenizer	**Tokens**
DeepSeek/Qwen 2.5 coder	`ld` `r` `␣` `r` `1` `,` `␣` `r` `2`
GGExtended Tokenizer	`ldr` `␣` `r1` `,` `␣` `r2`

Table 1: Comparison of tokenization approaches between DeepSeek/Qwen-Coder and our extended tokenizer. Spaces are represented as ␣ and shown with colored backgrounds to highlight token boundaries. Note how our tokenizer groups related tokens (e.g., `ldr` and `r1`) as singular units.

3.3 Tokenizer Extension

To improve our LLMs' capability in comprehending and generating assembly code, we augmented the tokenizer by incorporating the most common opcodes and register names from x86 and ARMv5/ARMv8/RISC-V64 architectures (as shown in Table 1). This targeted design improves token alignment with instruction semantics, enabling more precise and efficient assembly translation. As shown in table 2, our extension decreases the fertility rate (tokens/words) (Rust et al., 2020) of Qwen and Deepseek tokenizers by 2.65% and 6.9%, respectively. This corresponds to our model fitting 848 and 2.2k tokens respectively.

Model	**x86**	**ARMv5**	**ARMv8**	**RISC-V64**
Qwen-Coder (Hui et al., 2024a)	4.28	2.89	3.62	3.62
DeepSeek-Coder (Guo et al., 2024)	3.74	3.51	4.28	4.28
GG-Qwen (Ours)	**4.14**	**2.87**	**3.50**	**3.50**
GG-DeepSeek (Ours)	**3.47**	**3.26**	**3.99**	**3.37**
Δ Qwen (%)	↓3.3%	↓0.5%	↓3.4%	↓3.4%
Δ DeepSeek (%)	↓7.2%	↓6.9%	↓6.8%	↓6.8%

Table 2: Tokenizer fertility rate (tokens/words) across ISAs. Lower is better.

Model	ARMv5		ARMv8		ARMv8	
	HumanEval	HumanEval	HumanEval	HumanEval	BringupBench	BringupBench
	-O0	-O2	-O0	-O2	-O0	-O2
GPT-4o (OpenAI, 2024)	8.48%	3.64%	10.3%	4.24%	1.54%	0%
Qwen2.5-Coder-1.5B (Hui et al., 2024a)	0%	0%	0%	0%	0%	0%
Qwen2.5-Coder-3B (Hui et al., 2024a)	0.61%	0%	0%	0%	0%	0%
StarCoder2-3B (Lozhkov et al., 2024)	0%	0%	0%	0%	0%	0%
Deepseek-R1-1.5B (Guo et al., 2025)	0%	0%	0%	0%	0%	0%
Deepseek-R1-Qwen-7B (Guo et al., 2025)	0%	0%	0%	0%	0%	0%
GG-Deepseek-1.3B	79.25%	12.80%	75.15%	10.3%	3.08%	0%
GG-0.5B	90.85%	23.03%	86.06%	25.45%	27.69%	3.08%
GG-1.5B	**93.71%**	**50.30%**	**99.39%**	**45.12%**	**49.23%**	**15.38%**

Table 3: Models trained with our method outperform baselines across all benchmarks, at all optimization levels.

4 Experiments and Evaluation

In this section, we describe our experimental setup, training methodology, evaluation benchmarks, and the metrics used to assess the accuracy and robustness of our CISC-to-RISC transpiler.

4.1 Setup

We leveraged LLaMa-Factory (Zheng et al., 2024), DeepSpeed Zero3 (Rasley et al., 2020), liger kernels (Hsu et al., 2024), and FlashAttention2 (Dao, 2023) for efficient training and memory optimization. We also used caching to enhance inference speed and disabled sampling to ensure deterministic outputs. We used vLLM (Zheng et al., 2023) to deploy our model and achieve a throughput of 36x requests per second at 32.7k tokens context window on a single A100 40GB GPU. Additionally, We apply post-quantization using `llama.cpp` (Ggerganov) (e.g., `bfloat16`, `int8`, `int4`) to optimize inference for CPU-based deployment.

4.2 Evaluation

We evaluate GG using two complementary benchmarks: HumanEval-C (Tan et al., 2024) and BringUpBench (Austin, 2024). HumanEval was originally introduced by Chen et al. (2021) for Python code generation. The benchmark consists of 164 programming problems that assess language comprehension, reasoning, and algorithmic thinking. For our evaluation, we utilize the C-translated version from LLM4Decompile (Tan et al., 2024), which maintains the same problems while converting both function implementations and test cases to C code.

