NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

MSc THESIS

A configurable assembly program generator for x86 microprocessor architectures

Nikolaos D. Karystinos-Avgerantonis

Supervisor: **Dimitris Gizopoulos,** Professor

ATHENS

OCTOBER 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μια παραμετροποιήσιμη γεννήτρια προγραμμάτων συμβολικής γλώσσας για μικροεπεξεργαστές αρχιτεκτονικής x86

Νικόλαος Δ. Καρυστινός-Αυγεραντώνης

Επιβλέπων: **Δημήτρης Γκιζόπουλος,** Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2024

MSc THESIS

A configurable assembly program generator for x86 microprocessor architectures

Nikolaos D. Karystinos-Avgerantonis
S.N.: cs3190003

SUPERVISOR: **Dimitris Gizopoulos,** Professor

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μια παραμετροποιήσιμη γεννήτρια προγραμμάτων συμβολικής γλώσσας για μικροεπεξεργαστές αρχιτεκτονικής x86

Νικόλαος Δ. Καρυστινός-Αυγεραντώνης

Α.Μ.: cs3190003

ΕΠΙΒΛΕΠΩΝ: **Δημήτρης Γκιζόπουλος,** Καθηγητής

ABSTRACT

Automatic program generation is a useful hammer for various nails. Ranging from producing power viruses (as done in [4]) to fuzzing the input of a compiler [24], a hardware model [23], or even executing the generated programs on an actual microprocessor to verify its operation [3, 7, 6], automatic program generation presents itself as the ideal tool.

Manually specifying programs for any of the above tasks would be a tedious, error-prone, repetitive and time-consuming process for a human which may deliver effective programs only in specific use cases. A configurable program generator alleviates these challenges by producing thousands or even millions of program variants rapidly. These programs conform to user-specified templates and constraints and are bug-free (provided the template and the code generator are correct). Moreover, the fast automated generation process significantly accelerates testing cycles, enabling exhaustive exploration of potential edge cases.

Regarding assembly generation, code generators bridge another significant gap: writing assembly requires the programmer to be able to reason about the underlying architecture (registers, instruction formats and operands, memory references, etc.). Imagine trying to write out all x86 assembly sequences of length 10 that only use instructions with integer register operands. It probably takes more than an evening to even pinpoint the relevant instructions. The code generator abstracts away this complexity and can produce such sequences with directives that would closely resemble the language description we gave above.

In this thesis, we present the development and experimental evaluation of a configurable and modular x86 assembly program generator. Our generator is based on the *MicroProbe* framework [1, 4] extending its capabilities from RISC ISAs (RISC-V, PowerPC) to CISC ISAs like x86. The prevalence of the x86 ISA in current computing infrastructures makes an x86 code generator a powerful tool in the hands of researchers aiding in answering a diverse array of questions regarding x86 microprocessors or even helping to uncover previously unknown flaws in their operation.

SUBJECT AREA: Code Generation, Computer Architecture, Programming Languages, Intruction Set Architecture

KEYWORDS: code generation, assembly, x86, microprocessors

ΠΕΡΙΛΗΨΗ

Η αυτόματη δημιουργία προγραμμάτων είναι ένα χρήσιμο εργαλείο για διάφορες προκλήσεις. Από την παραγωγή power viruses (όπως έγινε στο [4]) μέχρι τη δοκιμή εισόδου ενός μεταγλωττιστή [24] ή ακόμη και την εκτέλεση των παραγόμενων προγραμμάτων σε έναν πραγματικό μικροεπεξεργαστή για την επαλήθευση της λειτουργίας του [3, 7, 6], η αυτόματη δημιουργία προγραμμάτων αποδεικνύεται το ιδανικό εργαλείο.

Η χειροκίνητη δημιουργία προγραμμάτων για οποιαδήποτε από τις παραπάνω εργασίες θα ήταν μια κουραστική, επιρρεπής σε λάθη, επαναλαμβανόμενη και χρονοβόρος διαδικασία για έναν άνθρωπο. Μια παραμετροποιήσιμη γεννήτρια προγραμμάτων αντιμετωπίζει αυτές τις προκλήσεις παράγοντας χιλιάδες ή ακόμα και εκατομμύρια παραλλαγές προγραμμάτων σε ελάχιστο χρόνο. Αυτά τα προγράμματα συμμορφώνονται με τα πρότυπα και τους περιορισμούς που έχουν καθοριστεί από τον χρήστη και είναι χωρίς σφάλματα (υπό την προϋπόθεση ότι το πρότυπο και η γεννήτρια είναι σωστά). Επιπλέον, η ταχύτατη διαδικασία δημιουργίας επιταχύνει σημαντικά τους κύκλους δοκιμών, επιτρέποντας την εξαντλητική εξερεύνηση πιθανών οριακών περιπτώσεων.

Σχετικά με τη δημιουργία κώδικα assembly, οι γεννήτριες κώδικα γεφυρώνουν ένα ακόμη σημαντικό κενό: η συγγραφή κώδικα assembly απαιτεί από τον προγραμματιστή να μπορεί να κατανοήσει την υποκείμενη αρχιτεκτονική (καταχωρητές, μορφές εντολών και τελεστέους, αναφορές μνήμης, κ.λπ.). Φανταστείτε να προσπαθείτε να γράψετε όλες τις ακολουθίες x86 assembly μήκους 10 που χρησιμοποιούν μόνο εντολές με τελεστέους ακέραιους καταχωρητές. Πιθανόν να χρειαστείτε περισσότερο από ένα βράδυ απλώς για να εντοπίσετε τις σχετικές εντολές. Ο δημιουργός κώδικα αφαιρεί αυτήν την πολυπλοκότητα και μπορεί να παράγει τέτοιες ακολουθίες με οδηγίες που θα μοιάζουν στενά με την περιγραφή σε φυσική γλώσσα που δώσαμε παραπάνω.

Σε αυτήν τη διατριβή, παρουσιάζουμε την ανάπτυξη και την πειραματική αξιολόγηση ενός ιδιαίτερα παραμετροποιήσιμου και αρθρωτού προγράμματος παραγωγής κώδικα assembly για την αρχιτεκτονική x86. Ο δημιουργός μας βασίζεται στο framework MicroProbe [1, 4], επεκτείνοντας τις δυνατότητές του από τις RISC ISAs (RISC-V, PowerPC) στις CISC ISAs όπως η x86. Η επικράτηση της ISA x86 στις τρέχουσες υπολογιστικές υποδομές καθιστά μια γεννήτρια κώδικα x86 ένα ισχυρό εργαλείο στα χέρια των ερευνητών, βοηθώντας στην απάντηση μιας ποικιλίας ερωτημάτων σχετικά με τους μικροεπεξεργαστές x86 ή ακόμη και στην αποκάλυψη άγνωστων μέχρι τώρα ελαττωμάτων στη λειτουργία τους.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γεννήτριες Κώδικα, Αρχιτεκτονική Υπολογιστών, Γλώσσες Προγραμματισμού, Αρχιτεκτονική Συνόλου Εντολών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: γεννήτριες κώδικα, συμβολική γλώσσα, x86, μικροεπεξεργαστές

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, professor Dimitris Gizopoulos, for his invaluable guidance and direction throughout the development of this thesis. Further, I would like to thank Dr. George Papadimitriou for taking the time to perform a thorough review of the contents which has been instrumental in shaping the clarity and quality of the final version.

I am also deeply grateful to Ramon Bertran, the primary author of MicroProbe, for generously taking the time to introduce me to the framework and for his assistance in addressing the many questions that arose during my work.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

An automatic program generator is in essence a program (software) that, when given a set of directions and/or constraints - usually algorithmically - produces programs in a target language, either high-level (e.g., C, Java) or low-level (e.g., assembly, machine-code). The details and intricacies of the target language are handled by the program generator (or code generator). At the same time, the programmer is minimally involved, merely specifying a generation strategy, creating a template or imposing some constraints on the generation. One common approach is randomly generating programs of the target language, optionally with the presence of a set of constraints. This process is called *constrained random generation* and is widely used in various domains. Automatic program generators are invaluable tools both in the software [24, 16] as well as in the hardware space [7, 6, 3, 22].

In software, they are most often used to fuzz programs that accept other programs as input (e.g., compilers, assemblers, code transformation tools) [8, 15, 13]. Fuzzing is essentially a software testing technique that involves automatically generating and inputting large amounts of random or semi-random data (known as ”fuzz”) into a program to discover vulnerabilities, bugs, or unexpected behaviors. The goal of fuzzing is to expose edge cases or errors that might not be found through traditional testing methods, by overwhelming the program with varied and often unexpected inputs [14].

Another significant application of automatic program generation in software is in automated refactoring [5, 9]. In large and complex software projects, maintaining code quality and consistency is a continuous challenge. Automated refactoring tools use program generation techniques to systematically improve the codebase, optimizing performance, enhancing readability, and ensuring adherence to coding standards. These tools can also assist in migrating legacy code to modern practices or new programming languages, making it easier to maintain and extend the software over time. By automating the process, developers can focus on higher-level design and problem-solving, rather than getting bogged down in tedious and error-prone manual refactoring tasks.

Turning our focus to applications in hardware, program generators often target the assembly language of the microprocessor, bypassing the compiler to gain full control over the instructions executed on the hardware. This direct approach allows for precise manipulation of the instruction sequences, which is crucial in many tasks. For instance, by generating millions of tight-looping sequences and selecting the ones with the highest power consumption during execution, one can stress-test the microprocessor’s power efficiency and thermal limits, thus constructing power viruses [4, 17].

One of the most well-known applications of program generators in hardware is functional testing [12, 20, 18, 21] . Microprocessor functional testing involves verifying that a microprocessor operates according to its design specifications and correctly performs all intended functions [12, 20, 10, 11]. This testing ensures that the microprocessor executes instructions, processes data, and interacts with memory and other components as expected, without errors or unintended behavior. During functional testing, a large number of generated programs are fed to the microprocessor design under test (DUT), and the traces, outputs and resulting microprocessor state are meticulously examined for deviations. The primary objective is to detect design flaws, logic errors, and other potential issues before the microprocessor is mass-produced or deployed in critical applications, and at later stages to detect manufacturing defects, latent faults, and lifetime reliability (aging) related defects. By rigorously testing all functional aspects, engineers can confirm

that the microprocessor will perform reliably in real-world scenarios, thereby minimizing the risk of failures that could lead to system crashes or data corruption.

It is evident from the above that code generators are powerful tools with wide applicability in various areas. In this thesis, we focus on extending the open-source code generation framework MicroProbe [1, 4], thereby adding support for the x86 instruction set architecture (ISA) the prevailing ISA in cloud computing and desktop computing. MicroProbe is written in Python and can produce assembly programs in the RISC-V or PowerPC ISAs. The generator is guided by what is referred to as a *policy*, which consists of a set of compiler-like *passes*. These *passes* iteratively transform MicroProbe's internal representation of the program into valid assembly for the target ISA. For example, a pass can perform register allocation (choosing which register operands to assign to each instruction) with a set dependency distance. The crux feature of MicroProbe is the decoupling of these passes from information that is specific to an ISA, making the passes highly reusable even across different ISAs.

To our knowledge, there are currently no other open-source generators for the x86 ISA with the configurability and extensibility that MicroProbe provides. Thus, our extension of MicroProbe fills in a significant gap in current open-source tooling.

The rest of the thesis is organized as follows:

- In chapter 2, we give some background on the x86 ISA as well as the YAML file format both of which are heavily utilized in later sections as well as some commentary on related work.
- In chapter 3, we introduce the MicroProbe framework and analyze its components and program generation flow.
- In chapter 4, we go into details regarding the additions and modifications we performed in the framework to add support for the x86 ISA.
- In chapter 5, we show the optimization process of the performance of our code generator and proceed to evaluate it via a set of experiments.
- In chapter 6, we conclude this thesis and give some possible directions for the evolution of our tool.

2. BACKGROUND AND RELATED WORK

2.1 x86 assembly language

2.1.1 Introduction

The x86 architecture is a widely used *instruction set architecture* (ISA) developed by Intel and subsequently extended by AMD. Originally introduced in 1978 with the Intel 8086 processor, it has evolved through multiple generations, maintaining backward compatibility while expanding its features. The x86 ISA is primarily used in desktop, laptop, and server systems, making it one of the most dominant computing architectures globally.

x86 is a *Complex Instruction Set Computer* (CISC) architecture. CISC architectures like x86 are characterized by their large set of variable length instructions, many of which are quite complex. These instructions can perform multiple low-level operations (such as memory access, arithmetic, and control flow) in a single instruction. This can simplify assembly programming and decrease code size but tends to increase the complexity of the hardware implementation.

Some key characteristics of x86 [2] include:

- **Variable-length instructions:** x86 instructions can range from 1 to 15 bytes, allowing for efficiently encoding simple and more complex operations but making instruction decoding more cumbersome.
- **Rich instruction set:** x86 supports hundreds of instructions, including both basic operations (like addition) and specialized tasks (like string manipulation or floating-point operations).
- **Backward compatibility:** Each new generation of x86 processors supports older instruction sets, providing seamless software compatibility across decades of hardware.
- **Multiple complex addressing modes**: Compared to RISC ISAs x86 does not include seperate load/store instructions. Instead, the memory operands of instructions can address memory in various complex manners.

In the following subsections, we will provide an introduction to basic concepts in x86 (registers, addressing, etc.). For a full reference on these subjects, the reader is encouraged to consult the official x86-64 manuals [2].

2.1.2 x86 (CISC) vs RISC

In contrast to x86, Reduced Instruction Set Computer (RISC) architectures, such as MIPS, ARM, SPARC, or the recent RISC-V, use a smaller, more uniform set of instructions. RISC architectures are designed to execute instructions in a single clock cycle, leading to leaner, simpler to design and faster pipelines.

Some key differences between x86 (CISC) and RISC:

- **Instruction complexity:** x86 has complex, variable-length instructions, which use various instruction formats while RISC instructions are simple, mostly canonical in formats and fixed-length, usually 4 bytes.
- **Hardware complexity:** The x86 CISC architecture, as mentioned earlier, requires much more complex decoding hardware due to its variable-length instructions. RISC simplifies hardware with its streamlined instruction set, allowing for higher clock speeds.
- **Efficiency:** RISC ISAs often rely on software to break down complex tasks into simpler instructions, potentially leading to more instructions executed for the same task, but with greater overall efficiency in modern systems.
- **Power consumption:** RISC processors, such as those based on ARM, tend to be more power-efficient, making them ideal for mobile and embedded systems. In contrast, x86 processors, while highly performant, consume more power and are more commonly found in desktops and servers.

To give further intuition on the RISC vs. CISC approaches, we can consider the classic performance equation [19]:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}} \tag{2.1}$$

The CISC approach attempts to minimize the number of instructions per program, i.e., the last term in (2.1). CISC instructions can perform complex functions, so if you choose right, even complex tasks can be accomplished in a few instructions. However, this pessimizes the number of cycles per instruction; the complex decoding hardware takes its toll on the cycles necessary for decoding and executing a CISC instruction.

RISC aims for the opposite, reducing the cycles per instruction as close to 1 as possible for almost every instruction, but at the cost of the higher number of instructions for the same program compared to the CISC philosophy.

To showcase how these differences manifest themselves in the actual assembly of each approach, let us look at the example of how string comparison might be implemented in x86 (CISC) vs. RISC-V (RISC).

In RISC-V, the comparison of two strings must be done explicitly using load and branch instructions. Here's a simple example that compares two strings byte-by-byte until a difference is found or a null terminator is encountered.

```
# RISC-V Assembly (string comparison)
# a0: pointer to string1
# a1: pointer to string2
# a2: result (0 if equal, 1 if not equal)

compare_strings:
    lb t0, 0(a0)        # Load byte from string1
    lb t1, 0(a1)        # Load byte from string2
    beq t0, t1, skip    # If bytes are equal, continue
    li a2, 1            # Set result to 1 (strings not equal)
    ret                 # Return
skip:
    beqz t0, done       # If we reached null terminator, strings are equal
    addi a0, a0, 1      # Move to next byte in string1
    addi a1, a1, 1      # Move to next byte in string2
    j compare_strings   # Repeat comparison
done:
    li a2, 0            # Set result to 0 (strings are equal)
    ret
```

Figure 2.1: RISC-V String Comparison Example

On the contrary, string comparison in x86 is more compact due to dedicated string instructions like ‘CMPSB‘, which compare byte-by-byte, and can be paired with a loop automatically.

```
; x86 Assembly (string comparison)
; RSI: pointer to string1
; RDI: pointer to string2
; AL: result (0 if equal, 1 if not equal)

compare_strings:
    cld                 ; Clear direction flag for forward comparison
    repe cmpsb          ; Compare bytes in strings while equal
    jne not_equal       ; Jump if strings are not equal
    xor eax, eax        ; Set result to 0 (strings are equal)
    ret
not_equal:
    mov eax, 1          ; Set result to 1 (strings not equal)
    ret
```

Figure 2.2: x86 String Comparison Example

- The RISC-V version uses explicit load and branch instructions in a loop to compare the strings byte-by-byte. This makes the code longer and more verbose, requiring multiple instructions to load data, compare, branch, and handle the result.
- The x86 version benefits from specialized string instructions (‘CMPSB‘ and ‘REPE‘) that automate the byte-by-byte comparison in a much shorter sequence. The instruction set includes implicit looping and memory handling, making the code more compact.

2.1.3 x86 programming

2.1.3.1 Registers

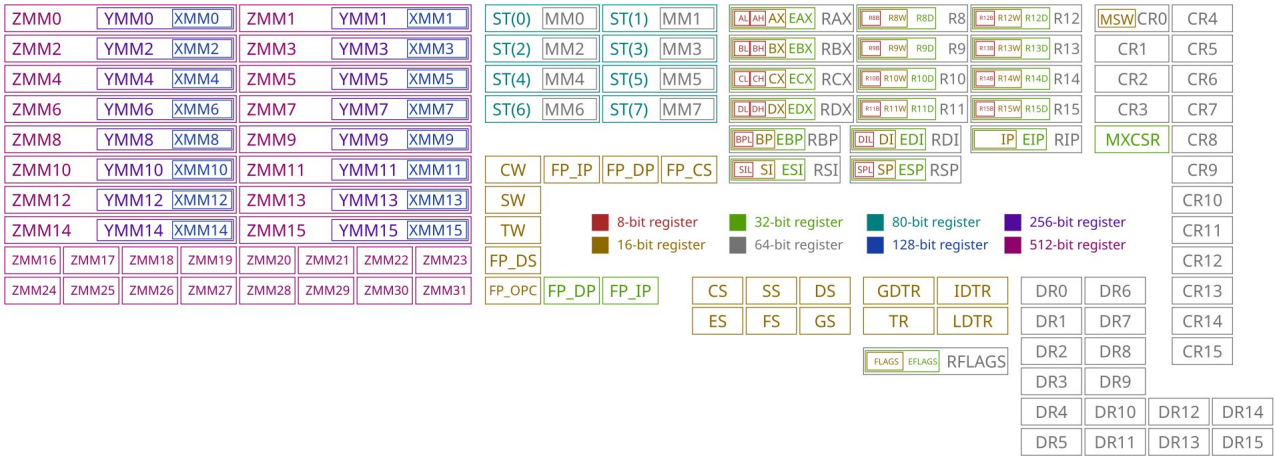

Figure 2.3: x86 Registers

Fig.2.3 illustrates the complete set of architectural registers available in the 64-bit x86 instruction set architecture and its extensions (x86-64). The following sections provide a detailed overview of these registers, categorized by their function and relevance. We begin by discussing the most critical registers, then proceed to describe more specialized registers introduced by optional extensions to the architecture; for instance, the 512-bit `ZMM` registers, located in the upper left of Fig.2.3, are part of the AVX extension, which is not universally implemented in all x86 processors.

General-Purpose Registers: This category of registers is crucial for data manipulation and are found in the upper right-hand section of Fig. 2.3. They are most often used to hold operands for logical and arithmetic operations, facilitate address calculation or store memory pointers. They include:

- *64-bit registers*: `RAX`, `RBX`, `RCX`, `RDX`, `RSP`, `RBP`, `RSI`, `RDI`, `R8-R15` (gray boxes). These are the primary registers used in 64-bit operations. For example, `RAX` is often used as an accumulator, `RBX` is the base register, and `RSP` is the stack pointer.
- *32-bit registers*: `EAX`, `EBX`, `ECX`, `EDX`, `ESP`, `EBP`, `ESI`, `EDI` and `R8D-R15D` (green boxes). These are the lower 32-bits of the 64-bit registers and are used in operations requiring 32-bit data widths.
- *16-bit registers*: `AX`, `BX`, `CX`, `DX`, `SP`, `BP`, `SI`, `DI` and `R8W-R15W` (brown boxes). These registers are often used for legacy 16-bit code, and their use in modern 64-bit systems is more limited.
- *8-bit registers*: `AH`, `AL`, `BH`, `BL`, `CH`, `CL`, `DH`, `DL` and `R8B-R15B` (red boxes). These are used when 8-bit operations are needed, typically in legacy code.

It is particularly important to note the **hierarchical relationships** between the general-purpose registers. These are portrayed in Fig. 2.3 by enclosing each ”sub”-register name in the full register whose lower bits it is aliasing. For example, looking at the 64-bit general-purpose register `RAX`, we can see that it encloses the 32-bit register `EAX`; thus the lower 32 bits of `RAX` are the bits contained in `EAX`. Further, `AX` is the lower 16-bits of `EAX`, `AH` is the

high-byte of AX and AL is the low-byte of AX. This hierarchical structure allows for flexible operation across different bit-widths within the same register framework.

Floating Point Registers: In the central section of Fig. 2.3, we have the floating-point registers (blue and teal boxes), and the control and status registers (below them) related to the floating-point operations. These registers are used for floating-point arithmetic but they also contain the registers for the MMX extension. Specifically:

- *80-bit ST registers (ST0-ST7)*: These are part of the x87 floating-point unit and are used for floating-point arithmetic with 80-bit accuracy.
- *MMX registers (MM0-MM7)*: These are 64-bit multimedia registers designed to handle integer-based SIMD (Single Instruction, Multiple Data) operations.
- *Control and Status Registers (CW, SW, and TW)*: These manage the control and status of the x87 floating-point unit.

Control Registers: Located on the right side of Fig. 2.3, the control registers (CR0-CR15) manage system-level functionality, such as memory management and operating modes of the processor.

SIMD Registers: In the upper-left section of Fig. 2.3, we have the SIMD registers used for vectorized operations, particularly in multimedia, graphics, and scientific computation. These registers are designed to handle multiple data elements simultaneously. Specifically:

- *128-bit XMM registers (XMM0-XMM15)*: These are the smallest SIMD registers and support operations like addition, subtraction, and multiplication on packed floating-point or integer data. They are part of the first versions of the SSE extension for SIMD operations.
- *256-bit YMM registers (YMM0-YMM15)*: These registers extend the XMM registers to 256 bits, allowing for more data processing in a single instruction. These are part of the later versions of SSE which introduces 256-bit registers (double the width of XMM which is 128).
- *512-bit ZMM registers (ZMM0-ZMM31)*: The most extended SIMD registers, used in conjunction with AVX-512 instructions (AVX-512 is an ISA extension) for high-throughput vector processing.

Segment Registers: These are shown at the bottom center of Fig. 2.3 and include CS, SS, DS, ES, FS, and GS. These registers hold segment selectors, used for memory segmentation (a legacy feature that is rarely used in modern 64-bit systems).

Debug Registers: DR0-DR7, located at the bottom right of Fig. 2.3, are used for hardware-based debugging by setting breakpoints.

2.1.3.2 Syntax Variants

In x86 assembly programming, two primary syntax variants are widely recognized: Intel syntax and AT&T syntax. These syntax conventions dictate how assembly instructions and their operands are expressed, and understanding their distinctions is essential for accurate code interpretation and development.

Intel syntax is predominantly used in Windows and some cross-platform tools. It is characterized by its emphasis on readability and alignment with common mathematical notation. In Intel syntax, instructions are written with the destination operand preceding the source operands. This format is intuitive for many programmers, as it mirrors the standard algebraic notation, where the result is presented first. For example, the instruction to move an immediate value into a register is expressed as `MOV EAX, 0x5`, where `EAX` is the destination register and `0x5` is the immediate value being moved.

Conversely, AT&T syntax, which is commonly used in Unix-like systems and in GNU assembler (GAS), adopts a different approach. This syntax places the source operands before the destination operand, which aligns with the operand order used in the assembly language's underlying machine code. Additionally, AT&T syntax employs different mnemonics and operand formatting conventions. For instance, the same operation in AT&T syntax would be written as `movl $0x5, %eax`. Notably, AT&T syntax also uses a prefix (such as `$` for immediate values) and a different format for registers, with percent signs (`%`) preceding register names.

These syntactic differences reflect varying historical and practical preferences, impacting how assembly code is written, read, and converted between formats. Intel syntax's operand order and formatting may be perceived as more straightforward, particularly for those accustomed to traditional mathematical expressions. In contrast, AT&T syntax's operand order and notation are deeply integrated with the conventions used in Unix-like systems and its associated development tools. Mastery of both syntax variants is essential for effective cross-platform assembly programming and for comprehending legacy codebases that utilize different assembly conventions.

For the purposes of this thesis, we adopt the Intel syntax from now on. However, the generator can be configured to produce AT&T syntax programs.

2.1.3.3 Addressing Modes

The x86-64 architecture features a complex set of addressing modes, which are crucial for determining how memory is accessed. The fundamental addressing modes revolve around using registers, constants, and combinations of both to compute effective memory addresses. At a high level, memory can be addressed using either absolute or relative methods, with more sophisticated forms allowing flexible arithmetic operations on addresses.

Displacement Addressing: This is the simplest form of addressing, where a fixed value (displacement) is treated as an absolute memory address. This mode is often used for accessing global variables or static data determined at compile time. Due to x86-64's 64-bit addressing, displacements are typically limited to 32 bits, restricting their use for addressing large memory regions directly. A notable exception occurs when using specific instructions that allow 64-bit displacements.

```
1 MOV RAX, [0x1000] ; Load value from memory address 0x1000 into rax
```

Figure 2.4: Displacement addressing example

Base Addressing: Base addressing introduces a general-purpose register (GPR) to store the address. Here, the displacement is replaced with a base register, enabling indirect access to memory. This mode is commonly used when a computed address is already stored in a register.

```
1 MOV RBX, [RAX] ; Load value from address stored in RAX into RBX
```

Figure 2.5: Base addressing example

Base + Index Addressing: In this mode, the effective address is computed by adding the values of both the base and index registers. This allows more flexibility, particularly for array accesses where the base represents the array's starting address, and the index corresponds to an offset.

```
1 MOV RAX, [RBX + RCX] ; Load value from address (RBX + RCX) into RAX
```

Figure 2.6: Base + Index addressing example

Base + Displacement Addressing: In this mode, the effective address is calculated by adding the contents of a base register to a displacement value. This is often used when accessing fields within a structure, where the base holds the address of the structure and the displacement represents the offset of a field.

```
1 MOV RAX, [RBX + 0x10] ; Load value from address (RBX + 0x10) into RAX
```

Figure 2.7: Base + Displacement addressing example

Base + Index + Displacement Addressing: This is an extension of the previous mode, where both the base and index registers are used alongside a displacement value. It allows for more complex data access patterns, such as accessing fields within arrays of structures.

```
1 MOV RAX, [RBX + RCX + 0x10] ; Load value from (RBX + RCX + 0x10) into RAX
```

Figure 2.8: Base + Index + Displacement addressing example

Base + (Index * Scale) Addressing: This mode introduces a scaling factor to the index register, allowing it to multiply the index before adding it to the base. The scale is restricted to values of 1, 2, 4, or 8, corresponding to common data sizes such as bytes, words, or pointers.

```
1 MOV RAX, [RBX + RCX * 4] ; Load value from (RBX + 4 * RCX) into RAX
```

Figure 2.9: Base + (Index * Scale) addressing example

Base + (Index * Scale) + Displacement Addressing: This combines the base register, a scaled index register, and a displacement. It is used to compute addresses in complex data structures, such as accessing a field within an array of structures.

```
1 MOV RAX, [RBX + RCX * 4 + 0x20] ; Load value from (RBX + RCX * 4 + 0x20) into RAX
```

Figure 2.10: Base + (Index * Scale) + Displacement addressing example

RIP-relative Addressing: A unique feature of x86-64 is RIP-relative addressing, where the effective address is computed as a displacement from the current instruction pointer (RIP). This is widely used in position-independent code (PIC).

```
1 MOV RAX, [RIP + 0x10] ; Load value from (RIP + 0x10) into RAX
```

Figure 2.11: RIP-relative addressing example

Important Note: Notice in the above examples that we did not specify the size of the memory operand pointed to by the memory address. It is implicitly deduced by the size of the destination operand. For example, in Fig. 2.12, since the destination operand is a 64-bit general purpose register (`RAX`), the instruction will transfer 64 bits of data starting from the memory address specified within the brackets.

We note that there are cases where the memory operand size cannot be determined automatically, so from now on, our code will also explicitly include the referenced memory operand's size. The sizes are named as shown in Table 2.1:

Operand Size	Bit Size
BYTE	8 bits
WORD	16 bits
DWORD	32 bits
QWORD	64 bits
TBYTE	80 bits

Table 2.1: Memory Operand Sizes in x86 Assembly

so the code in Fig. 2.12 would become:

```
1 MOV RAX, QWORD PTR [RIP + 0x10] ; Load 64-bits from (RIP + 0x10) into RAX
```

Figure 2.12: RIP-relative addressing example with pointer size

2.1.3.4 x86 instructions

The x86 instruction set provides a variety of fundamental operations for data movement, arithmetic, logical operations, and control flow. These instructions form the core of programming on x86-based systems. Below is a table with some of the basic x86 instructions and their descriptions. We note that there are several variations for each mnemonic, with different operand types, that are encoded in distinct ways in machine code. We shall explain some of the operand type variations in later sections. For a full reference, one can consult the official Intel manuals for the x86-64 architecture [2].

Instruction	Description	Additional Information
MOV	Move data	Supports registers, memory, and immediates
ADD	Add two operands	Typically used with registers or memory
SUB	Subtract one operand from another	Can use both signed and unsigned integers
MUL	Multiply unsigned integers	Operates on registers, result in EDX:EAX
IMUL	Multiply signed integers	Result in EDX:EAX for 32-bit, sign-extended
DIV	Divide unsigned integers	Dividend in EDX:EAX, quotient EAX, remainder EDX
IDIV	Divide signed integers	Same as DIV, but for signed operands
AND	Perform bitwise AND operation	Used for masking bits
OR	Perform bitwise OR operation	Combines bits from operands
XOR	Perform bitwise XOR operation	Used for toggling bits or clearing registers
NOT	Perform bitwise NOT (complement)	Inverts each bit in the operand
SHL	Shift bits to the left	Equivalent to multiplication by powers of 2
SHR	Shift bits to the right (logical)	Logical shift (fills with zeros)
SAR	Shift bits to the right (arithmetic)	Preserves the sign bit (for signed numbers)
INC	Increment by 1	Affects flags except carry
DEC	Decrement by 1	Affects flags except carry
CMP	Compare two operands	A subtraction without storing the result, sets flags
JMP	Unconditional jump to instruction	Direct control flow change, no flags affected
JE	Jump if equal (zero flag is set)	Used after a comparison or arithmetic operation
JNE	Jump if not equal (zero flag is not set)	Executes if zero flag is cleared
JG	Jump if greater (signed comparison)	Checks signed comparison result
JL	Jump if less (signed comparison)	Checks signed comparison result
PUSH	Push a value onto the stack	Decreases stack pointer, pushes value
POP	Pop a value from the stack	Increases stack pointer, retrieves value
CALL	Call a subroutine	Pushes return address onto stack and jumps
RET	Return from a subroutine	Pops return address from stack and jumps to it
NOP	No operation (do nothing)	Typically used for timing or alignment

Table 2.2: Table of Basic x86 Instructions

2.1.3.5 Example Programs

This subsection presents a couple of example programs to showcase the features we have covered in the previous subsections regarding registers and addressing modes. These are important to later understand the operation of the x86 program generator. The first program simply calculates the sum of integers from 1 to N with a loop.

sum_1_N.asm

```
    ; Assume RDI points to N
    ; Assume RSI points to sum

    ; Initialize N to 10 (example value)
    MOV QWORD PTR [RDI], 10
    ; Initialize sum to 0
    MOV QWORD PTR [RSI], 0
    ; Initialize work registers
    MOV RAX, QWORD PTR [RDI]   ; RAX = N
    MOV RCX, 1                 ; RCX = loop counter (i = 1)

LOOP_START:
    CMP RCX, RAX    ; Compare i with N
    JG LOOP_END     ; If i > N, exit loop
    ADD QWORD PTR [RSI], RCX   ; *sum += i
    INC RCX                    ; i++
    JMP LOOP_START             ; Repeat loop

LOOP_END:
    ; sum pointed to by RSI should now be 1 + 2 + ... + N
```

Figure 2.13: Sum Calculation from 1 to N

Let us now look at the same sum calculation using the closed form equation:

sum_1_N_closed.asm

```
    ; Assume RDI points to N
    ; Assume RSI points to sum
    ; Initialize N to 10 (example value)
    MOV QWORD [RDI], 10
    MOV RAX, [RDI]       ; RAX = N

    ; Compute N * (N + 1) / 2
    MOV RBX, RAX         ; RBX = N
    ADD RBX, 1           ; RBX = N + 1
    IMUL RAX, RBX        ; RAX = N * (N + 1)
    SHR RAX, 1           ; RAX = (N * (N + 1)) / 2 (RAX >> 1 = RAX / 2)

    ; Store the result in sum
    MOV [RSI], RAX
```

Figure 2.14: Closed Form Sum Calculation from 1 to N

We will now present an example program that numerically computes on array elements.

arrays.asm

```
    ; Assume RDI points to array A
    ; Assume RSI points to array B
    ; Assume RDX points to array C
    ; Assume RCX points to size

    ; Initialize size to 10 (example size)
    MOV QWORD PTR [RCX], 10

    ; Set index to 0
    XOR R8, R8          ; R8 = index (i = 0)

COMPUTE_LOOP:
    MOV RAX, QWORD PTR [RCX]      ; RAX = size
    CMP R8, RAX         ; Compare index with size
    JGE LOOP_END        ; If index >= size, exit loop

    ; Load A[i] and B[i]
    MOV RAX, [RDI + R8 * 8] ; RAX = A[i] (R8 * 8 = index * 8 = byte offset)
    MOV RBX, [RSI + R8 * 8] ; RBX = B[i]

    ; Compute A[i] * A[i]
    IMUL RAX, RAX       ; RAX = A[i] * A[i]

    ; Add B[i] to the result
    ADD RAX, RBX        ; RAX = A[i] * A[i] + B[i]

    ; Store result in C[i]
    MOV QWORD PTR [RDX + R8 * 8], RAX

    ; Increment index
    INC R8
    JMP COMPUTE_LOOP    ; Repeat loop

LOOP_END:
```

Figure 2.15: Array Calculation ($C[i] = A[i] * A[i] + B[i]$)

2.2 YAML format

YAML (YAML Ain't Markup Language) is a human-readable data serialization format commonly used for configuration files. It is designed to be simple and easy to understand. YAML uses indentation to represent structure, similar to Python, and avoids the use of braces or brackets.

The primary motivation behind YAML was to provide a format that is easy to read and write, while being simple enough for data serialization. Unlike XML or JSON, YAML aims to be more user-friendly, particularly for configuration files and data exchange between languages with different data structures.

Below, we outline a few key points about YAML and give relevant examples:

2.2.1 Key-Value Pair Structure

YAML stores data as key-value pairs, where the key is followed by a colon and a space. Values can be strings, numbers, booleans, or more complex data types. In the following example we have three keys with string, number, and boolean values accordingly.

```yaml
name: "John Doe"
age: 30
is_student: false
```

Figure 2.16: YAML Key-Value Example

2.2.2 Lists

YAML uses a dash (-) to represent items in a list.

```yaml
fruits:
  - apple
  - banana
  - orange
```

Figure 2.17: YAML List Example

2.2.3 Nested Structure

YAML allows for nesting by indentation. Nested data can be represented using spaces (commonly 2 spaces per level).

```yaml
address:
  street: 123 Main St
  city: Anytown
  country: USA
```

Figure 2.18: YAML Nested Structure Example

2.2.4 Comments

Comments in YAML are denoted by a # symbol.

```yaml
# This is a comment
name: Jane Doe
```

Figure 2.19: YAML Comment Example

2.2.5 Booleans and Null

YAML supports booleans (true, false) and null values (null or $\sim$).

```yaml
is_active: true
has_membership: false
expiration: null
```

Figure 2.20: YAML Boolean and Null Example

Using all the concepts of YAML we presented above, we will now give a more complete example of a YAML file which contains the definition and attribute values for a person.