To evaluate real-world generalization, we leverage BringUpBench (Austin, 2024), a challenging benchmark of 65 bare-metal programs ranging from 85 to 5751 lines of code. Unlike HumanEval, which consists of isolated functions, BringUpBench programs are embedded in full project structures with

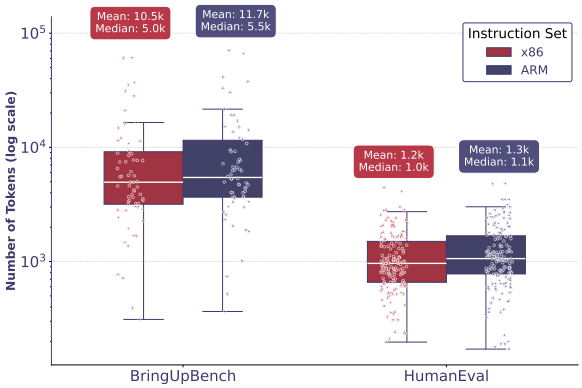


Figure 2: Token counts by ISA and benchmark; BringUpBench is substantially longer than HumanEval.

internal libraries and cross-linked components. This setup more accurately reflects real-world embedded systems development, where executing even a single file often requires compiling and linking the entire codebase. As a result, BringUpBench imposes significantly greater context length demands. On average, each BringUpBench sample requires 8.9× more tokens for x86 and 8.8× more for ARM compared to HumanEval, as shown in Figure 2. The benchmark's diverse control flow and I/O patterns further elevate its difficulty, making it a strong testbed for assessing the robustness and scalability of our transpiler.

We use `gcov`, GNU's coverage tool, to measure line coverage, a core metric in software testing that captures which code lines were executed at least once, thereby exposing untested paths and blind spots (Myers et al., 2011). HumanEval and BringupBench achieved 98.81% and 97.32% average coverage, respectively, indicating near-complete execution of all code lines during testing.

We evaluate functional correctness by executing the transpiled ARM code against full unit test suites. A prediction is deemed correct only if all test cases pass, partial correctness is not counted. For HumanEval, this involves compiling the

predicted code, linking it with the provided tests, and executing the binary as shown inf figure 1. For BringUpBench, we leverage its Makefile to build the static library and link it with the target file. The output is then compared against the expected output using a diff-based check. This strict pass@1 evaluation, based solely on the most probable sample, even when beam search (beam size = 8) is used, ensures that only fully functional translations contribute to final accuracy.

5 Results and Analysis

We evaluate the efficacy of our transpiler for CISC-to-RISC assembly translation, focusing on the correctness of the output ARM assembly. Utilizing the metrics defined above (§4), we compare our approach with state-of-the-art coding LLMs and evaluate our approach for x86 to ARM transpilation (Table3).

5.1 Transpiler Validation

Baselines. As shown in Table 3, most baseline models, including state-of-the-art LLMs such as StarCoder2 (Lozhkov et al., 2024), DeepSeek (Guo et al., 2024), and Qwen2.5 (Hui et al., 2024a), achieve 0% accuracy in all transpilation tasks, underscoring the unique difficulty of low-level ISA translation. These models, while effective on high-level programming benchmarks, lack the architectural grounding and token-level inductive bias needed to generalize from x86 to ARM. GPT-4o was the only exception, achieving 1.5-8% accuracy, which remains far below usable thresholds, highlighting that general-purpose LLMs are not yet suitable for assembly-level translation without specialized training. This performance gap reinforces the need for task-specific instruction tuning and architectural adaptation to handle the deep structural mismatch between CISC and RISC.

GG Results. Our GG models, particularly the GG-1.5B variant, substantially outperform all baselines, reaching 99.39% accuracy on ARMv8 and 93.71% on ARMv5 under the `-O0` setting. This validates the effectiveness of architecture aware training, tokenizer extension, and longer context modeling in capturing fine-grained register and memory semantics. For `-O2` optimized code, accuracy drops to 45.12% (ARMv8) and 50.30% (ARMv5), exposing the fragility of current LLMs under aggressive compiler transformations. This suggests that while our model learns to generalize well under minimal optimization, it struggles with control/data flow reordering and register coalescing introduced by `-O2` passes. Addressing this challenge may require incorporating optimization-invariant representations, such as symbolic traces or control/data-flow graphs, or extending the training set with more aggressively optimized samples.A detailed error analysis can be found in Appendix A.1.

Error Type	Files with Errors after Guess
Input + output out of context window	LongDiv, Regex-Parser, RLE-Compress, FFT-Int, Blake2B, Anagram, C-Interp, Totient, Banner, Lz Compress, Satomi, Rho-Factory
Duplicate function error	Frac-Calc, Minspan
Stack/memory error	Boyer-Moore-Search, Topo-Sort, Audio-Codec, Weekday, Simple-Grep, Max-Subseq, Priority-Queue, Dhrystone, Cipher, AVL-Tree, QSort-Demo, Vectors-3D, Pascal
Missing function error	Fuzzy-Match, Tiny-NN, Kadane, Audio-Codec, Frac-Calc, Kepler, Dhrystone, Cipher, Graph-Tests, Quaternions, AVL-Tree, K-Means, QSort-Demo, Vectors-3D
Labels referred but not defined	Fuzzy-Match, Life, AVL-Tree, K-Means
Register mislabel error	Bloom-Filter, Topo-Sort, Weekday, Knights-Tour, Simple-Grep, Max-Subseq, Mersenne, Audio-Codec, K-Means, QSort-Demo, Vectors-3D, Pascal, Minspan
Incorrect immediate value	Kadane

Table 4: Failed files on BringupBench. Errors after the Guess stage are largely around dataflow reasoning. File names are grouped by error type.

RISC-v64. To demonstrate the generality of our method, we also trained our model on the task of transpiling from x86 to RISC-V64, achieving a pass@1 of 89.63%. Notably, our model significantly outperforms existing models like GPT4o and DeepSeekCoder2-16B, which achieved much lower test accuracies of 7.55% and 6.29%, respectively. This result is 9% lower than ARMv8 which shows how much different RISC-v64 from x86 compared ARMv8.

(-O2) Opt. Compiler optimizations (-O2) introduce complex patterns that increase failure frequency compared to `-O0`. A common error is the motion of the instruction; for example, misplacing `cbz`[2] alters the control flow, revealing the difficulty of the model in interpreting optimized sequences. While hard to detect automatically, such errors can be repaired via manual inspection (Liu et al., 2025), symbolic solvers (Lee et al., 2024; Mora et al., 2024), or reasoning models. Hybrid human-AI approaches may improve correctness guarantees.

[2]Compare and Branch if Zero

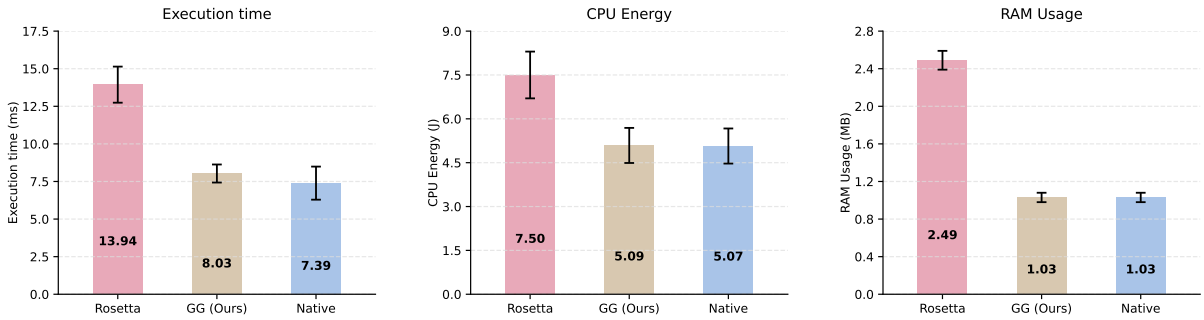


Figure 3: Comparison of execution time, energy consumption, and memory usage across Rosetta, GG, and native binaries.

BringUpBench. We evaluate GG-1.5B on BringUpBench (Austin, 2024) and manually analyze over 200 unit-tested binaries. Our model achieves 49.23% exact match accuracy under `-O0` (Table 3) with virtually no syntax errors, outputs consistently adhere to valid ARM assembly with correct opcodes, registers, and memory access. This reflects a strong surface-form prior, shifting focus to semantic errors like incorrect dataflow. Notably, 17% of failures stem from context truncation, indicating a key limitation of current context window sizes. Table 4 summarizes common failure types, including duplicate code, invalid control flow, misused registers / intermediaries, and stack errors - most symptomatic of broken data flow rather than syntax issues. These may be alleviated through longer training, symbolic repair, or richer representations. Lastly, the benchmark's extensive unit tests offer a valuable semantic signal in the absence of ground truth, suggesting a compelling path for test-driven transpilation and iterative repair.