2.2.6 Complete Example

```yaml
# Definition for a person
person:
  name: Alice
  age: 25
  hobbies:
    - reading
    - swimming
  address:
    street: 456 Elm St
    city: Metropolis
```

Figure 2.21: YAML Complete Example

2.2.7 YAML schema enforcements

YAML's flexibility can sometimes pose dangers. Without enforcing a schema, it's easy for errors such as missing fields, incorrect types, or invalid values to go unnoticed. YAML schema validation ensures that the data follows a predefined structure, reducing errors and increasing reliability in systems that depend on correct configurations.

YAML validation is crucial for the following reasons:

- ***Avoiding Errors***: Validation helps catch mistakes like missing required fields or using the wrong data types before they cause problems in the system.
- ***Enforcing Structure***: A schema ensures that configuration files follow the same structure, even when multiple people work on them.
- ***Early Detection***: Validation allows you to detect issues during development rather than at runtime, preventing system failures caused by invalid configurations.

The YAML schemas can be specified in YAML themselves. A tool then checks the YAML files against the schema and verifies compliance. Below, we present a simple example of a schema.

```yaml
# Schema for server configuration
type: //arr
contents:
  type: //rec
  required:
    name:
      type: //str
      length: { min: 1 }
    ip_address:
      type: //str
  optional:
    port:
      type: //int
      range: { min: 1, max: 65535 }
```

Figure 2.22: YAML Schema Example

This schema ensures that:

1. Each server entry must have a non-empty `name` (string).
2. Each server entry must have an `ip_address`.
3. The `port` field is optional, but if present, it must be an integer between 1 and 65535.

```yaml
- name: server1
  ip_address: 192.168.1.10
  port: 8080
- name: server2
  ip_address: 192.168.1.11
- name: server3
  ip_address: 192.168.1.12
  port: 443
```

Figure 2.23: YAML File Adhering to Schema Example

The file in Fig, 2.23 is valid for the previous schema because all required fields are present, and the optional port field has values within the acceptable range when used. By validating the YAML file against this schema, it is ensured that the configuration adheres to the expected structure, avoiding issues caused by missing or incorrect data.

2.2.8 Summary

In summary, YAML's design focuses on human readability and ease of use for data serialization and configuration. Its simplicity and flexibility have made it a popular choice for various applications in software development, configuration management, and data interchange. Moreover, YAML file structure can be verified using schemas, which is an important consideration when it is crucial to ensure minimization of errors due to missing or incorrect data.

2.3 Related Work

Automatic program generation has been employed in the software as well as in the hardware space and provides solutions to various challenges.

In hardware, automatic code generators are an integral part of functional testing of a microprocessor's design [6, 3, 7]. They are also leveraged to construct power viruses which can aid in the systematic energy characterization of systems, as [4]. Moreover, they are used for constructing stressmarks for early reliability characterization [22].

Software application include machine learning techniques for effective program generator construction [16], or using a program generator to fuzz compilers, assemblers or any tool that takes programs as input [24].

3. MICROPROBE

In this chapter we will dive into the inner workings of MicroProbe [1, 4]. We begin by giving a high-level overview of the entire system, continue by giving a more thorough description of each module in the system and we will conclude the chapter with a concrete code generation example.

3.1 Overview

MicroProbe [1, 4] is a modular, extensible, and almost fully ISA-independent framework designed for code generation. Originally developed to facilitate precise energy characterization of CPUs [4], MicroProbe's key strength lies in its separation of architectural (ISA) and microarchitectural details from the code generation process.

The framework is structured into two primary modules: the *Architecture Module* and the *Code Generation Module*, as depicted in Fig. 3.1.

- The **Architecture Module** comprises configuration text files that define all the details related to the ISA (e.g., registers, instructions, formats) and, optionally, certain microarchitectural characteristics (e.g., core components, cache configurations, latencies). During code generation, this information—particularly the ISA specifics—is accessed to ensure that the resulting microbenchmark is valid for the target ISA.
- The **Code Generation Module**, on the other hand, orchestrates the entire code generation process. The microbenchmark being produced is initially encoded in an internal representation, which is then refined through a series of compiler-like transformations, known as *passes*, specified by the user. These transformations manage tasks such as initialization, branch resolution, memory operand resolution, and register allocation. The specific sequence of passes used to generate the final microbenchmark is collectively referred to as a *policy*.

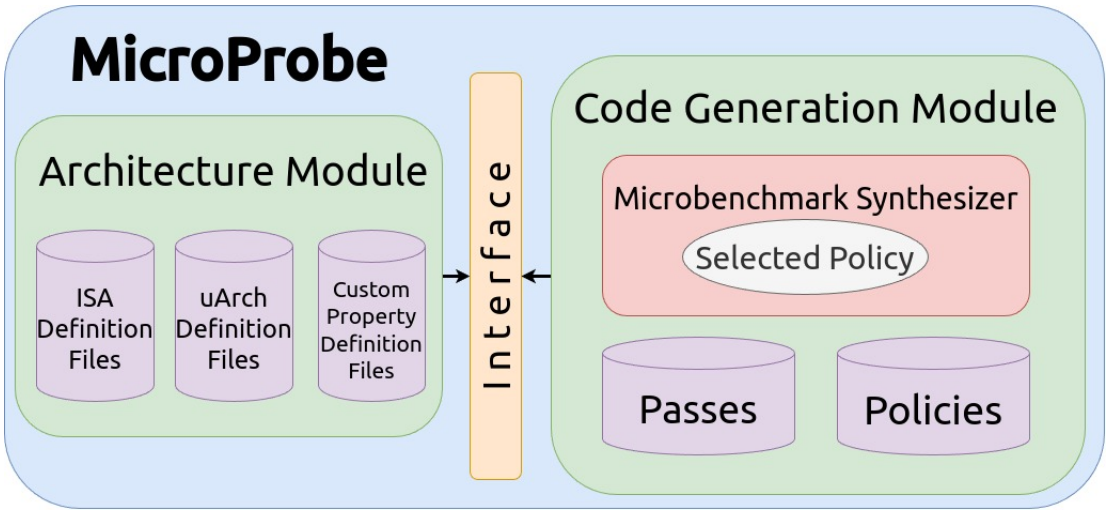

Figure 3.1: The MicroProbe framework

Thanks to the architectural separation, the collection of *passes* and the resulting *policies* are both generic and reusable. For instance, existing register allocation passes within the

code generation module can be applied to any target ISA defined within the architecture module. The code generation process is directed by the *synthesizer* object, to which the specified sequence of passes (or policy) is attached. Ultimately, the microbenchmark must be lowered to a valid assembly program, a process that requires detailed architectural information. This information is retrieved through interfaced communication between the **Code Generation Module** and the **Architecture Module**.

MicroProbe's approach is particularly well-suited for program generation in many tasks due to several key features:

1. **Extensibility**: The framework's inherent modularity and extensibility allow for seamless integration with other custom modules.
2. **ISA-Awareness**: The generated programs are guaranteed to be valid, provided that all ISA constraints are correctly encoded.
3. **Assembly-Level Code Generation**: Working at the assembly level is crucial because it grants complete control over code generation. Using a higher-level programming language would delegate instruction selection to the compiler's backend, which might prioritize certain instructions over others and translate high-level constructs into specific patterns. While a compiler typically optimizes for performance or code-size, such behavior would limit our flexibility.

3.2 Architecture Module

Focusing on the architecture module, we will now go over some concrete examples of the different kinds of text files containing definitions or properties related to the ISA and the microarchitecture (uarch). The examples are taken from the x86 definition files, the creation of which was part of the extension process, as we will describe in section 4. For now, we simply showcase a few important samples from these files and comment on their structure. The files are written in the YAML format. Readers unfamiliar with the YAML format are encouraged to consult section 2.2.

3.2.1 ISA Definition Files

This set of definition files is **required for every target** (e.g., x86, RISC-V, Power) supported by MicroProbe. They comprise the framework's "knowledge" regarding each target's ISA. Without this information, valid assembly generation for a target is an impossible task.

On the contrary, uarch definition files and property definition files are optional and are only used in more specific generation tasks. For example, one might like to include in the generation process only instructions whose latency is less than 5 cycles. This is a microarchitectural constraint and this information - i.e., the latency of each instruction - must be contained in uarch definition files, which we will talk about in the next subsection.

Let us now present samples from the ISA definition files for x86. The presentation moves in a bottom-up approach, starting from basic files that do not reference any structure from the rest of the files, and building up to complex files (like *instruction.yaml*) which define high-level constructs referencing basic structures defined in multiple other files.

The first ISA definition file example (`register_type.yaml`) is shown in Fig. 3.2. Note that the figure only contains a portion of the full-text file. We adopt the approach of showing representative portions of the files to save significant space in the text, as some files are hundreds of lines long.

The contents of this file should come as no surprise, as we have already discussed x86 registers and their types in 2.1.3.1.

register_type.yaml

```yaml
- Name: GR64
  Size: 64
  Description: General Register 64 bits
  AddressArithmetic: True

- Name: VR64
  Size: 64
  Description: Vector Register 64 bits

- Name: IP64
  Size: 64
  Description: Instruction Pointer Register 64 bits

- Name: CR64
  Size: 64
  Description: Control Register 64 bits
...
```

Figure 3.2: register_type.yaml sample for x86

It is very important to note that all the ISA definition files are constrained by corresponding schema files. Schema enforcement in YAML is discussed in 2.2.7. Fig. 3.3 presents the schema file for Fig. 3.2.

register_type.yaml

```yaml
type: //arr
contents:
  type: //rec
  required:
    Name:
      type : //str
      length : { min: 1 }
    Size:
      type : //int
      range : { min: 1 }
  optional:
    Description:
      type : //str
      length : { min: 1 }
    AddressArithmetic:
      type : //bool
    FloatArithmetic:
      type : //bool
    VectorArithmetic:
      type : //bool
```

Figure 3.3: register_type.yaml schema definition

Moving on with registers, Fig. 3.4 gives a subset of the concrete x86 register definitions

based on the register types we have just defined. Again, the reader is prompted to consult 2.1.3.1 if unfamiliar with x86 registers.

register.yaml

```yaml

- Name: RAX
  Type: GR64
  Representation: 'RAX'
  Description: General Register RAX

- Name: EAX
  Type: GR32
  Representation: 'EAX'
  Description: DWord General Register EAX

- Name: AX
  Type: GR16
  Representation: 'AX'
  Description: Word General Register AX

- Name: AL
  Type: GR8
  Representation: 'AL'
  Description: General Register Low Byte AL

- Name: AH
  Type: GR8
  Representation: 'AH'
  Description: General Register High Byte AH

...

- Name: R8
  Type: REX_GR64
  Representation: 'R8'
  Description: General Register 8
  Repeat:
    From: 8
    To: 15

...

```

Figure 3.4: register.yaml sample for x86

operand.yaml

```yaml
...

- Name: R_64
  Description: Quad word registers
  Registers:
    RAX : [ AL, AH, AX, EAX, RAX ]
    RBX : [ BL, BH, BX, EBX, RBX ]
    RCX : [ CL, CH, CX, ECX, RCX ]
    RDX : [ DL, DH, DX, EDX, RDX ]
    RBP : [ BPL, BP, EBP, RBP ]
    RSI : [ SIL, SI, ESI, RSI ]
    RDI : [ DIL, DI, EDI, RDI ]
    RSP : [ SPL, SP, ESP, RSP ]

...

- Name: XMM
  Description: 128-bit vector registers
  Registers:
    XMM0 : [ ZMM0, YMM0, XMM0 ]
    XMM1 : [ ZMM1, YMM1, XMM1 ]
    XMM2 : [ ZMM2, YMM2, XMM2 ]
    XMM3 : [ ZMM3, YMM3, XMM3 ]
    XMM4 : [ ZMM4, YMM4, XMM4 ]
    XMM5 : [ ZMM5, YMM5, XMM5 ]
    XMM6 : [ ZMM6, YMM6, XMM6 ]
    XMM7 : [ ZMM7, YMM7, XMM7 ]

...

- Name: SImm32
  Description: Signed immediate (bit size 32)
  Min: -2147483648
  Max: 2147483647

- Name: UImm32
  Description: Unsigned immediate (bit size 32)
  Min: 0
  Max: 4294967295

- Name: '@SImm32'
  Description: Relative signed displacement (bit size 16)
  Relative: True
  MinDisplacement: -2147483648
  MaxDisplacement: 2147483647

...
```

Figure 3.5: operand.yaml sample for x86

The next file is `operand.yaml`, shown in Fig. 3.5. As the name suggests, we define the possible operands of an x86-instruction. The first operand shown, `R_64`, defines all 64-bit registers (without a REX prefix - which is why `R8-R15` are missing), as well as the register hierarchy (noted with bracketed lists) that we discussed in 2.1.3.1. If the code generator uses `RAX` for example, it knows that the values of all subregisters might be affected.

Another important type of operand, apart from registers, are immediate operands. Fig. 3.5 shows three definitions:

- `SImm32`, a signed 32-bit immediate (with the minimum and maximum values shown) which can be an operand of instructions that use immediates of such width.
- `UImm32`, an unsigned 32-bit immediate.
- `@SImm32`, a signed, relative displacement immediate which is used as an addressing mode operand. Addressing modes were discussed in 2.1.3.3.

Samples from the next file, namely `instruction_field.yaml` are shown in two figures, Fig. 3.6 and Fig. 3.7.

The first figure's fields are x86-specific (apart from the opcode field, which is present in virtually all ISAs). The `REX-header` is part of the `REX-byte` which is a prefix used in x86-64 instructions to extend the capabilities of certain operations (e.g., using more registers or 64-bit operands). The `ModRM` byte contains the fields shown. In summary, the `ModRM` byte is used to encode operands for many instructions in x86. Its main parts are:

1. `Mod`: Defines the addressing mode (register or memory and optionally displacement).
2. `Reg/Opcode`: Defines a register operand or an opcode extension.
3. `RM`: Defines the second register or memory operand.

instruction_field.yaml (sample 1)

```yaml

- Name: "REX_header"
  Description: First 4 bit of REX byte with fixed value
  Size: 4
  Show: False
  IO: I
  Operand: UImm4_0100

- Name: "opcode_8"
  Description: Opcode size 8
  Size: 8
  Show: False
  IO: '?'
  Operand: UImm8

- Name: "ModRM_Mod"
  Description: Mod bits of ModRM byte
  Size: 2
  Show: False
  IO: '?'
  Operand: UImm2

- Name: "ModRM_RegOpc"
  Description: Register field in ModRM byte
  Size: 3
  Show: True
  IO: 'I'
  Operand: UImm3

- Name: "ModRM_RegOpc_Opext"
  Description: Opcode extension field in ModRM byte
  Size: 3
  Show: False
  IO: '?'
  Operand: UImm3

- Name: "ModRM_RM"
  Description: Register/memory field in ModRM byte
  Size: 3
  Show: True
  IO: 'I'
  Operand: UImm3

...

```

Figure 3.6: instruction_field.yaml sample 1 for x86

The second sample in Fig. 3.7, contains more straightforward instruction fields, like displacements (used in addressing) and immediates.

instruction_field.yaml (sample 2)

```yaml

- Name: "SD1_32"
  Description: Signed displacement of instruction operand 1
  Size: 32
  Show: True
  IO: I
  Operand: SImm32

- Name: "D2_32"
  Description: Displacement of instruction operand 2
  Size: 32
  Show: True
  IO: I
  Operand: UImm32

- Name: "I1_32"
  Description: 32 bit immediate operand 1 signed
  Size: 32
  Show: True
  IO: I
  Operand: SImm32

- Name: "I1_U32"
  Description: 32 bit immediate operand 1 unsigned
  Size: 32
  Show: True
  IO: I
  Operand: UImm32

- Name: "AI_S32"
  Description: Address Immediate (32-bit)
  Size: 32
  Show: True
  IO: I
  Operand: AImm32

...

```

Figure 3.7: instruction_field.yaml sample 2 for x86

The next figure (Fig. 3.8) focuses on instruction formats. It is important to note that our handling of instruction formats in x86 departs from a 1:1 correspondence with the formats defined in x86 manuals so our formats are somewhat customized to better suit the generation framework's needs. The main reason for this is the following: In MicroProbe, instruction formats are tied to the assembly that is generated for an instruction. However, a single instruction can have more than 10 variants in some cases (according to the operands and specified operand types). Groups of these variants are often encoded with the same instruction format, which poses a problem: the assembly replacement should be different for the variants but the format is shared. To alleviate this problem, we opt

to use a different, distinct, instruction format for each combination of operands and operand types in an instruction. That means that if ADD has 10 variants we will separately specify 10 different instructions ADD_V0, ADD_V1, etc., with all these instructions having a separate instruction format according to their operands. In this way, even though we depart from x86 formats, we restore the 1:1 correspondence between instruction formats and assembly replacements required by the MicroProbe framework.

Taking a look at the format definitions in Fig. 3.8, we can see that each format contains a number of fields and the assembly encoding, which positions the fields in their proper places. In the assembly encoding, whatever is not a field, is treated as a literal (e.g., brackets). Fields will be eventually replaced by proper operand names during code generation.

The first definition for `O8MRR` generates the assembly: `OPC ModRM_RM, ModRM_RegOpc`. Let's explain:

- `O8` means that there are 8 bits of opcode
- `MR` means the first operand is a memory/register operand and the second is a register
- The final `R` explains that the second operand is, in fact, a register

To summarize, this is an instruction with two register operands. There is a variant of `ADD` fitting this format. For example, we could write: `ADD RAX, RBX` which does the operation: `RAX += RBX`. Now contrast the above format with the next one `O8MRM8`. There is one essential difference: The second operand is now memory (with 8 bits length). This is why we get the assembly: `OPC BYTE PTR [ModRM_RM + AI_S32], ModRM_RegOpc`. Note the `BYTE PTR` specifier.