5.2 Real-World Case Study

To evaluate the efficiency of our transpiler, we conducted a real-world study on an Apple M2 Pro (ARM64v8-A). This setup offers two advantages: (1) native ARM toolchain support, avoiding cross-compilation; and (2) Apple's Rosetta 2 layer, enabling consistent evaluation across execution modes on the same hardware. We assess performance across three environments: (i) native ARM64 binaries, (ii) x86 binaries via Rosetta 2, and (iii) GG-transpiled x86-to-ARM64 assembly. For each, we measure execution time, CPU energy (via `powermetrics`), and memory usage. Each program is executed 100 times, reporting the geometric mean (Fleming and Wallace, 1986), under controlled conditions.

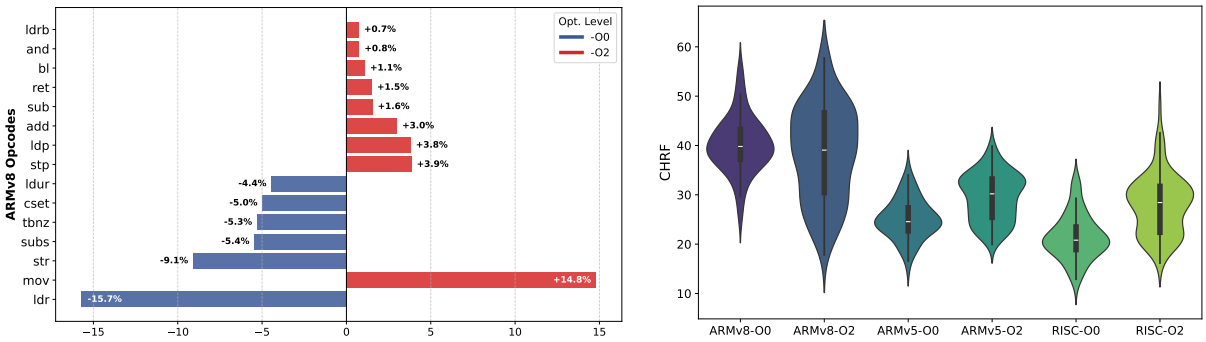
Figure 3 shows that GG achieves near-native performance: matching execution time, 1.73× faster than Rosetta, with 1.47× better energy efficiency and 2.41× better memory usage. GG's memory footprint (1.034 MB) is nearly identical to native (1.03 MB), while Rosetta uses 2.49 MB.

These results demonstrate that LLM-based binary translation offers a compelling alternative to traditional dynamic translation layers like Rosetta. Unlike Rosetta, which incurs a persistent runtime overhead, GG performs a one-time transpilation, avoiding the cumulative "runtime tax" and enabling leaner, faster execution. Moreover, our approach is general-purpose and untethered to Apple's ecosystem, enabling broader cross-ISA deployment and efficient CISC-to-RISC translation across diverse platforms. See Appendix A.1 for scaling, quantization, and error analysis.

5.3 Similarity Analysis Across ISAs

In Figure 4b, we observe that ARMv8 exhibits the highest average similarity to x86 (40.19%), followed by ARMv5 (25.09%) and RISC-V64 (21.41%). This gradient of similarity directly correlates with the drop in model accuracy from ARMv8 (99.39%) to ARMv5 (93.71%) and further down to RISC-V (89.63%). We hypothesize that this discrepancy is rooted in the increasing divergence in instruction semantics and register abstractions across these ISAs. ARMv8's shift toward CISC-like design (Red Hat, 2022) likely boosts its alignment with x86, aiding model generalization. In contrast, ARMv5 and RISC-V have simpler, more divergent instruction sets and addressing schemes, making the x86-to-RISC mapping less predictable and thus harder to learn.

Figure 4a highlights a significant shift in ARMv8 opcode usage between `-O0` and `-O2`. At `-O2`, `mov` becomes dominant (+14.8%), indicating more register reuse and reduced memory traffic via



(a) Opcode shift distribution in ARMv8

(b) CHRF similarity scores

Figure 4: Side-by-side comparison of opcode shift and CHRF similarity in ARM assembly analysis.

explicit `ldr/str`. This hides direct data movement, making it harder for the model to learn memory interaction. Paired instructions like `ldp/stp` appear more frequently, packing semantics into fewer lines, while conditional ops (`tbnz`, `cset`) are folded into predicated sequences. These changes, introduced by the compiler, abstract both control and data flow. We hypothesize that the model, trained only on `-O2`, must decode complex x86 semantics into a highly optimized and compressed ARMv8 form. This transformation increases learning difficulty and explains the drop in `-O2` accuracy (to 45.12%) despite strong `-O0` performance.