Another point to be drawn out is that concerning the addressing modes we describe in 2.1.3.3, we opt to implement `Base + Offset` addressing (notice the immediate offsets inside the brackets when the instruction format contains memory operands), as well as `RIP-relative` addressing which is adequate for most code generation tasks. The framework would easily allow us to implement the other addressing modes as well.

instruction_format.yaml

```yaml
- Name: "O8MRR"
  Fields:
  - opcode_8
  - ModRM_Mod
  - ModRM_RegOpc
  - ModRM_RM
  Assembly: OPC ModRM_RM, ModRM_RegOpc

- Name: "O8MRM8"
  Fields:
  - opcode_8
  - ModRM_Mod
  - ModRM_RegOpc
  - ModRM_RM
  - AI_S32
  Assembly: OPC BYTE PTR [ModRM_RM + AI_S32], ModRM_RegOpc

- Name: "O8MRM16"
  Fields:
  - opcode_8
  - ModRM_Mod
  - ModRM_RegOpc
  - ModRM_RM
  - AI_S32
  Assembly: OPC WORD PTR [ModRM_RM + AI_S32], ModRM_RegOpc

- Name: "O8MRM32"
  Fields:
  - opcode_8
  - ModRM_Mod
  - ModRM_RegOpc
  - ModRM_RM
  - AI_S32
  Assembly: OPC DWORD PTR [ModRM_RM + AI_S32], ModRM_RegOpc

- Name: "O8RMR"
  Fields:
  - opcode_8
  - ModRM_Mod
  - ModRM_RegOpc
  - ModRM_RM
  Assembly: OPC ModRM_RegOpc, ModRM_RM

...
```

Figure 3.8: instruction_format.yaml sample for x86

Putting all the previous definitions to use, we can finally define an instruction. For example, the first definition in Fig. 3.9 specifies a variant of the `ADD` instruction, namely `ADD_V0`. The

Name, Mnemonic, Opcode and Description attributes are trivial to interpret. Moving on to the Format attribute we notice the `O8MRR` format we discussed in the previous paragraphs. To summarize, we have an 8-bit opcode (verified by the two hex digits - 4 bits each - in the Opcode attribute), and two register operands.

Ignoring the first entry in the Operands attribute (which poses no interest, so we do not bother to expand upon), the second entry encodes the fact that the first register operand is of operand type `R_8` i.e. an 8-bit register. Further, the `'IO'` specifier means that this operand is both a source (input) and a destination (output) operand, which is common in x86. Similarly, the third line encodes the fact that the second register operand is only a source operand (`'I'`) and is an 8-bit register.

Given this information one possible concrete assembly resolution could be: `ADD AH, BH` where `AH, BH` are 8-bit registers (as discussed in 2.1.3.1) and the operation encoded is `AH += BH`.

Finally, the InstructionChecks attribute allows us to encode special checks (essentially Python functions) which trigger during code generation when any operand of that instruction is modified.

instruction.yaml

```yaml
- Name: "ADD_V0"
  Mnemonic: "ADD"
  Opcode: "00"
  Description: "Add"
  Format: "O8MRR"
  Operands:
    ModRM_Mod : [ '3', 'ModRM_Mod', '?' ]
    ModRM_RM : [ 'R_8', 'ModRM_RM', 'IO']
    ModRM_RegOpc : [ 'R_8', 'ModRM_RegOpc', 'I']
  InstructionChecks:
    C0: [check_rex_registers]

- Name: "ADD_V1"
  Mnemonic: "ADD"
  Opcode: "00"
  Description: "Add"
  Format: "O8MRM8"
  Operands:
    ModRM_Mod : [ '3', 'ModRM_Mod', '?' ]
    ModRM_RM : [ 'ABR_64', 'ModRM_RM', 'IO']
    ModRM_RegOpc : [ 'R_8', 'ModRM_RegOpc', 'I']
    AI_S32: ['AImm32', 'AI_S32', 'I']
  MemoryOperands:
    MEM1 : [ ['ModRM_RM', 'AI_S32'], [1], 1, 'IO']
  InstructionChecks:
    C0: [check_rex_registers]
...
```

Figure 3.9: instruction.yaml sample for x86

3.2.2 uarch Definition Files

Microarchitecture (uarch) Definition Files are not mandated by MicroProbe for any ISA. The information in them is used in a complementary fashion so that we are able to encode or formulate additional constraints or requirements for the code generation process. For example, perhaps we would like to resolve all data accesses in a memory area that is sized exactly equal to the L2 data cache. The L2 data cache size is microarchitectural information, so MicroProbe would try to query objects related to the uarch Definition Files for caches to retrieve this information.

We will not present comprehensive examples for uarch Definition Files, as they are not required for basic code generation tasks. Fig. 3.10 shows a sample `element.yaml` file, which simply defines some microarchitectural elements of the processor in a hierarchical manner.

element.yaml

```
- Name: Processor
  Type: Processor
  Subelements:
    - Core0
    - L3
    - MEM
- Name: Core0
  Type: Core
  Subelements:
    - L1D
    - L1I
    - L2
- Name: L3
  Type: L3
- Name: MEM
  Type: MEM
- Name: L1D
  Type: L1D
- Name: L1I
  Type: L1I
- Name: L2
  Type: L2
```

Figure 3.10: element.yaml sample

3.2.3 Property Definition Files

Similarly to uarch definition files, property definition files are not mandatory. However, some simple property files like `branch.yaml` shown in Fig. 3.11 or `memory.yaml` essentially encode large instruction classes, all branch instructions and all instructions that address memory with at least one operand correspondingly.

One can define arbitrary many property files with custom properties which could enable different categorizations of instructions, or even encode some other property of the ISA.

branch.yaml

```yaml
- Name: branch
  Description: Boolean indicating if the instruction is a branch
  Default: False
  Values:
    JO_V0: True
    JNO_V0: True
    JB_V0: True
    JNAE_V0: True

...
```

Figure 3.11: branch.yaml sample

3.3 Code Generation Module

This module is essentially the core of the framework. It instantiates the program under construction, encoding it in an internal representation that is ISA-agnostic. This representation is then repetitively transformed and refined in steps that resemble compiler passes.

These MicroProbe ***passes***, as we shall see, are essentially Python functions that implement discrete transformation-refinement tasks (e.g., register allocation, memory operand resolution). Many of these tasks necessitate the use of ISA-specific information; to perform register allocation, one has to know which registers and register types the ISA supports. The ***crux idea*** in MicroProbe is that the ***passes*** need not be ISA-specific. Register allocation can be implemented as an ISA-agnostic procedure that gathers ISA-details from another independent module (the Architecture Module we presented in the previous section) to complete the register allocation process.

The benefit of this approach is that the register allocation strategy is written ***once***, and can perform correct register allocation with any ***ISA*** that is supported by the Architecture Module. The interfaced communication and separation between the architecture and code generation modules avoids a significant amount of code repetition. It would be cumbersome to redefine the same register allocation strategy (e.g., round-robin) for every new ISA that we include in the framework.

3.3.1 Passes

This section gives an outline of the passes that the code generation module supports and some concrete examples of some simple passes. The user of MicroProbe can also implement any number of custom passes, if the implemented passes do not fit their generation needs. These passes are seamlessly integrated into the framework and can be used for the code generation task the user intended.

The following list presents the core passes defined in the MicroProbe framework, also grouping them by category:

- **Structure**
  - ***SimpleBuildingBlockPass***: Adds a building block (analogous to a basic block, but it can contain branches) of N instructions to the Control Flow Graph (CFG). Does not resolve to specific instructions, only allocates the “space” for them to be filled in.

- **Initialization**
  - ***AddInitializationAssemblyPass***: Append given instructions at the initialization of the building block, i.e., right before the first instruction of the building block.
  - ***AddFinalizationAssemblyPass***: Append given instructions at the end of the building block, i.e., right after the last instruction of the building block.
  - ***ReserveRegistersPass***: Can reserve a set of registers so that the other passes (e.g., register allocation) consider them used and ignore them. For example, a register used as a loop counter should be reserved; otherwise register allocation might reuse it in unrelated instructions and corrupt its value.
  - ***UnReserveRegistersPass***: Make a set of reserved registers available.
- **Branches**
  - ***BranchNextPass***: Set the branch target of every branch instruction to be the next instruction. (i.e., taken and not-taken behave identically)
  - ***LinkBblsPass***: Add unconditional branch between consecutive instructions without consecutive addresses if the first instruction is not a branch.
- **Register Allocation**
  - ***DefaultRegisterAllocationPass***: Round-robin register allocation. It can also achieve a specific dependency distance. For example, if the distance requested is 2 and `RAX` is used, its next use will be exactly after 2 instructions (if that is possible).
  - ***NoHazardsAllocationPass***: Avoid all data hazards:
    * read after write (RAW), a true dependency
    * write after read (WAR), an anti-dependency
    * write after write (WAW), an output dependency
  - ***RandomAllocationPass***: Randomly allocates registers for all instructions with register operands.
  - ***FixRegistersPass***: Forbid writes or reads to a register.
- **Memory**
  - ***SingleMemoryStreamPass***: Resolve every memory access to a set of N memory locations (round-robin fashion) separated by a stride of B bytes.
  - ***GenericMemoryModelPass*** - Complex pass, requires a model object to be provided, and is usually combined with uarch information. Can achieve things like: 20% accesses to the L2 and the rest are resolved in L1.

Below, we show two simple examples from the aforementioned passes. The rest of the passes' implementations are lengthy and not friendly for a presentation, so the reader is encouraged to study them on their own pace. The repository can be accessed from this link: [1].

The ***SimpleBuildingBlockPass*** in Fig. 3.12, simply adds the building block of the specified size to the Control Flow Graph (CFG). Notice that passes initialize their state in the

constructor (`__init__` method) and perform their function when called as callable objects by modifying the python built-in `__call__` method.

The ***AddInitializationInstructionsPass*** in Fig. 3.13, takes in a list of instructions as well as a list of operands that are applied 1-to-1 to these instructions, and when called, it constructs for each instruction and operands pair an internal MicroProbe instruction object. It proceeds to set its type and its operands and finally adds it to the initialization block of the given building block. The initialization block's instructions are placed right before the first instruction of the building block.

```
class SimpleBuildingBlockPass(passes.Pass):
    """SimpleBuildingBlockPass Class.

    This :class:`~.Pass` adds a single building block of a given instruction
    size to the given building block. The pass fails if the building block
    size differs in ratio more than the threshold provided.
    """

    def __init__(self, bblsize, threshold=0.1):
        """Create a SimpleBuildingBlockPass object.

        :param bblsize: Size of the building block to add
        :type bblsize: :class:`int`
        :param threshold: Allowed deviation from the size provided
                          (Default value = 0.1)
        :type threshold: :class:`float`
        :return: A new SimpleBuildingBlockPass object.
        :rtype: :class:`SimpleBuildingBlockPass`
        """
        super(SimpleBuildingBlockPass, self).__init__()
        self._bblsize = bblsize
        self._description = "Create a basic block with '%d' " \
                            "instructions" % self._bblsize
        self._threshold = threshold

    def __call__(self, building_block, dummy):
        """
        :param building_block:
        :param dummy:
        """

        building_block.cfg.add_bbl(size=self._bblsize)
```

Figure 3.12: SimpleBuildingBlock pass

3.3.2 Wrappers

Wrappers are in essence predefined containers for the core program instructions that are generated, so that the final result of the generation process can be easily utilized. For example, a raw assembly sequence cannot produce a binary. To solve this we could define a simple wrapper; a C program with an empty main method, in which we insert the generated program's instructions (as inline assembly) is compilable and solves the problem.

In fact, wrappers can have any structure or format, and the users of the framework can define any wrapper that suits their needs.

```
1
2  class AddInitializationInstructionsPass(passes.Pass):
3      """AddInitializationInstructionsPass pass.
4      """
5
6      def __init__(self, instr, operands):
7          """
8          :param instr:
9          :param operands:
10         """
11         super(AddInitializationInstructionsPass, self).__init__()
12         self._instr = instr
13         self._operands = operands
14         self._description = "Add %s in the init sequence with " \
15             "operands: %s" % (instr, operands)
16
17     def __call__(self, building_block, dummy_target):
18         """
19         :param building_block:
20         :param dummy_target:
21
22         """
23         for instr, operands in zip(self._instr, self._operands):
24             newinstr = microprobe.codegen.ins.Instruction()
25             newinstr.set_arch_type(instr)
26             newinstr.set_operands(operands)
27             building_block.add_init([newinstr])
28
```

Figure 3.13: AddInitializationInstructionsPass pass

3.3.3 Policies

We have already presented MicroProbe's ***passes***, the reusable, ISA-agnostic transformations that define the generated program's structure and resolve all (memory, branch, register) operands so that, as a last step, the internal representation can emit valid assembly. The full collection of passes that defines and transforms a program under generation to its final state (from which we can emit assembly) is referred to as a ***policy***.

Policies are most often user-specified and bound to the generation task at hand. It is particularly easy to compose a new policy by stitching together properly parameterized passes. MicroProbe offers some generic policies which tackle common scenarios and can be adapted to accommodate a more complex generation scenario's needs.

There is also the notion of the ***synthesizer*** object. Essentially, the policy attaches its passes to the synthesizer, which applies those passes to the internal representation one-by-one in the specified order. The synthesizer is also responsible for finalizing emitting assembly into the predefined wrapper (essentially the program's skeleton).

In Fig. 3.14 we provide an example of a simple policy.

- The `sequence` variable in line 3 contains the list of mnemonics (as strings) that were passed in to the policy. Sequences can be either randomly or systematically generated but that is not the policy's concern.
- The `wrapper_class` variable contains the specific wrapper we have opted to use for the policy. In this case it is a C wrapper for x86 assembly with loop support. Details

```
1  def policy(target, wrapper, **kwargs):
2
3      sequence = kwargs['instructions']
4      context = microprobe.codegen.context.Context()
5
6      # Choose a custom wrapper
7      NUM_LOOPS = kwargs['loops']
8      wrapper_class = get_wrapper("CX86LoopGenResult")
9
10     wrapper = wrapper_class(
11         size=NUM_LOOPS,
12     )
13
14     synthesizer = microprobe.codegen.Synthesizer(
15         target, wrapper, value=0b01010101
16     )
17
18     # Initialize each register to a random 64-bit value
19     synthesizer.add_pass(passes.initialization.InitializeRegistersPass(
20             value=lambda: RNDINT(maxmin=(-2 ** 63, 2 ** 63 - 1)),
21             # give two doubles for XMM (128-bit)
22             v_value=(lambda: (RNDFP(), RNDFP()), 64)
23         )
24     )
25
26     # If we have loops, we implement them with R15 so reserve that reg!
27     if NUM_LOOPS > 0:
28         synthesizer.add_pass(passes.initialization.ReserveRegistersPass(['R15']))
29
30     synthesizer.add_pass(passes.structure.SimpleBuildingBlockPass(kwargs['benchmark_size']))
31     synthesizer.add_pass(passes.instruction.SetInstructionTypeBySequencePass(sequence))
32     synthesizer.add_pass(passes.address.UpdateInstructionAddressesPass())
33     synthesizer.add_pass(passes.branch.BranchNextPass(avoid_loopback=True))
34
35     # Get region size & stride
36     memsz = kwargs['mem_access_size']
37     stride = kwargs['mem_access_stride']
38
39     # How many locations needed to get to that memory size with specific stride
40     num_locations = math.ceil(memsz / stride)
41
42     synthesizer.add_pass(passes.memory.SingleMemoryStreamPass(num_locations, stride))
43
44     synthesizer.add_pass(passes.register.DefaultRegisterAllocationPass(
45         dd=kwargs['dependency_distance']))
46     synthesizer.add_pass(passes.address.UpdateInstructionAddressesPass())
47
48     return synthesizer
```

Figure 3.14: Simple policy example

on wrappers can be found in subsection 3.3.2. The wrapper object is instantiated at `line 10`.

- We then instantiate the synthesizer object to which we will attach our passes in `line 14`.
- The first pass that is attached is the register initialization pass, which is configured to generate random values to both integer and floating-point registers. `lines 19-24`

- Following that, we add a register reservation pass for `R15` which will be used as a loop counter in case we have opted for loops.
- ***SimpleBuildingBlockPass*** adds a single building block to the CFG with the specified size (`line 30`)
- ***SetInstructionTypeBySequencePass*** constructs the internal representation for each instruction that is specified in the sequence (which is simply a list of mnemonics of string type) (`line 31`)
- ***UpdateInstructionAddressesPass*** performs some internal bookkeeping related to the building block and its offsets (`line 32`)
- ***BranchNextPass*** resolves all branch operands to the following instruction's address, thus equating taken and not-taken behaviour (`line 33`)
- Next, `line 40` calculates the number of memory locations we have to construct based on the size and stride requested. `line 42` attaches the memory stream pass which will resolve memory operands to these locations in a round-robin fashion.
- Finally, we attach the ***DefaultRegisterAllocationPass*** which will allocate registers with the specified dependency distance constraints. Another ***UpdateInstruction-AddressesPass*** is attached to perform final bookkeeping on instruction addresses, and the synthesizer object, to which we have now attached all the passes, is returned for use.

3.4 Code Generation Example

An important detail we have not stressed is the fact that policies only specify the passes that are applied to the instructions of the program under generation. The actual instruction sequence of the program is supplied to the policy as an argument.

Consequentially, to perform actual code generation with the framework, we also need a way to specify or generate the input instruction sequences. MicroProbe solves this problem by providing a set of tools that can generate sequences based on user supplied arguments. Moreover, the framework easily allows the integration of custom sequence generation strategies added by the user for a specific task.

For the code generation example that follows, we focus on the ***mp_seq.py*** tool, which is one of the default tools provided by the framework. The full list of available tools and their functions can be found in the documentation [1].

This is the description for ***mp_seq.py*** as presented in the documentation: "The ***mp_seq*** tool provides an interface to generate stressmarks that execute sequences that combine the instructions provided in a certain way to comply the requirements provided. Sequence stressmarks are loops with a particular sequence repeated several times. They are ideal to perform core characterizations during the first steps of the maximum power generation process (i.e., find the sequence that maximizes the power consumption of the cores). Users can control the level of instruction-level parallelism and the loop size via configurable parameters.

There are many features to ***mp_seq.py*** but we will use it in the following way: Given a instruction sequence length (*-is* argument), for example 3, and a single instruction group

(*-ig* argument) - essentially a comma-separated list of instructions - the tool will generate all length-3 permutations (repeats allowed) of the instructions in this group.

Our invocation of the tool will look like this:

```
mp_seq.py -T intel64-x86generic-x86_64_linux_gcc \
          -D /home/nick/PycharmProjects/x86-microprobe/generation_results \
          -policy seq -dd 0.0 \
          -mem-access-size 32768 --mem-access-stride 16 \
          -p --loops 0 \
          -is 3 -ig ADD_V0,OR_V0,AND_V0
```

Figure 3.15: mp_seq.py invocation example

Let's explain each argument:

- The first argument (`-T`) specifies the target tuple. Microprobe follows a GCC-like target definition scheme, where a target is defined by a tuple as follows:
  `<arch-name>-<uarch-name>-<env-name>`
  In our case:
  `<arch-name>:intel64|<uarch-name>:x86generic|<env-name>:x86_64_linux_gcc`

  The `arch-name` and `uarch-name` essentially select the ISA and the microarchitectural information which will be used from the Architecture Module we presented in 3.2.

- The second argument (`-D`) is the directory in which the program generation results will be placed in.

- The third argument (`-policy`) chooses the policy which will be applied during the generation process. In this case the `seq` policy is nearly identical to the policy example we gave in Fig. 3.14.

- The fourth argument (`-dd`) specifies the dependency distance for the register allocation pass. A value of 0 means "no dependencies". Of course that is not always possible since we have a finite amount of architectural registers in any ISA, so in practice a value of 0 achieves the maximum possible dependency distance for a given ISA.

- The fifth and sixth arguments (`--mem-access-size` and `--mem-access-stride`) specify the memory region size and stride, thus the available addresses, for the resolution of memory operands.

- `-p` specifies that we want parallel generation of the programs (utilizing all cores).

- `--loops` with a value of 0 specifies that we do not want the sequence to loop, it will only execute once.

- Finally, as we mentioned earlier, `-is 3` specifies that we want sequences of length 3 and `-ig ADD_V0,OR_V0,AND_V0` specifies that the sequences are constructed via permutations (with repetition allowed) of these instructions.

Since repetition is allowed, and we have three possible instructions for each of the three slots, the total sequences that will be generated are $3 \times 3 \times 3 = 27$.

```
uint8_t INTEL64_SCRATCH_VAR[256]  __attribute__ ((aligned (8)));

int main(int argc, char **argv, char **envp)
{
// Init 64-bit general registers
__asm__(" MOV R8, 7194188941638899757 ");
__asm__(" MOV R9, 6811109015807608225 ");
__asm__(" MOV R10, 4316973530061587657 ");
__asm__(" MOV R11, -8670449709817618415 ");
__asm__(" MOV R12, 9186602996245478945 ");
__asm__(" MOV R13, 4514843480026582523 ");
__asm__(" MOV R14, -4487945648925832719 ");
__asm__(" MOV R15, -4906320255886452125 ");
__asm__(" MOV RAX, 8880660406703186701 ");
__asm__(" MOV RBX, 2557542285177312533 ");
__asm__(" MOV RCX, -7058461242704356945 ");
__asm__(" MOV RDI, 8959236520150022023 ");
__asm__(" MOV RDX, -3812801892791624063 ");
__asm__(" MOV RSI, 418617522375225635 ");

// Init 128-bit XMM registers
__asm__(" PUSH 2144375704 "); __asm__(" PUSH 1515419843 ");
__asm__(" PUSH 2145775010 "); __asm__(" PUSH 732126844 ");
__asm__(" MOVDQA XMM0,  [RSP + 0x0] ");
__asm__(" PUSH 2145772543 "); __asm__(" PUSH -1019920204 ");
__asm__(" PUSH 2146119107 "); __asm__(" PUSH 126835601 ");
__asm__(" MOVDQA XMM1,  [RSP + 0x0] ");
...
__asm__(" PUSH 2146094613 "); __asm__(" PUSH 113022118 ");
__asm__(" PUSH 2145613831 "); __asm__(" PUSH -939657862 ");
__asm__(" MOVDQA XMM7,  [RSP + 0x0] ");

// Core instructions
__asm__(" first:ADD AL, AH ");
__asm__(" AND BL, BH ");
__asm__(" OR DL, DH ");

// Storing register state in memory
__asm__(" mov INTEL64_SCRATCH_VAR[rip + 0], rax ");
__asm__(" mov INTEL64_SCRATCH_VAR[rip + 8], rbx ");
...
__asm__(" mov INTEL64_SCRATCH_VAR[rip + 104], r15 ");
__asm__(" movdqa INTEL64_SCRATCH_VAR[rip + 112], xmm0 ");
...
__asm__(" movdqa INTEL64_SCRATCH_VAR[rip + 224], xmm7 ");
__asm__(" mov INTEL64_SCRATCH_VAR[rip + 240], rbp ");
__asm__(" mov INTEL64_SCRATCH_VAR[rip + 248], rsp ");
...
}
```

Figure 3.16: mp_seq.py result C program example

Below we show an example snippet of one of the generated C programs (remember that

our wrapper generates a C program in which the core instructions are placed):

As seen in `lines 34-36` of Fig. 3.16, this program's core instructions are the permutation `ADD_V0, AND_V0, OR_V0`, which are the variants of the `ADD, AND, OR` instructions that operate on 2 byte registers. The code generation module has allocated the byte registers `AL, AH, BL, BH, DL, DH` as operands for these instructions.

The basic block that was constructed originally had a length of 3 (for the core instructions), however initialization and finalization assembly is added so the final block is much longer in this case. Initialization assembly (`lines 6-31`) handles register initialization for the 64-bit general purpose registers as well as for the 128-bit `XMM` vector registers.

The finalization assembly (`lines 39-47`) stores the final register state (both general purpose and vector) in memory. That is often convenient, since we would usually issue some type of system call to retrieve the register state (e.g., `printf, write`). Issuing the system call from C without saving the register contents to memory first would disturb the final register state; for example, the arguments to the system call would have to be loaded into registers, overwriting their previous values.

4. X86 EXTENSION

This chapter details the process of incorporating x86 support into the MicroProbe framework. We discuss how the ISA definition files were automatically generated using a parser and an x86 ISA XML specification, the extensions made to MicroProbe to ensure proper functionality for the x86 ISA, and some unique aspects of x86 that must be addressed for correct code generation. Lastly, we describe the `mp_random.py` tool, which introduces support for the constrained random generation of programs in MicroProbe.

4.1 Crafting Configuration Files for x86 Automatically

Let us turn our attention back to the ISA definition files discussed in subsection 3.2.1. Some of these files, such as `register_type.yaml`, `register.yaml`, `operand.yaml`, and `instruction_field.yaml`, are straightforward and can be fully specified by hand.

However, when it comes to instruction formats and the instructions themselves, the files become significantly larger, particularly for a CISC ISA like x86. Manually specifying them is not only impractical but also prone to errors. We are dealing with hundreds of instruction formats and thousands of x86 instruction variants, each with slight variations in operands.

To automate this process, we first obtain a comprehensive, machine-readable x86 ISA specification in XML format. We then build a Python parser to interpret each instruction in the XML specification and generate the corresponding entry in the ISA definition file `instruction.yaml` for MicroProbe. Additionally, if the instruction's format is not already defined, the parser will append the new format definition to `instruction_format.yaml`.

4.1.1 x86 Reference

The full XML reference that was used can be obtained here: `http://ref.x86asm.net/`.

Fig. 4.1 shows a snippet of the XML reference which presents the first two variants of the `ADD` instruction.

Below some of the XML tags and their values are discussed for clarification on the specification's structure. Note that we mainly discuss the first instruction variant (up to `line 16`). The second variant is defined in similar fashion. The full file contains more than 1000 such entries.

- `<one-byte>`
  This section refers to the group of one-byte opcodes. It is organized by the primary opcode byte ('pri_opcd'), which is the first byte of an instruction.
- `<pri_opcd value="00">`
  This indicates that the primary opcode byte (opcode) for this instruction is `00` in hexadecimal. Each primary opcode defines a specific operation.
- `<syntax>`
  The `syntax` tag defines the mnemonic and operands for the instruction:
  - `<mnem>ADD</mnem>`: The mnemonic is the operation to be performed, in this case, the `ADD` instruction.

```
<x86reference version="1.11">

  <one-byte>

    <pri_opcd value="00">
      <entry direction="0" op_size="0" r="yes" lock="yes">
        <syntax>
            <mnem>ADD</mnem>
            <dst><a>E</a><t>b</t></dst>
            <src><a>G</a><t>b</t></src>
        </syntax>
        <grp1>gen</grp1><grp2>arith</grp2><grp3>binary</grp3>
        <modif_f>oszapc</modif_f><def_f>oszapc</def_f>
        <note><brief>Add</brief></note>
      </entry>
    </pri_opcd>

    <pri_opcd value="01">
      <entry direction="0" op_size="1" r="yes" lock="yes">
        <syntax>
            <mnem>ADD</mnem>
            <dst><a>E</a><t>vqp</t></dst>
            <src><a>G</a><t>vqp</t></src>
        </syntax>
        <grp1>gen</grp1><grp2>arith</grp2><grp3>binary</grp3>
        <modif_f>oszapc</modif_f><def_f>oszapc</def_f>
        <note><brief>Add</brief></note>
      </entry>
    </pri_opcd>
...
```

Figure 4.1: x86 XML reference snippet

  - `<dst><a>E</a><t>b</t></dst>`: The destination operand's addressing mode is of type 'E', which means that the operand is either memory or a register, and its type is 'b', which refers to an 8-bit (one-byte) register or memory operand.
  - `<src><a>G</a><t>b</t></src>`: The source operand's addressing mode is of type 'G', denoting a register operand, with a type 'b', that is an 8-bit (one-byte) general-purpose register.

- `<grp1>gen</grp1><grp2>arith</grp2><grp3>binary</grp3>`
  These tags categorize the instruction:
  - `gen`: General-purpose operation.
  - `arith`: Arithmetic operation group.
  - `binary`: A binary operation (operates on two operands).
- `<modif_f>oszapc</modif_f><def_f>oszapc</def_f>`
  These tags describe the flags affected by the instruction:
  - `oszapc`: This refers to the specific flags: Overflow (O), Sign (S), Zero (Z), Auxiliary Carry (A), Parity (P), and Carry (C) flags.

  - The instruction modifies and defines these flags based on the result of the operation.

- `<note><brief>Add</brief></note>`
  This note provides a brief description of the instruction's functionality. In this case, it simply states "Add," reflecting the nature of the `ADD` instruction.

The full XML reference file is 15200 lines long, containing every documented instruction in the x86 ISA.

4.1.2 Parser

To convert the XML reference presented in the previous section into valid entries for `instruction.yaml` (discussed in 3.2.1), we implemented a Python parser consisting of approximately 1000 lines of code. Since the parser is largely composed of lookup tables and conditional statements to handle the various cases, no code snippets are provided here. The details of the decoding process are fairly straightforward and would likely be of little interest to the reader.

Instead, we will show again the relevant entries of `instruction.yaml` that were produced from parsing the above entries of the XML file in Fig. 4.1. It is important to note that the conversion is one-to-many and not one-to-one. This means that each XML entry can produce one or more entries for `instruction.yaml`. This happens because even the same opcode can encode either register or memory operands for example. In MicroProbe, as we mentioned in Sec. 3.2, we encode each operand combination variant for the same opcode of the same mnemonic separately.

In our case, the first XML entry (`lines 5-16`) in Fig. 4.1 produces the two entries shown in Fig. 4.2, whereas the second XML entry (`lines 18-29`) produces six entries shown in Fig. 4.3. We have already discussed the interpretation of entries such as those in Fig. 4.2 and Fig. 4.3 in 3.2.1, specifically at Fig. 3.9 and the relevant text.

```yaml
- Name: "ADD_V0"
  Mnemonic: "ADD"
  Opcode: "00"
  Description: "Add"
  Format: "O8MRR"
  Operands:
    ModRM_Mod : [ '3', 'ModRM_Mod', '?' ]
    ModRM_RM : [ 'R_8', 'ModRM_RM', 'IO']
    ModRM_RegOpc : [ 'R_8', 'ModRM_RegOpc', 'I']
  InstructionChecks:
    C0: [check_rex_registers]
- Name: "ADD_V1"
  Mnemonic: "ADD"
  Opcode: "00"
  Description: "Add"
  Format: "O8MRM8"
  Operands:
    ModRM_Mod : [ '3', 'ModRM_Mod', '?' ]
    ModRM_RM : [ 'ABR_64', 'ModRM_RM', 'IO']
    ModRM_RegOpc : [ 'R_8', 'ModRM_RegOpc', 'I']
    AI_S32: ['AImm32', 'AI_S32', 'I']
  MemoryOperands:
    MEM1 : [ ['ModRM_RM', 'AI_S32'], [1], 1, 'IO']
  InstructionChecks:
    C0: [check_rex_registers]

```

Figure 4.2: instruction.yaml entries from first XML entry

```yaml

- Name: "ADD_V2"
  Mnemonic: "ADD"
  Opcode: "01"
  Description: "Add"
  Format: "O8MRR"
  Operands:
    ModRM_Mod : [ '3', 'ModRM_Mod', '?' ]
    ModRM_RM : [ 'R_16', 'ModRM_RM', 'IO']
    ModRM_RegOpc : [ 'R_16', 'ModRM_RegOpc', 'I']

- Name: "ADD_V3"
  Mnemonic: "ADD"
  Opcode: "01"
  Description: "Add"
  Format: "O8MRR"
  Operands:
    ModRM_Mod : [ '3', 'ModRM_Mod', '?' ]
    ModRM_RM : [ 'R_32', 'ModRM_RM', 'IO']
    ModRM_RegOpc : [ 'R_32', 'ModRM_RegOpc', 'I']

- Name: "ADD_V4"
...

- Name: "ADD_V5"
...

- Name: "ADD_V6"
...

- Name: "ADD_V7"
  Mnemonic: "ADD"
  Opcode: "01"
  Description: "Add"
  Format: "REXO8MRM64"
  Operands:
    REX_W : [ '1', 'REX_W', '?' ]
    REX_X : [ '0', 'REX_X', '?' ]
    ModRM_Mod : [ '3', 'ModRM_Mod', '?' ]
    ModRM_RM : [ 'ABR_64_REX', 'ModRM_RM', 'IO']
    ModRM_RegOpc : [ 'R_64_REX', 'ModRM_RegOpc', 'I']
    AI_S32: ['AImm32', 'AI_S32', 'I']
  MemoryOperands:
    MEM1 : [ ['ModRM_RM', 'AI_S32'], [8], 8, 'IO']


```

Figure 4.3: instruction.yaml entries from second XML entry

4.2 x86-specific Extensions to MicroProbe

In this section, we present the necessary extensions to MicroProbe for x86 code generation that are specific to the x86 ISA.

4.2.1 Code generation primitives

Various elements and primitives required for the code generation process are specific to each ISA. MicroProbe organizes these in the `isa.py` file located within the subdirectory for the respective ISA. For x86, this file defines the `X86ISA` class, which implements these primitives for the x86 architecture. These primitives include, but are not limited to:

1. Retrieving the value of the instruction pointer
2. Storing a value to a register
3. Storing an address to a register
4. Retrieving the set of registers that can store addresses
5. Retrieving the set of registers that can store float values
6. Retrieving the set of control registers
7. Returning the `"NOP"` instruction of the ISA

All of the aforementioned primitives are valuable tools for constructing valid programs in any ISA. In Fig. 4.4, we present the implementation of the second primitive: storing a value in a register. This implementation supports storing a value in general-purpose registers with bit sizes of 8, 16, 32, or 64. Additionally, it handles setting a 128-bit `XMM` register to a target 128-bit value (composed of two 64-bit values).

Let us go over some key points in the implementation:

- The `load_cmd_by_size` dictionary maps the bit-size of a general-purpose register to the appropriate x86 instruction (`MOV_V16`, `MOV_V17`, etc.) which moves the right-sized immediate to the right-sized register.
- For `XMM` registers, which are 128-bit, the implementation handles floating-point values and the process of storing the value involves several steps.
  - It first validates that the value is a list of two 64-bit floating-point numbers.
  - The helper function `ieee_float_to_int64` is used to convert the 64-bit floats into their IEEE 754 integer representations, making them easier to work with.
  - The `toSigned32` function ensures that the converted values are treated as signed 32-bit integers. These values are pushed onto the stack as four 32-bit signed immediate values.
  - Once the values are on the stack, the instruction `MOVDQA_V1` is used to move the 128-bit value from the stack (from `RSP`) into the `XMM` register. This multi-step is necessary due to x86 being unable to directly handle 128-bit immediates.

- For registers other than the `XMM`, the implementation simply uses the appropriate `MOV` instruction (determined by the register size) to move the value directly into the register.
- Notice there are three steps to registering the instruction in this function: 1) Construct the instruction object using the correct mnemonic (e.g. `line 49`), 2) Set the operands, in this case, a destination register and a value, (e.g. `line 50`), 3) Append the instruction object with resolved operands to the list of instructions (e.g. `line 51`).

```
    def set_register(self, register, value, context):
        instrs = []
        # TODO Be careful, these might change if instruction generation is modified
        load_cmd_by_size = {
            8: 'MOV_V16',
            16: 'MOV_V17',
            32: 'MOV_V18',
            64: 'MOV_V19',
        }
        LOG.debug(register)
        LOG.debug(f"Register: {register} Value: {value}")
        # Let's do some tricks to initialize XMM :)
        # We will push 4 32-bit immediates on the stack and then
        # MOVDQA from RSP
        if register.type.name == 'VR128':
            # Expect 2 "double" 64-bit values
            if len(value) != 2 or not isinstance(value[0], float) \
                               or not isinstance(value[1], float):
                raise MicroprobeCodeGenerationError(
                    f"Cannot initialize XMM reg with {value}")

            # Convert to their byte representations
            val0 = ieee_float_to_int64(value[0])
            val1 = ieee_float_to_int64(value[1])

            def toSigned32(n):
                n = n & 0xffffffff
                return (n ^ 0x80000000) - 0x80000000

            # Construct 4 signed 32-bit immediates for stack
            imms_32 = [
                toSigned32(val0 >> 32),
                toSigned32(val0),
                toSigned32(val1 >> 32),
                toSigned32(val1),
            ]

            for imm in imms_32:
                push_imm32_ins = self.new_instruction("PUSH_V1")
                push_imm32_ins.set_operands([imm])
                instrs.append(push_imm32_ins)

            load_xmm_ins = self.new_instruction('MOVDQA_V1')
            load_xmm_ins.set_operands([register, self.registers["RSP"], 0])
            instrs.append(load_xmm_ins)

        # just MOV the value
        else:
            load_ins = self.new_instruction(load_cmd_by_size[register.type.size])
            load_ins.set_operands([register, value])
            instrs.append(load_ins)

        return instrs
```

Figure 4.4: `set_register` **method for x86**

4.2.2 Addressing Modes

Another important aspect unique to x86 is its versatile addressing modes. Unlike load-store RISC architectures, where loading from and storing to memory is restricted to specific instructions, x86 allows memory access across a wide range of instructions, including those like arithmetic instructions. This flexibility enables memory operands to be directly accessed within various operations.

We have already discussed the x86 addressing modes in section 2.1.3.3. We opt to elaborate on how the ***Base + Displacement*** addressing mode is supported in MicroProbe. Similarly, we can support any of the other addressing modes.

To enable support for ***Base + Displacement*** addressing, an additional operand must be encoded for each instruction that can operate on memory. The ***Base***—a register—is already encoded; thus, the focus shifts to encoding the ***Displacement*** operand, which is a signed immediate value that specifies the offset from the base address.