Model Variant	ARMv8 Accuracy	Impact (Δ)
Qwen2.5-Coder	0%	–
+ 1M AnghaBench	93.94%	+93.94%
+ 0.3M Stackv2	95.38%	+1.44%
+ RoPE Extrapolation	97.14%	+1.76%
+ Extended Tokenizer	98.18%	+1.04%
+ 8 Beam Search	**99.39%**	**+1.21%**

Table 5: Ablation study showing incremental improvements on ARMv8 accuracy from each added component.

5.4 Ablation Study

To understand what contributed most to model performance, we performed ablations shown in Table 5, focusing on four key aspects: training data size, RoPE extrapolation, the extended tokenizer, and decoding strategy.

First is the training data. As we increased the amount of training data to 1M AnghaBench, the accuracy jumps from 0% to 93.94%; including an additional 0.3M Stackv2 data points further improves accuracy to 95.38%. While effective, this scaling approach depends on high-quality, large-scale datasets and longer training time. Second is the architectural enhancement through RoPE Extrapolation, which pushes performance to 97.14%, indicating a +1.76% improvement. This suggests that enabling better generalization beyond the fixed context window substantially benefits instruction understanding and long-range dependency modeling.

The third contributing factor is tokenizer coverage: by extending the tokenizer to include additional subword units and symbols, we observe a further gain to 98.18%, adding +1.04%, highlighting the importance of adapting the tokenizer to the domain-specific vocabulary of assembly code. Finally, decoding strategy plays a non-trivial role; switching to 8-beam search yields the final boost to 99.39%, adding another +1.21%. Altogether, this progression shows that while data scaling gives the biggest leap, fine architectural and decoding choices compound gains toward near-perfect accuracy.

6 Conclusion

We introduce *Guaranteed Guess* (GG), a language-model-based CISC-to-RISC transpiler that unifies pre-trained LLMs with a test-driven validation framework. GG directly transpiles x86 assembly into efficient ARM and RISC-V binaries while embedding unit tests to enforce functional correctness. Through architectural enhancements, such as tokenizer extension, RoPE extrapolation, and beam decoding, GG achieves 99. 39% accuracy in HumanEval and 49. 23% in BringUpBench, outperforming both strong LLMs and dynamic virtualization systems like Rosetta. Our analysis highlights how ISA similarity and compiler optimizations affect accuracy, with GG achieving 1.73× faster execution, 1.47× lower energy use, and 2.41× smaller memory footprint than Rosetta on real-world binaries. These results position GG as a scalable, test-verified solution for efficient, cross-ISA binary translation.

7 Limitations

While *Guaranteed Guess* presents a significant advancement in CISC-to-RISC transpilation using LLMs, several limitations remain. First, the model's performance degrades substantially under compiler optimization flags (e.g., `-O2`), highlighting its sensitivity to code transformation patterns that abstract data and control flow. This suggests a need for stronger semantic modeling or auxiliary representations such as control/data-flow graphs. Second, the "guarantee" provided by **GG** is inherently bounded by the quality and coverage of the unit tests. While unit test success is a strong functional proxy, it cannot ensure full semantic equivalence or optimality of the transpilation. Lastly, the evaluation excludes compiler-, symbolic-, or heuristic-based transpilation baselines, leaving open questions about hybrid system effectiveness and competitive upper bounds.

References

Apple Inc. 2020. Apple's rosetta 2 overview. Accessed: 2024-10-31.

Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O'Boyle. 2024. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembly. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

Todd Austin. 2024. bringup-bench.

Fabrice Bellard. 2005. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*.

Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. 2013. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

CloudPanel. 2023. What are arm-based servers? comparison with x86, benefits and drawbacks. Accessed: 2024-10-31.

Matthew Connatser. 2023. Intel's ceo says moore's law is slowing to a three-year cadence, but it's not dead yet. *Tom's Hardware*.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. 2024. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*.

Anderson Faustino Da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quinão Pereira. 2021. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390. IEEE.

Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*.

Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. 1974. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of solid-state circuits*, 9(5):256–268.

DeepSeek-AI et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Philip J Fleming and John J Wallace. 1986. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*.

Ggerganov. Github - ggerganov/llama.cpp: Llm inference in c/c++. `https://github.com/ggerganov/llama.cpp`. Accessed: 2024-10-31.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2021. Chasing carbon: The elusive environmental footprint of computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867. IEEE.

Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680.

Mark Horowitz. 2014. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pages 10–14. IEEE.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. 2024. Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989*.

Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium (2024). https://api. semanticscholar. org/CorpusID*, volume 267622140.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024a. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024b. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

IONOS. 2024. Arm processor architecture explained. https://www.ionos.com/digitalguide/server/know-how/arm-processor-architecture/. Accessed: 2025-04-12.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and 1 others. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, and 1 others. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

Chris Lattner. 2008. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, pages 1–20.

Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M Rush. 2024. Guess & sketch: Language model guided transpilation. In *The Twelfth International Conference on Learning Representations*.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, and 1 others. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. 2025. CraftRTL: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair. In *The Thirteenth International Conference on Learning Representations*.

I Loshchilov. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Alfonso Maruccia. 2025. Arm is aiming to win half of data center cpu market by year's end. Accessed: 2025-05-20.

Federico Mora, Justin Wong, Haley Lepe, Sahil Bhatia, Karim Elmaaroufi, George Varghese, Joseph E. Gonzalez, Elizabeth Polgreen, and Sanjit A. Seshia. 2024. Synthetic programming elicitation for text-to-code in very low-resource programming and formal languages. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Timothy Prickett Morgan. 2022. Inside amazon's graviton3 arm server processor. *The Next Platform*.

Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

NVIDIA Corporation. 2024. NVIDIA Grace CPU and Arm Architecture. https://www.nvidia.com/en-us/data-center/grace-cpu/. Accessed: 2025-04-12.

OpenAI. 2024. Hello gpt4-o. https://openai.com/index/hello-gpt-4o/. Accessed: 2024-10-31.

David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*.

GNU Project. 2025. riscv64-linux-gnu-gcc: The gnu compiler collection for risc-v (64-bit). https://gcc.gnu.org/. Accessed: 2025-04-12.

Radcolor. n.d. Radcolor/ARM-linux-gnueabi: Bleeding edge GNU gcc toolchains (cc only) built from sources with latest binutils and glibc (for arm). https://github.com/radcolor/arm-linux-gnueabi. GitHub.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506.

Red Hat. 2022. Arm vs x86: What's the difference? Accessed: 2025-05-19.

Reuters. 2025. Arm expects its share of data center cpu market to surge as sales rocket 50% this year. https://www.reuters.com/technology/arm-expects-its-share-data-center-cpu-market-sales-rocket-50-this-year-2025-03-31/. Accessed: 2025-04-12.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Phillip Rust, Jonas Pfeiffer, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych. 2020. How good is your tokenizer? on the monolingual performance of multilingual language models. *arXiv preprint arXiv:2012.15613*.

Richard L Sites, Anton Chernoff, Matthew B Kirk, Maurice P Marks, and Scott G Robinson. 1993. Binary translation. *Communications of the ACM*, 36(2):69–81.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. *arXiv*.

Sourabh Kumar Verma. 2024. Exploring windows on arm: The future of computing. https://techcommunity.microsoft.com/blog/educatordeveloperblog/exploring-windows-on-arm-the-future-of-computing/4260186. Microsoft Tech Community Blog.

Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Zhe Wang, John Smith, and Jane Doe. 2024. Evaluating the effectiveness of decompilers. In *Proceedings of the 2024 ACM Conference on Software Analysis*, New York, NY, USA. ACM.

Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving binary code similarity transformer models by semantics-driven instruction deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*.

Siyuan Zheng, Zhi Yang, Cedric Renggli, Yuxiang Pu, Zixuan Li, Mohammad Shoeybi, Lin Zhang, Dheevatsa Narayanan, Haotian Zhao, Zhewei Yao, and Tianqi Chen. 2023. vllm: A high-throughput and memory-efficient inference engine for llms. https://github.com/vllm-project/vllm. GitHub repository.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*.