To implement this, we have essentially modified our parser, so that, for every memory operand, an additional displacement operand is encoded automatically, both in the instruction's format and in the instruction's definition. Below, we give an example of an instruction's format and definition with and without the additional displacement operand.

```
# Format definition
- Name: "O8MRM8"
  Fields:
  - opcode_8
  - ModRM_Mod
  - ModRM_RegOpc
  - ModRM_RM
  Assembly: OPC BYTE PTR [ModRM_RM], ModRM_RegOpc

# Instruction definition
- Name: "ADD_V1"
  Mnemonic: "ADD"
  Opcode: "00"
  Description: "Add"
  Format: "O8MRM8"
  Operands:
    ModRM_Mod : [ '3', 'ModRM_Mod', '?' ]
    ModRM_RM : [ 'ABR_64', 'ModRM_RM', 'IO']
    ModRM_RegOpc : [ 'R_8', 'ModRM_RegOpc', 'I']
  MemoryOperands:
    MEM1 : [ ['ModRM_RM'], [1], 1, 'IO']
  InstructionChecks:
    C0: [check_rex_registers]
```

Figure 4.5: `ADD_V1` **base only addressing**

```yaml
# Format definition
- Name: "O8MRM8"
  Fields:
  - opcode_8
  - ModRM_Mod
  - ModRM_RegOpc
  - ModRM_RM
  - AI_S32
  Assembly: OPC BYTE PTR [ModRM_RM + AI_S32], ModRM_RegOpc

# Instruction definition
- Name: "ADD_V1"
  Mnemonic: "ADD"
  Opcode: "00"
  Description: "Add"
  Format: "O8MRM8"
  Operands:
    ModRM_Mod : [ '3', 'ModRM_Mod', '?' ]
    ModRM_RM : [ 'ABR_64', 'ModRM_RM', 'IO']
    ModRM_RegOpc : [ 'R_8', 'ModRM_RegOpc', 'I']
    AI_S32: ['AImm32', 'AI_S32', 'I']
  MemoryOperands:
    MEM1 : [ ['ModRM_RM', 'AI_S32'], [1], 1, 'IO']
  InstructionChecks:
    C0: [check_rex_registers]
```

Figure 4.6: `ADD_V1` **base + displacement addressing**

Looking at Fig. 4.5 compared to Fig. 4.8 we notice the following:

- Regarding the formats, the base only addressing format is missing the `AI_S32` (Address Immediate, Signed 32-bits) field.
- Moreover, the assembly encoded differs. In the base + displacement case the additional field is used to encode the memory address `ModRM_RM + AI_S32`.
- Regarding the instruction definition, again, the `AI_S32` operand is missing in the first case, both from the operand list, as well as from the memory operand specification, which in the second case includes both the base `ModRM_RM` and the offset `AI_S32` since their combination encodes a ***single*** memory location.

4.3 x86 Peculiarities

4.3.1 Implicit operands

One unique characteristic of the x86 architecture is the use of implicit operands in certain instructions (which is very rarely happening in RISC ISAs). Unlike explicit operands, which are directly specified in the instruction, implicit operands are predefined by the instruction itself. For example, many x86 arithmetic operations, such as `MUL` and `DIV`, implicitly use the `AX`, `EAX`, or `RAX` register depending on the operand size. Similarly, instructions like `LOOP` and `REP` implicitly rely on the `CX`, `ECX`, or `RCX` register to control iteration counts.

These implicit operands streamline instruction encoding, reducing the need for extra bits to specify registers. However, they also introduce additional complexity for code generation and analysis, as the usage of certain registers is implicitly tied to specific instructions. Consequently, MicroProbe must account for these implicit operands during code generation to ensure correct register allocation and operand handling.

Consider the following case: Some variants of the `MUL` instruction implicitly store part of their result in `RAX`. Assume the `RAX` register is used as an address base register, but the generator unknowingly inserts a `MUL` instruction. In this case, a segmentation fault is very likely during execution, as the address is most likely corrupted after the `MUL` operation which has overwritten the `RAX` register.

Thankfully, MicroProbe allows us to handle these cases. Each instruction definition optionally includes an attribute named `ImplicitOperands` that lists of the instruction's implicit operands. For example, in the `MUL` instruction, we encode the following:

```
1 ImplicitOperands:
2     RAX: ['RAX', 'O']
```

Figure 4.7: Implicit Operand attribute encoding for MUL

This specifies the fact that `RAX` is an implicit operand and moreover that it is an `'O'` operand, meaning output operand, since part of the multiplication result is stored there. In other cases, the implicit operand might be an `'I'` (input) operand.

4.3.2 Stack Alignment

Stack-related instructions, such as `PUSH` and `POP`, and their variants, can lead to potential issues if not handled correctly. For instance, operations like popping an empty stack can cause unintended crashes due to stack underflow. Additionally, improper use of these instructions can disrupt stack alignment, which is particularly critical for the x86-64 (or amd64) architecture.

The x86-64 ABI specifies that the stack must be aligned to a 16-byte boundary when a function is called. This alignment is crucial as it ensures compatibility with certain instructions and optimizations that assume this alignment, as well as compliance with system conventions. Misalignment can lead to performance penalties or even crashes in some cases, particularly when using SIMD instructions that require aligned memory access.

To address this, we have implemented a re-aligning snippet that is appended after any generated sequence. The generator cannot dynamically trace the state of the stack to decide if it is aligned, that would require simulating a dynamic execution of the code. Our re-aligning snippet instead ensures that the stack is realigned to the 16-byte boundary, preventing any downstream issues caused by misalignment. Thus, it is guaranteed that any generated sequence is ABI-compliant and the risk of stack-related errors during execution is alleviated.

We could realign the stack in many ways, however we opt for the simplest possible approach, simply zeroing the 4 last bits of RSP with the following instruction: `AND RSP, 0xfffffffffffffff0`.

It might not be immediately obvious why we would need to do that, but in practice it is almost always useful. Consider that wrappers are most often C programs and after the

core sequence it is certain that C methods will be called; e.g. `exit()`. Had the stack been misaligned, these methods might cause a crash, so even though our sequence has successfully run without errors, it prevents the error-free termination of the program.

4.3.3 Loading the Address of a Named Variable

This subsection is relevant both to code generation primitives (4.2.1) and addressing modes (2.1.3.3, 4.2.2), however we choose to include it here due to its peculiarity.

To motivate the usefulness of the loading the address of named, static variables consider the following example: We would like to construct sequences that operate on the data of a `1MB` array. Moreover, imagine that after our sequence has operated on that array, we would like to perform some analysis on its values. This can of course be implemented in assembly, however notice that we can easily leverage a C wrapper to both define the array statically and also perform the value analysis in a high-level language, with much less effort.

The question that arises from the example is the following: Say I define the global C array `uint8_t data[1024*1024];` How does I load its base address into a register so that my generated assembly sequence can reference that array?

The answer comes from the combination of the `LEA` (Load Effective Address) instruction with a unique addressing mode of x86, `RIP`-relative addressing. This is a specific form of addressing that leverages the `RIP` (Instruction Pointer) register, allowing the address of a named variable to be loaded in a position-independent manner.

In practice, this means that `LEA` can be used to access global variables or constants that are located at a known offset from the current instruction. For example:

```
LEA RAX, [RIP + my_variable]
```

Here, `my_variable` is a named variable whose address is calculated relative to `RIP`. This ensures that `RAX` will contain the address of `my_variable`, regardless of where the code is loaded in memory.

`RIP`-relative addressing facilitates position-independent code (PIC), meaning the code can be loaded at any memory location without modification. This is particularly useful for shared libraries or dynamically loaded modules, where absolute addressing is impractical due to varying memory locations at runtime.

4.4 Constrained-Random Generation Support

This section explains how constrained-random generation support was added to the framework, via the implementation of the `mp_random` tool. Moreover, we discuss the available constraints users can impose on the sequences being randomly generated. Finally, we give an actual example of constrained-random generation by invoking the `mp_random` tool with some example parameters.

In section 3.4, we presented how we can utilize the `mp_seq` tool to generate all permutations of specific length (with repeats) of a list of provided instructions. Of course, this can only scale up to some tens of instructions and a sufficiently small length. Otherwise, the

sequence space is too enormous to perform the generation of all the sequences in an acceptable amount of time.

In this case, we might like to randomly sample some of the programs from this enormous space, to statistically observe their average behavior. However, the `mp_seq` tool does not allow us such a facility. To that end, we have implemented the `mp_random` tool, with features that allow such random sampling of enormous sequence spaces or the constriction of these spaces to smaller ones via constraints.

4.4.1 The `mp_random` tool

The core of the tool is the following lines of Python code:

```
1
2 # Generate the random sequence for Nth file
3 seq = [
4     ins.name
5     for ins in random.choices(instruction_pool, k=cnt)
6 ]
7
```

Figure 4.8: `mp_random` **core operation**

As we can see from the above code, there exists the notion of an `instruction_pool`. This instruction pool starts with all available x86 instructions that MicroProbe supports. Constraints are then applied to the pool, possibly reducing its size. For example, we might opt to remove all instructions that contain memory operands. The Nth generated sequence is simply a random choice of k instructions from the filtered instruction pool.

The `mp_random` tool is built as a wrapper around the `mp_seq` tool. `mp_seq` already contains the necessary logic to pass the sequences to the code generation module, so that the selected policy is applied to them, in parallel, and they eventually produce a final, valid assembly. We do not replicate this logic in `mp_random`; we simply construct the random instruction sequences and then delegate their generation to the `mp_seq` tool, with appropriate arguments.

The above has the following consequence: `mp_random` has to carry over some of the arguments that `mp_seq` needs. In total, `mp_random` is equipped with the following set of arguments:

1. `-T/--target`: The target-triple for which we perform sequence generation. In our case, `intel64-x86generic-x86_64_linux_gcc`. This argument is also needed by `mp_seq`.
2. `-D/--rnd-output-dir`: The output directory for the generated sequences and final programs. This is passed on to `mp_seq` so that it constructs the final programs in the same directory.
3. `-N/--program-count`: The number of program instruction sequences and programs to generate.
4. `-k/--instruction-count`: The number of random instructions to pick for each generated sequence.

5. `-policy/--seq-policy`: The generation policy that `mp_seq` will apply to the generated sequence to transform into a final program.
6. `-bf/--blacklist-file`: A text file with newline separated instruction mnemonics that are removed from the instruction pool.
7. `-loops`: The number of times the generated sequence should execute in a loop. Passed on to `mp_seq`.
8. `-dd/--dependency-distance`: Register allocation parameter. Passed on to `mp_seq`.
9. `-memsize/--mem-access-size`: Memory pass parameter. Passed on to `mp_seq`.
10. `-memstride/--mem-access-stride`: Memory pass parameter. Passed on to `mp_seq`.
11. `-no-mem-insts`: Remove instructions with memory operands from the instruction pool.
12. `-no-branch-insts`: Remove branch instructions from the instruction pool.
13. `-no-imm-insts`: Remove instructions with immediate operands from the instruction pool.
14. `-no-zero-op-insts`: Remove zero operand instructions from the instruction pool.
15. `-mem-insts-only`: Only keep instructions that have at least one memory operand in the instruction pool.
16. `-branch-insts-only`: Only keep branch instructions in the instruction pool.
17. `-64-bit-only`: Only keep instructions with 64-bit operands in the instruction pool.
18. `-nondet-randint`: Make random values non-deterministic by initializing the random seed with the current time.

Moreover, if a generation policy requires more arguments, these can be added and passed to `mp_seq` which forwards them to the policy implementation function. Also, it is trivial to add new filtering methods for the instruction pool and a corresponding keyword argument.

4.4.2 Implementing Additional Constraints

The `mp_random` tool can be easily customized to support more complex constraints.

A possible example can be requiring a target ***instruction mix*** for the generated sequences. This means we would like our sequence's instructions to obey a specific distribution: e.g. 20% branch instructions, 40% arithmetic operations on integers, 40% arithmetic operations on floats.

MicroProbe already categorizes instructions by their types, so it is easy to implement the above constraint in the following way: 1) Draw $X\%$ of instructions from the relevant group of instructions. 2) Repeat for each component in the target instruction mix. 3) Shuffle all the chosen instructions into the final sequence.

Another example of more complex constraints is ***positional constraints***. When encoding such a constraint, we might require that instruction X is never followed by Y for example.

Another example might be that instruction X should never appear twice any subsequence of length 10.

A simple way to implement this is to repeatedly generate random sequences and test all the positional constraints, until the current sequence satisfies them all. There exist more efficient implementations but this naive ***generate and test*** approach is trivial to implement and sufficient in most cases.

4.4.3 Random Generation Example

For our example, we invoke the `mp_random` tool in the following manner:

```
mp_random.py -T intel64-x86generic-x86_64_linux_gcc \
          -D ./THESIS_RANDOM_GEN/ \
          -N 5 -k 10
          -policy seq -dd 0.0 \
          -memsize 32768 --memstride 16
```

Figure 4.9: mp_random.py invocation example

Essentially, we are asking for the generation of ***5 sequences*** with ***10 instructions each***. The sequences will then be made into programs via the ***seq*** policy with the specified additional parameters. For this example, we have not imposed any constraints on the instruction pool; we are picking our instructions from the complete list of supported instructions.

This invocation generates 5 sequence text files (suffixed with `seq`) and 5 compilable and runnable C programs that contain the sequence of the corresponding sequence file.

Fig. 4.10 gives the contents of two sequence files side-by-side, whereas Fig. 4.11 gives the core contents of the two C files side-by-side.

```
// Sequence File 1          // Sequence File 2
XOR_V22                     MULPD_V1
PSUBB_V1                    SUBSD_V1
IMUL_V11                    CMOVNB_V2
CVTTPD2DQ_V1                DIVSD_V1
CVTSS2SI_V2                 CVTSS2SI_V0
AND_V9                      SHR_V7
CVTPD2PS_V0                 MOVDQA_V2
SAL_V2                      ADDSUBPD_V0
PACKSSWB_V1                 MOVNTPS_V0
JNG_V1                      JC_V1
```

Figure 4.10: `mp_random` sequences example

```
; Program 1                          ; Program 2
LEA RDI, [RIP + ST_2048_16_0]        LEA RDI, [RIP + ST_2048_16_0]
first: XOR R8, -23380                first:MULPD XMM0,  [RDI + 0x0]
PSUBB XMM0,  [RDI + 0x0]             SUBSD XMM1,  [RDI + 0x10]
IMUL EAX, DWORD PTR [RDI + 0x10]     CMOVNB EAX, EBX
CVTTPD2DQ XMM1,  [RDI + 0x20]        DIVSD XMM2,  [RDI + 0x20]
CVTSS2SI R9, XMM2                    CVTSS2SI EDX, XMM3
AND BH, BYTE PTR [RDI + 0x30]        SHR QWORD PTR [RDI + 0x30], 7
CVTPD2PS XMM3, XMM4                  MOVDQA XMM5, XMM4
SAL BL, 78                           ADDSUBPD XMM6, XMM7
PACKSSWB XMM5,  [RDI + 0x40]         MOVNTPS  [RDI + 0x40], XMM0
JNG rel_branch_0                     JC rel_branch_0
rel_branch_0:NOP                     rel_branch_0:NOP
```

Figure 4.11: `mp_random` programs example

Looking at the generated assembly of Fig. 4.11, one might start to appreciate what the framework has automatically done for us. A human programmer might have had a hard-time even guessing valid operands for the above instructions, but the framework has effortlessly filled them in according to our policy.

The complexity that is alleviated is enormous. We can specify various generation tasks that come to mind, and let the framework worry about x86 specifics and how to resolve them in each case. For example, notice that we requested the generation of 10 instructions. However, it so happened, that in both cases the last instruction was a jump. Our policy dictates that jumps are resolved to the next instruction. MicroProbe automatically adds an 11th instruction, a single `NOP`, so that the branch is actually resolved as we had instructed.

Moreover, notice that the memory operands are resolved round-robin with a stride of 16 in an array called `ST_2048_16_0`. As we had requested this array is defined as follows: `uint8_t ST_2048_16_0[32768];` It contains 2048 memory locations, 16 bytes apart, for a total of 32768 bytes.

5. EVALUATION

This chapter presents an in-depth performance analysis of our x86-extension to the MicroProbe framework. The evaluation is based on a series of experiments designed to assess the efficiency, scalability, and resource utilization of the extended framework under various workloads. The results herein aim to provide insights into potential optimizations (some of which have already been implemented) and the expected throughput of our generator under various circumstances.

In the first section we look to identify code hotspots or inefficiencies and potentially optimize them. The second section focuses on comprehensively evaluating the generation throughput of the x86-extension to MicroProbe.

The MicroProbe framework is implemented in Python. Our system runs Python 3.12.3 with the default CPython interpreter where it is not stated otherwise. All experiments in this chapter were carried out on a server with an AMD EPYC 9654 96-Core Processor and 384 GB of DDR5 RAM.

5.1 Performance Optimizations

5.1.1 Caching

Caching is a widely used optimization strategy in computer science, essential for enhancing the efficiency of systems by reducing access times to frequently requested data. By temporarily storing data in faster, more readily accessible locations, caching minimizes the need for repeated access to slower primary storage systems, resulting in substantial improvements in overall system performance. This technique is applicable across numerous domains, including web servers, databases, content delivery networks, and application-level memory management, in addition to traditional CPU caching.

The power of caching lies in its ability to exploit patterns of data access, such as temporal locality—where recently accessed data is likely to be accessed again—and spatial locality—where data near recently accessed information is also likely to be needed. By taking advantage of these patterns, caching can significantly reduce latency and bandwidth consumption, optimize resource utilization, and enable systems to scale efficiently to handle larger volumes of data.

In our case, there is a prime opportunity to apply caching due to the following observation:

Every time a MicroProbe tool (like `mp_seq` or `mp_random`) starts, it has to import all definitions related to the target (e.g. x86) from the architecture module. This causes considerable latency.

Note: Of course there are many opportunities for caching in the whole framework, however this was the most significant unexploited opportunity.

To put this into numbers: Importing the target definitions if they are uncached, takes 0.5 seconds. Restoring the cached target definitions takes almost ***1/10th*** of the time at 0.06 seconds.

Saving 0.45 seconds might seem insignificant but there are cases where a tool is repeatedly called by other scripts. In this case, even more so when the generation job is short, it is important for the tool to not cause additional latencies every time it is brought

up. For example, generating a single program with 1000 instructions using our default `seq` policy takes 1.6 seconds when the target definitions are uncached and 1.2 seconds when they are restored from our caching file. Imagine a user issues that command 1000 times in a loop. Had we not implemented the target definition caching, we would have spent ***1/3 of the total generation time*** loading the target definition, 1000 times in total.