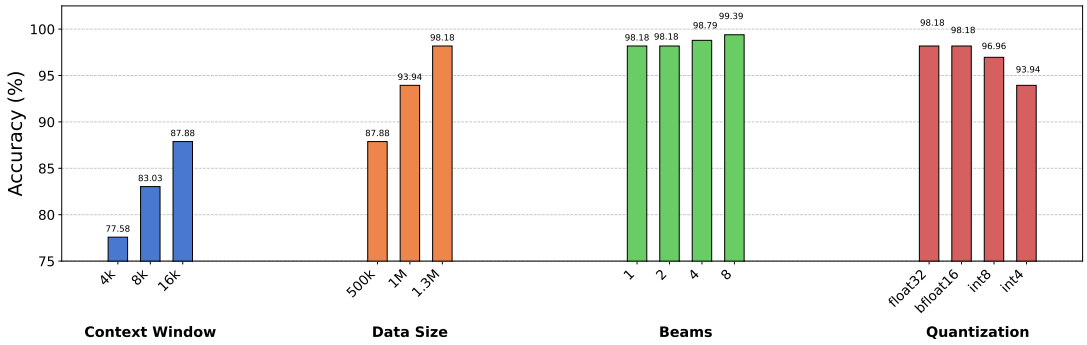


Figure 5: **Impact of scaling and quantization on Qwen2.5-Coder 1.5B variant evaluated using the *code coverage* metric on HumanEval with -O0 compiler optimization.**

A Appendix

A.1 Extra Data Analysis

Scaling and quantization effect on Qwen2.5-coder models. Figure 5 represents an study to understand where most of the training benefit for our transpiler originates. In particular, we focus on three fundamental modeling aspects and describe their impact on the asm-to-asm transpiler.

Our first and most significant result relates to the context window size, and its impact on the transpiler. Recall that a model's context window is the amount of text, in tokens, that the model can consider or "remember" at any one time. We found that programs do not fully fit in the context window (which includes both the input and output of the model, i.e., the x86 asm and the generated ARM asm), are very likely to not pass all our tests. Increasing the context window length during training had a big impact on our model's accuracy, where going from 4k to 16k improved the total number of fully correct transpiled programs by 10% points, roughly an additional 16 programs out of the 164 total in HumanEval.

The second effect of scaling we observed and leveraged was that training on more data also played a major role in our transpiler's efficacy. As shown in Figure 5, using a context window of 16k and increasing the training data from 500k samples to 1.3 million samples further increased and pushed the accuracy up to about 98% from 87%. This is generally a challenging method of scaling, as obtaining more data with good quality is not always available and also results in increased total training time of the model.

The third scaling impact we found was the benefit of increasing the number of beams and doing a beam search. Beam search is a heuristic search algorithm which allows the model to explore multiple token paths in parallel during an inference. Intuitively, beam search allows the model to explore alternative options for next token generation, settling on the most likely token. Beam searching presents an obvious trade-off between computational resources utilization for an inference and prediction accuracy. Combined with a large context window, this is a very powerful technique which we found to be more pronounced when a model was not already near perfect accuracy: in Figure 5, we show an increase going up to 99.39% with the use of beam search for assembly transpilation. We found diminishing returns for using more than 4 beams on accuracy.

Finally, from an efficiency perspective, we show that aggressive quantization does not severely impact our transpilers accuracy. Going from FP32 down to INT4 substantially reduces the transpilers inference footprint, with a minimal (less than

Prog ID	Edit Dist	Example
P37	1	**Incorrect immediate value causes wrong division factor and early loop termination** Ground truth: `asr r2, r2, #2` Predicted: `asr r2, r2, #1`
P127	1	**Array index offset error causes wrong element comparison** Ground truth: `sub r3, r3, #2` Predicted: `sub r3, r3, #1`
P63	12	**Register overwrite corrupts loop counter before multiplication** Ground truth: `mov r0, r2; ldr r1, [r3, r1, lsl #2]; mul r0, r0, r1` Predicted: `ldr r0, [r3, r1, lsl #2]; mul r0, r0, r1`
P153	17	**Incorrect instruction sequence fails to compute absolute value** Ground truth: `sub r2, r2, r3; cmp r2, #0; rsblt r2, r2, #0` Predicted: `sub r1, r2, r3; eor r2, r1, r2; sub r2, r2, r1`
P47	19	**Mismatched memory access offsets cause incorrect data retrieval** Ground truth: `str r1, [fp, #-404]; ldr r2, [fp, #-404]` Predicted: `str r1, [fp, #-404]; ldr r2, [r3, #-20]`

Table 6: Armv5 Syntactically similar generations can still produce critical semantic errors.

4%) impact on model prediction accuracy. This shows the potential of designing small enough models for deployment on edge devices, which we would envision the GG transpiler to be used for CISC-to-RISC translations in practice.