Figure 5.1 includes the relevant code snippet that implements the cache. As we mentioned, if the target definition is cached it is simply reloaded from the file (fast-path), otherwise it is recomputed (slow-path).

```
1     # Import target definition, either cached, or reload
2     ...
3     print_info("Importing target definition...")
4     import_start_t = time.time()
5     cache_path = str(Path(__file__).parent.joinpath('target_def.cache'))
6
7     # Either load from cache (super-fast)
8     if os.path.exists(cache_path) and "drop_target_cache" not in arguments:
9         recreate_module_environment()
10        with open(cache_path, "rb") as f:
11            target = pickle.load(f)
12        print_info("Successful load from cache!")
13    # Or re-import
14    else:
15        print_info("Recomputing target definition...")
16        target = import_definition(arguments.pop('target'))
17        with open(cache_path, 'wb') as f:
18            pickle.dump(target, f)
19
20        # Verify read
21        with open(cache_path, 'rb') as f:
22            test = pickle.load(f)
23            print_info("Pickle read verified.")
24
25        target = test
26        print_info(f"Pickle-stored it to {cache_path}...")
27
28    import_elapsed_t = time.time() - import_start_t
29    print_info(f"Target definition import took {import_elapsed_t:.6f} seconds!")
30    ...
```

Figure 5.1: Target definition cache snippet

5.1.2 String Builders

Take a look at the code in Figure. 5.2. The aim of the snippet here is to produce C code that initializes an array variable. An example result might be: `uint8_t my_array[5] = {1, 2, 3, 4, 5};`

The code might seem fine but a big inefficiency hides in this simple snippet. Assume that the array has one million elements. What will line 19 do in this case? Construct 999 999 concatenated strings by string copying into a new string each time, even when the string has already become kilobytes long? This is a performance nightmare.

Let us give the actual numbers for two simpler examples. The first example (Fig. 5.3) performs the inefficient string building we showcased from MicroProbe's code base, whereas the second example (Fig. 5.4) performs optimized string building with a list and the `join` method in python.

The difference in performance is astounding: ***For 100,000 elements, the first program executes in 13.5 seconds whereas the second in only 0.01 seconds!!! That's more than a 1000x slowdown in the first case!***

```
...
if var.array():

    if var.value is not None:

        valuestr = ""
        value = var.value
        if not isinstance(value, list):
            value = [value]

        get_value = getnextf(itertools.cycle(value))

        for dummy_idx in range(var.elems):
            value = get_value()
            if callable(value):
                value = value()

            valuestr = "%s%s," % (valuestr, value)

        return "%s %s[%d] %s = {%s};\n" % (
            var.type, var.name, var.size, align, valuestr
        )
```

Figure 5.2: String copying inefficiency in MicroProbe

```
def inefficient_string_build(num_elements):
    valuestr = ""
    for i in range(num_elements):
        valuestr = "%s%d," % (valuestr, i)

    return valuestr[:-1]  # Remove the trailing comma

```

Figure 5.3: Inefficient string join

```
def efficient_string_build(num_elements):
    valuestr_list = []

    for i in range(num_elements):
        valuestr_list.append(str(i))

    return ",".join(valuestr_list)  # Join all values into a single string
```

Figure 5.4: Efficient string join

The key point is that in string processing, using string builders is crucial for efficient manipulation of strings, particularly in scenarios involving frequent concatenation or modification. Unlike immutable strings, which require creating new copies for each change, string builders allow modifications directly on the existing buffer, avoiding the overhead of constant memory allocation and copying.

String builders come in different forms in each programming language. For example, Java has an explicit StringBuilder class whereas in Python we have ”string-builder like” behaviour using an array and the `join` method.

5.1.3 Flame Graphs

It is well-known that premature optimization is the root of all evil. Before embarking on any optimization efforts, it is crucial to measure performance accurately to identify bottlenecks and understand where time is being spent within an application. By collecting and analyzing performance data, developers can focus their optimization efforts where they will have the most significant impact.

Flame graphs are a powerful visualization tool that help in this regard. They provide a compact and intuitive way to represent profiled stack traces, allowing developers to see which functions consume the most CPU time at a glance. In a flame graph, each box represents a function, and the width of each box correlates with the amount of time spent in that function, including time spent in child functions. This hierarchical representation enables quick identification of hotspots and inefficiencies within the codebase.

The rest of the subsection is structured as follows: We first describe an optimization that was implemented into MicroProbe. The high-value of an optimization in that snippet was discovered through flame graph inspection. Then, we give flamegraphs of executions and try to find other chances for optimization of hot code.

Let us give some background regarding the optimized snippet so that the context is better set: MicroProbe’s code generation module contains passes that perform register allocation as we have already seen in previous chapters. During register allocation there exists the need to compare registers for equality or symmetrically inequality. Registers in MicroProbe are instantiated as object of a `Register` class and contain many other attributes other than their short name. Moreover, dynamic attributes might be inserted to the register objects. Thus, the proper way to compare these objects is shown in the following figure.

```
1 def __eq__(self, other):
2     """x.__eq__(y) <==> x==y"""
3
4     self._same_class(other)
5     for attr in self._cmp_attributes:
6         if not getattr(self, attr) == getattr(other, attr):
7             return False
8     return True
```

Figure 5.5: Naive `__eq__` for register object

Essentially, we define a set of comparison attributes and use Python’s reflection features to dynamically retrieve these attributes from both objects (provided they are of the same, correct class!). If the attributes’ values match, then the objects are considered equal.

During a register allocation pass, this method and its symmetric (not equal) get called ***lots*** of times to compare registers. We found that the following shortcut to the comparison greatly improves performance in almost all cases, without any loss to the correctness of the comparison. MicroProbe also compares register types in the same way, so these also benefit from the same optimization!

```
def __eq__(self, other):
    """x.__eq__(y) <==> x==y"""

    # shortcut
    if self._hash != other._hash:
        return False

    self._same_class(other)
    for attr in self._cmp_attributes:
        if not getattr(self, attr) == getattr(other, attr):
            return False
    return True
```

Figure 5.6: Optimized `__eq__` for register object

If the built-in hash values of the objects are different, they are certainly not equal. This early return pays off massively, given the amount of times this method is called during generation. An inspection of the new flame graph after the optimization, shows significant reduction to the time spent in these methods.

Let us now look at the flame graph of an execution after the aforementioned optimizations. We are using the `cProfile` Python module to collect the performance data and stack traces. The generated profile file is then passed into the `snakeviz` module for visualization of the flame graph.

Specifically, we task the generator with the following:

```
mp_random.py -T intel64-x86generic-x86_64_linux_gcc -policy seq -D . -N 1 -k
1000 -dd 0 -memsize 16384 -memstride 16
```

which means: *Generate one x86-64 program with 1,000 instructions using the default seq policy. Resolve memory accesses in a 16KB area with 16 bytes of stride, round-robin. Allocate registers with the maximum possible dependency distance.*

As we mentioned earlier, `mp_random` is a wrapper for `mp_seq` so the code generation job mainly happens inside `mp_seq`. `mp_random` merely produces the random instruction mnemonics sequence, which takes up almost no time.

Fig. 5.7 shows the resulting flame graph. As we mentioned earlier the boxes represent time spent inside a function. They are organized in a hierarchical fashion, top-down. We notice the following:

1. Overall, there is a good distribution of time among different functions. There is no single leaf-level box that takes up the whole graph.
2. However, almost half of the time is spent in the grey box, which corresponds to the ***register allocation pass***.
3. Notice the largest leaf-call box. It is the `__eq__` method of the register type class we discussed and optimized earlier!

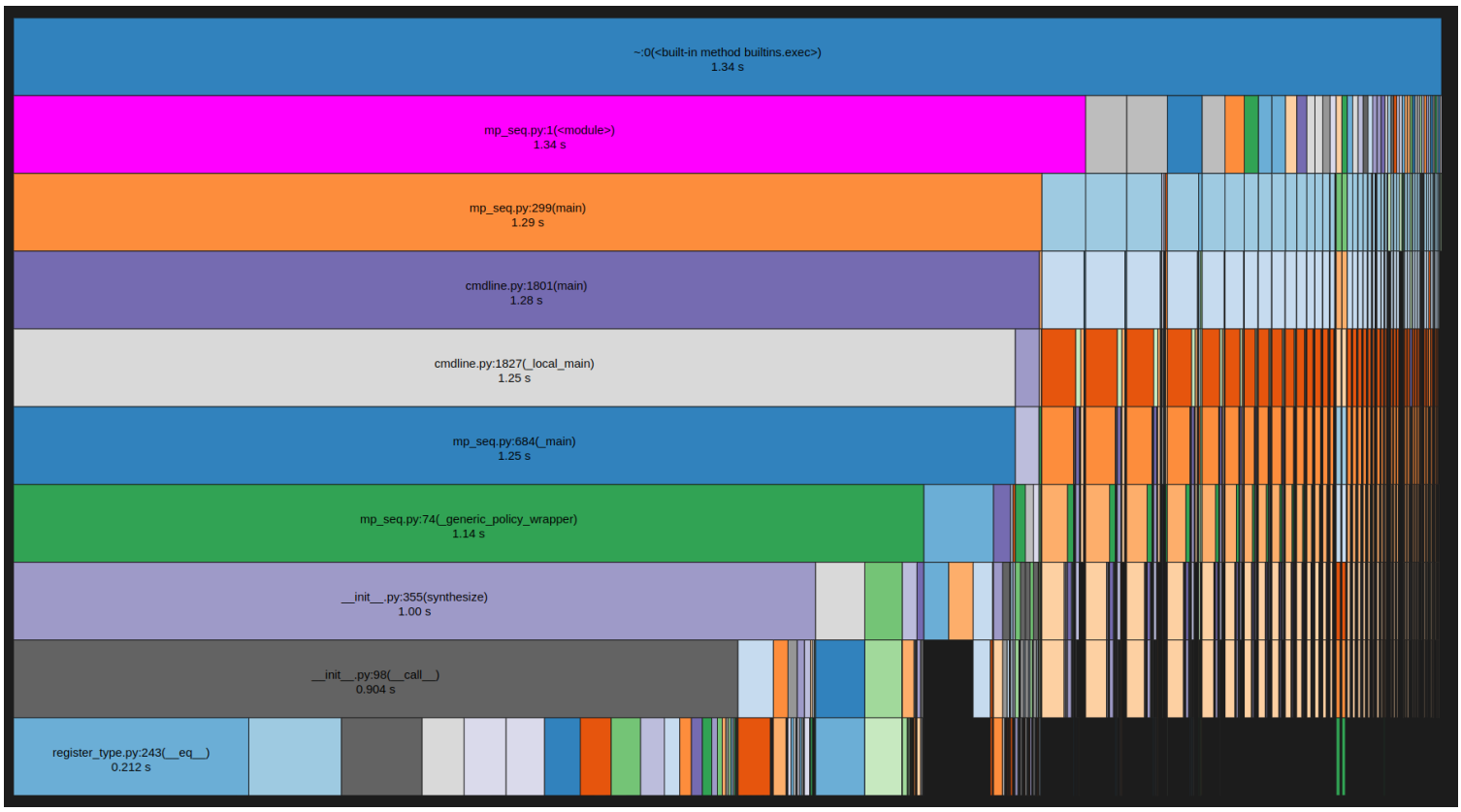

Figure 5.7: Flame graph 1,000 instructions generation

The situation - other than the register equality comparison - seems balanced. There is no single function that desperately calls for optimization because it is taking up all the runtime.

Let us stress the code generation module further. We will ask for the same generation job with one difference: program size will now be 100,000 instructions. Fig. 5.8 shows the results and they are not good.

This is what a terrible flame graph looks like. Notice the following:

1. We spend 80% of the execution inside the register allocation pass (light grey-box at the third level)
2. The rest of the boxes, excluding the blue equality comparison box which is huge, are tiny. This shows that we have a big timewaster. The register allocation pass and especially the equality comparisons that are called millions of times.

Let us now see if we can do something more about the register type equality (blue box in Fig. 5.8). The implementation is the following:

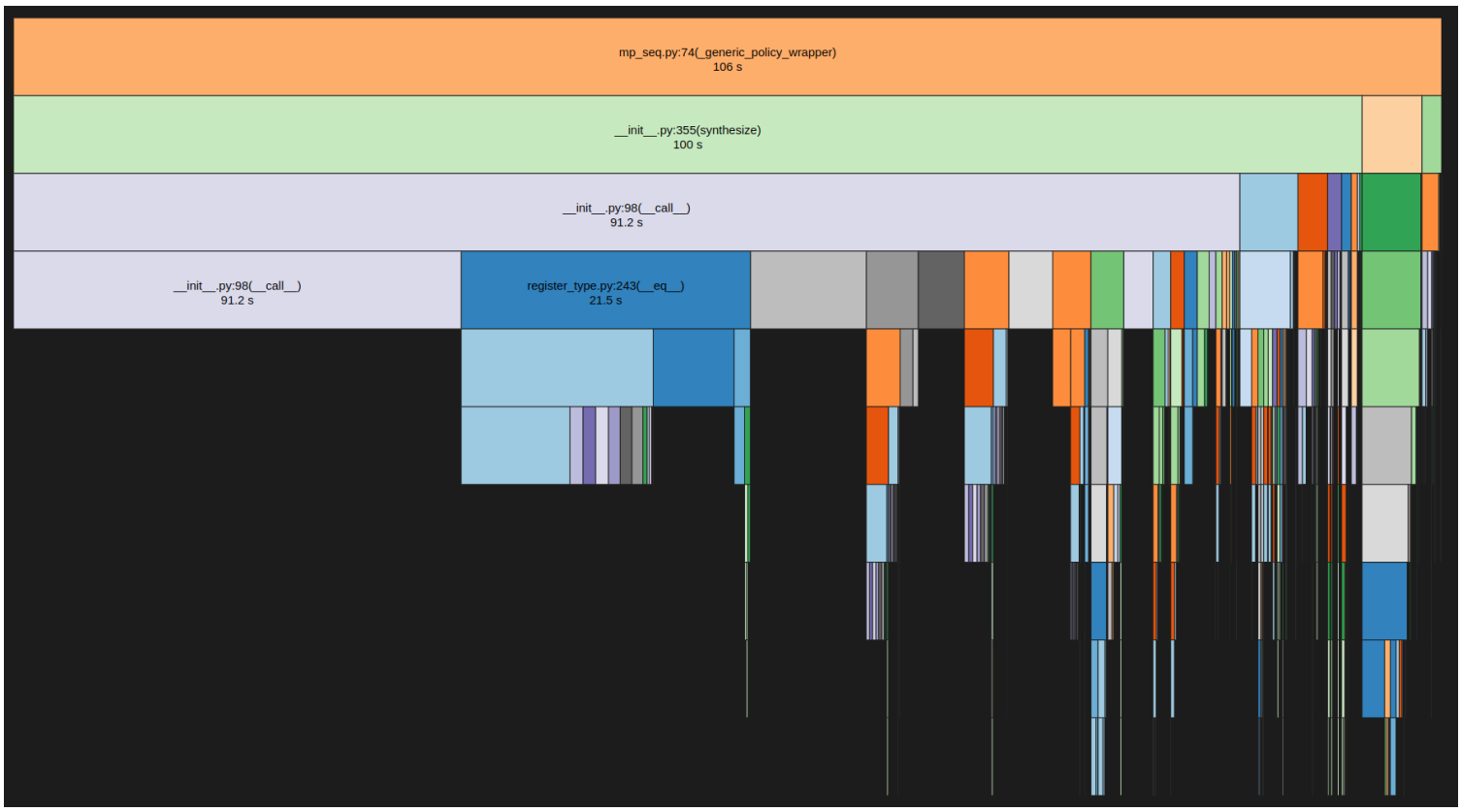

Figure 5.8: Flame graph 100,000 instructions generation

```
1  def __eq__(self, other):
2      """x.__eq__(y) <==> x==y"""
3
4      # shortcut
5      if self._hash != other._hash:
6          return False
7
8      self._same_class(other)
9      for attr in self._cmp_attributes:
10         if not getattr(self, attr) == getattr(other, attr):
11             return False
12     return True
```

Figure 5.9: Current `__eq__` for register object

Even though we have implemented a comparison shortcut, there is still a problem that remains: We compare about 5 attributes that are retrieved dynamically via `getattr`. This is relatively slow. However if we think of the actual problem at hand, we just have to decide if two register type objects are the same. Provided we have unique names for each register type the problem would simply boil down to comparing the names. In practice, this happens for the x86 definitions so we could define the following method to compare register types, though it is not general and possibly unsafe in some other cases.

```
def __eq__(self, other):
    """x.__eq__(y) <==> x==y"""
    if self.name != other.name:
        return False
    if self.size != other.size:
        return False
    return True
```

Figure 5.10: Fast (possibly unsafe) `__eq__` for register object

Let us repeat the experiment with the new method and collect the new flame graph. Fig. 5.11 shows the new results. We have succeeded in reducing the time spent in the `__eq__` method however the problem still remains: ***We are spending most of the time solely in the register allocation pass***.