Transpilation Error Analysis. We provide a detailed analysis of functionally equivalent predictions produced by our model that deviate syntactically from the ground truth. Such cases reveal the model's ability to generalize instruction patterns while maintaining semantic correctness, a desirable trait in low-level code generation where multiple implementations can achieve the same functional outcome.

Prog ID	Edit Dist	Example
P108	16	**Different registers can be chosen for temporary values while maintaining same data flow** Ground truth: `mov r2, r0; add r2, r2, #1` Predicted: `mov r3, r0; add r3, r3, #1`
P8	12	**Local variables can be stored at different stack locations while maintaining correct access patterns** Ground truth: `str r1, [fp, #-8]; str r2, [fp, #-12]` Predicted: `str r1, [fp, #-12]; str r2, [fp, #-8]`
P119	6	**Compiler-generated symbol names can differ while referring to same data** Ground truth: `.word out.4781` Predicted: `.word out.4280`
P135	12	**Multiple instructions can be combined into single equivalent instruction** Ground truth: `mov r3, r0; str r3, [fp, #-8]` Predicted: `str r0, [fp, #-8]`
P162	4	**Stack frame offsets can vary while maintaining correct variable access** Ground truth: `strb r3, [fp, #-21]` Predicted: `strb r3, [fp, #-17]`
P88	23	**Memory allocation sizes can vary if sufficient for program needs** Ground truth: `mov r0, #400` Predicted: `mov r0, #800`
P103	52	**Different instruction sequences can achieve same logical result** Ground truth: `cmp r3, #0; and r3, r3, #1; rsblt r3, r3, #0` Predicted: `rsbs r2, r3, #0; and r3, r3, #1; and r2, r2, #1; rsbpl r3, r2, #0`
P69	50	**Constants can be loaded directly or from literal pool** Ground truth: `mvn r3, #-2147483648` Predicted: `ldr r3, .L8; .L8: .word 2147483647`

Table 7: Simple Variation Patterns in Functionally Equivalent Code

Table 7 enumerates a range of examples with moderate edit distances, where syntactic differences arise from register allocation, operand ordering, and memory layout choices. For instance, the model often selects different temporary registers (e.g., `r3` instead of `r2`) or reorders commutative operands without altering the underlying operation. It also adjusts stack frame offsets or memory allocation sizes, provided that the modifications do not violate data dependencies or correctness constraints.

These variations suggest that the model is not merely memorizing instruction patterns but is instead learning high-level register-to-variable mappings and instruction equivalence classes. This flexibility enables generalization beyond the exact reference format and increases robustness to minor program transformations.

Prog ID	Edit Dist	Combined Patterns and Examples

P128	78	**Multiple Optimization Patterns:**
		Groud truth: `mul r1, r2, r3`
		Predicted:
		`lsl r1, r2, #2;`
		`add r1, r1, r2`

P113	74	**Memory and Instruction Patterns:**
		Ground truth:
		`str r1, [fp, #-12]`
		`mov r3, r2`
		`add r3, r3, #4`
		Predicted:
		`str r1, [fp, #-8]`
		`add r2, r2, #4`

Table 8: Complex Variation Patterns with Multiple Differences

Furthermore, Table 8 presents more substantial structural rewrites that nonetheless retain functional fidelity. These include compound transformations such as converting multiplications into equivalent shift-add sequences, or restructuring memory operations while preserving access order and scope. In one example, a multiplication instruction is replaced with a pair of shift and add instructions demonstrating the model's awareness of performance-equivalent alternatives. In another case, memory writes and register arithmetic are reordered while maintaining the intended result, revealing the model's competence in preserving state consistency across instruction sequences.

While these examples have higher edit distances, they exemplify a deeper form of equivalence: one grounded in operational semantics rather than surface-level syntax. The ability to produce such alternative forms underscores the potential of language models to reason compositionally about program structure and to synthesize diverse yet correct outputs for the same task.

In contrast, Table 6 presents failure cases where minor syntactic deviations result in critical semantic errors. These include incorrect immediate values,

register mismanagement, and mismatched memory offsets that compromise program correctness despite appearing superficially similar to the ground truth.

Together, Tables 7, 8, and 6 reveal that syntactic deviation does not necessarily imply failure. On the contrary, these examples support the argument that token-level metrics alone are insufficient to evaluate low-level transpilation tasks, and that functional correctness should take precedence in model assessment.