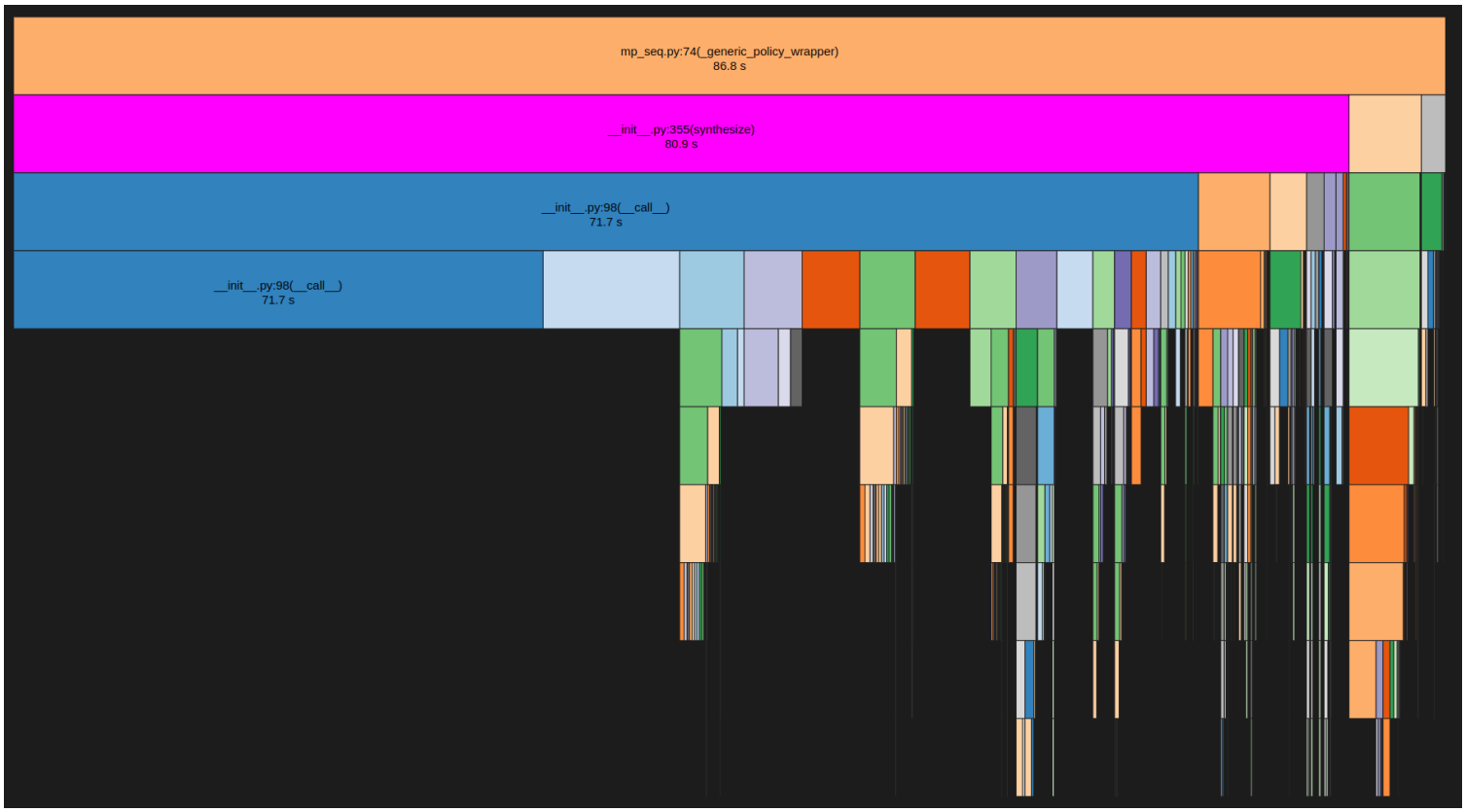

Figure 5.11: Flame graph 100,000 instructions generation, fast-unsafe __eq__

This calls either for a redesign of the pass (i.e. use a better algorithm!) or to simply avoid using the pass where not needed. To demonstrate, let us swap the register allocation pass to random register allocation. Surely, the cost of randomly allocating the registers is lower.

The new flame graph is shown in Fig. 5.13. Obviously, the situation has improved dramatically. We can see that the passes called under the `synthesize` grey block, each take up a similar amount of time. Overall the flame graph is well-balanced. Moreover note the **total generation time is now 20 seconds down from 100!!!**

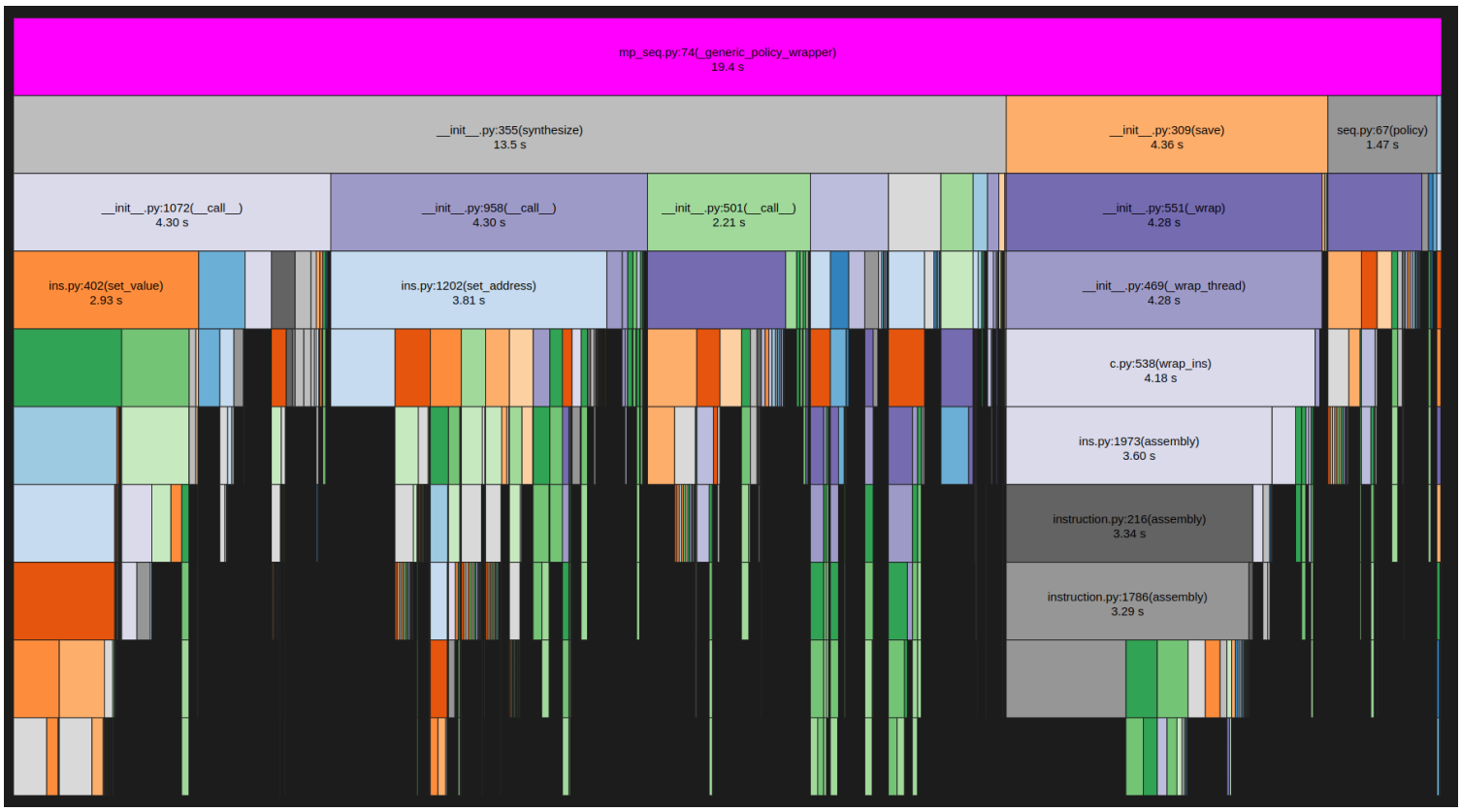

Figure 5.12: Flame graph 100,000 instructions generation, random register allocation

5.2 Python Backend Performance Comparison

In this section we will test the throughput of our generator in various scenarios: We compare serial generation throughput as well as parallel throughput to observe how well the framework's code generation scales.

Moreover, we compare two Python backends, the default CPython implementation, as well as PyPy, which is a JIT implementation, much faster in some cases.

For all the following experiments the base command issued is: `mp_random.py -T intel64-x86generi 64_linux_gcc -policy seq -D . -memsize 16384 -memstride 16 -N <programs_to_generate> -k <instructions_in_sequence>`.

Note that we present two measurements of time: wall time and CPU time. In summary, wall time represents the total time taken for a task to complete, while CPU time measures the active processing time by the CPU. A detailed comparison follows.

- **Wall Time (Real Time)**: This is the total elapsed time from the start to the finish of a task, measured in real-world seconds. It includes all the time spent, including time waiting for resources, I/O operations, and time when the CPU is not actively processing the task. Essentially, it represents the "real" time it takes for a user to see the results of a computation.
- **CPU Time (User Time)**: This measures the actual time the CPU spends processing instructions for a given task. It accounts for the time the CPU is actively engaged in executing the program's code, **excluding** any time spent waiting for I/O operations or when the program is not being executed (e.g., waiting for memory access). CPU

time reflects the computational workload rather than the total time taken to complete a task.

5.2.1 Serial Performance

For this set of experiments $N = 1$, thus we generate one program with 1000, 5000 and 10000 instructions.

5.2.1.1 CPython

First, we examine the serial case for the default CPython interpreter. Table 5.1 shows the results. We can see that increasing the number of instructions does not linearly increase the generation time. If this was the case, generating 10,000 instruction would have taken $1.175 \times 10 = 11.75$ seconds however it takes only 3.65 seconds!

Table 5.1: [CPython] N=1 program for various k (number of instructions)

Instructions	Wall Time (s)	CPU Time (s)
1000	1.175	0.852
5000	2.261	1.932
10000	3.654	3.306

5.2.1.2 PyPy

For the PyPy experiments we are running PyPy 7.3.15 with GCC 13.2.0 (Python 3.9.18). As we mentioned PyPy is a JIT implementation. In some cases, its performance can far exceed the default CPython implementation.

Table 5.2: [PyPy] N=1 program for various k (number of instructions)

Instructions	Wall Time (s)	CPU Time (s)
1000	3.746	3.204
5000	4.062	3.546
10000	4.174	3.668

In Table 5.2, we observe that PyPy exhibits slower performance than CPython for serial program generation, particularly with smaller instruction numbers. This discrepancy can be attributed to several factors related to the nature of JIT (Just-In-Time) compilation:

1. **Warm-Up Time:** JIT compilation introduces an initial overhead, as it compiles bytecode into machine code at runtime. For small tasks, this overhead may outweigh the benefits of optimized execution, resulting in slower overall performance.
2. **Optimization Delay:** PyPy's JIT compiler optimizes code during execution based on runtime profiling. For shorter tasks, the time available for the JIT to optimize the code may be insufficient, leading to less efficient execution compared to the already-optimized bytecode in CPython.

3. **Overhead of Tracing:** JIT compilation involves tracing the execution of code to identify hot paths and apply optimizations. In smaller tasks, the proportion of time spent in tracing and compiling can be significant, contributing to increased execution time.

4. **Garbage Collection:** PyPy's garbage collection mechanisms can also introduce latency, particularly if frequent allocations and deallocations occur during short-lived tasks.

As the instruction count increases, PyPy's JIT optimizations can be fully leveraged, resulting in performance that closely matches or even surpasses that of CPython. This highlights the importance of task size and complexity in determining the effectiveness of JIT compilation.

We conducted an additional experiment to showcase the power of JIT compilation. We generate programs with 1,000,000 instructions both with CPython and PyPy. The results are astounding: ***CPython requires 4.5 minutes or 270 seconds to complete the generation process while PyPy is done in only 52 seconds!!! That is a 5x speedup.***

5.2.2 Parallel Performance

For evaluating the scalability of MicroProbe's generation of x86 code when the generation of multiple programs is requested, we conduct the following experiments: We request parallel code generation for N = 4, 16, 48, 96 programs with k = 1000, 5000, 10000 instructions. These experiments are repeated both for the default CPython interpreter as well as the PyPy JIT implementation.

The below table contains all the relevant numbers.

Table 5.3: Comparison of Real Time and User Time for PyPy and CPython

N	k	PyPy Real (s)	PyPy User (s)	CPython Real (s)	CPython User (s)
4	1000	4.095	4.071	2.115	1.858
4	5000	4.974	4.911	6.425	6.144
4	10000	6.130	6.064	11.807	11.529
16	1000	5.890	11.462	2.558	6.232
16	5000	7.015	15.211	6.998	23.834
16	10000	7.997	19.618	12.487	45.266
48	1000	10.572	31.147	3.646	18.127
48	5000	11.258	43.238	8.268	69.669
48	10000	12.516	55.353	14.013	134.999
96	1000	17.578	60.794	5.382	35.545
96	5000	18.747	85.478	10.293	139.825
96	10000	19.271	110.749	16.506	269.882

We succinctly summarize the throughput insights given by the wall (real) time in the following figure.

From the graph we can draw the following conclusions:

1. CPython is faster by almost an average factor of 3 when the program generated is small. (k = 1000 instructions)
2. As N (number of programs generated in parallel) increases, the overhead of instantiating multiple PyPy instances (each with their own JIT) becomes larger. For example, PyPy performs almost the same for $N = 96$ irrespective of k. This means that managing the 96 JIT environments is much more costly than the actual generation happening.
3. For the N = 4, 16 and k = 5000, 10000 PyPy comes out on top. Increasing the program's size means their is more repetitive work to do for each instruction. This is where the JIT excels, and its costs (which we mentioned earlier) are covered by the benefits.

The general conclusion is as follows:

- **Small Programs:** Use CPython due to JIT overheads.
- **Large Programs:** Use PyPy because of JIT benefits.
- **Massively Parallel Generation:** Use CPython due to lower management and synchronization costs compared to JIT instances.

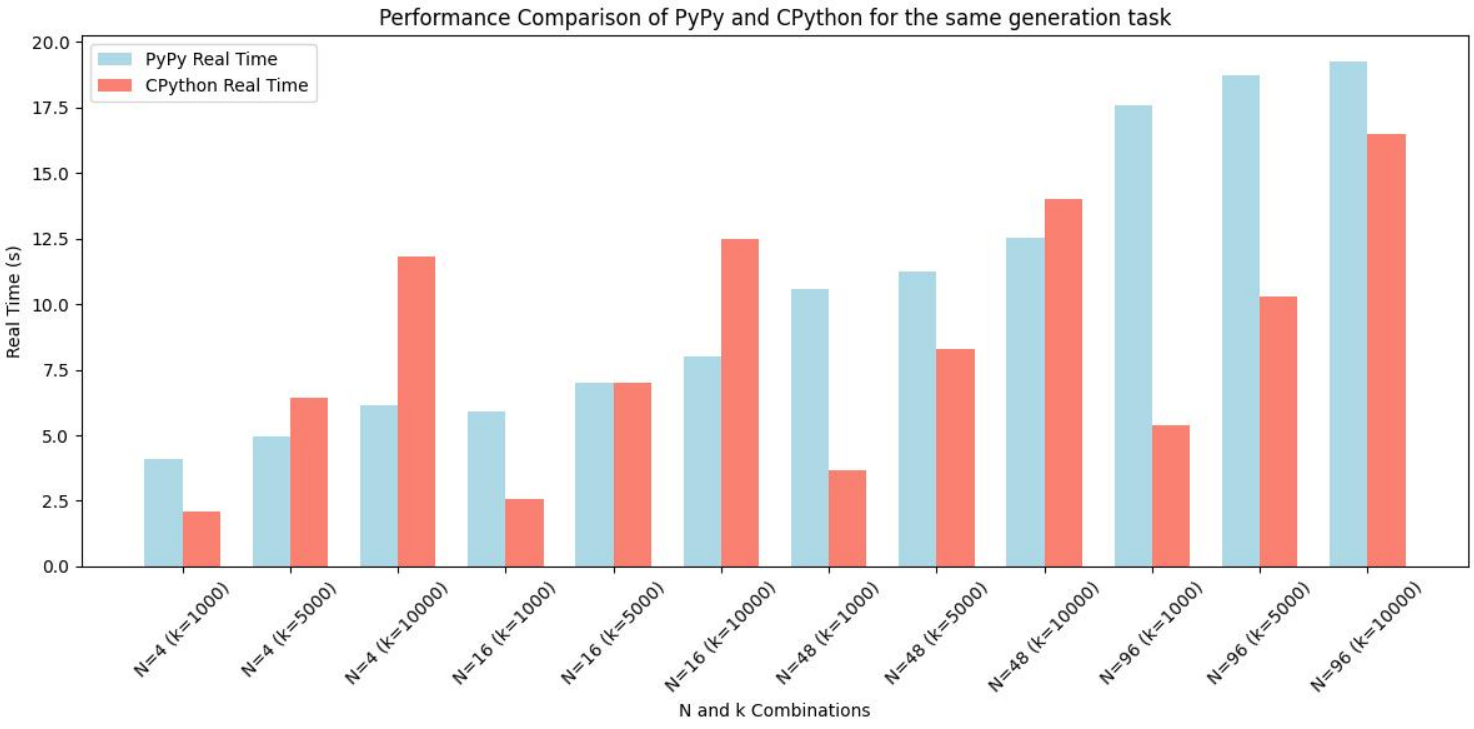

Figure 5.13: Parallel performance comparison of Python backends for MicroProbe

6. CONCLUSION

This thesis has presented a comprehensive overview of the x86 extension we have implemented into the MicroProbe framework, alongside the related concepts and background necessary for understanding the framework's capabilities. To our knowledge this is the first highly configurable and modular x86-64 code generator.

The primary points of discussion in this thesis have been:

- **The x86 ISA**: History, comparison to RISC ISAs, introduction to x86 programming.
- **The YAML format**: Used for configuration files in MicroProbe.
- **The MicroProbe framework**: In depth explanation of MicroProbe's structure and code generation examples.
- **The x86 extension of MicroProbe**: Comprehensive and step-by-step explanation of the extension process.
- **Evaluation of the x86 generator**: We discussed code optimizations to the framework discovered via profiling as well as a evaluated its throughput with various experiments.

Overall, this work lays the groundwork for further exploration into x86 assembly generation and the MicroProbe framework, paving the way for further advancements in automated code generation.

BIBLIOGRAPHY

[1] MicroProbe official documentation: `https://ibm.github.io/microprobe/`.

[2] Combined volume set of intel® 64 and ia-32 architectures software developer's manuals.

[3] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. Genesys-pro: innovations in test program generation for functional processor verification. *IEEE Design Test of Computers*, 21(2):84–93, 2004.

[4] Ramon Bertran, Alper Buyuktosunoglu, Meeta S. Gupta, Marc Gonzalez, and Pradip Bose. Systematic energy characterization of cmp/smt processor systems via automated micro-benchmarks. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–211, 2012.

[5] Paulo Borba and Sérgio Soares. Refactoring and code generation tools for aspectj. In *Proc. of the Workshop on Tools for Aspect-Oriented Software Development (with OOPSLA)*, 2002.

[6] F. Corno, G. Cumani, M. Sonza Reorda, and G. Squillero. Fully automatic test program generation for microprocessor cores. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 1006–1011, 2003.

[7] F. Corno, E. Sanchez, M.S. Reorda, and G. Squillero. Automatic test program generation: a case study. *IEEE Design Test of Computers*, 21(2):102–109, 2004.

[8] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 95–105, 2018.

[9] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, 2007.

[10] Nikos Foutris, Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera, and Antonio Gonzalez. Accelerating microprocessor silicon validation by exposing isa diversity. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 386–397, 2011.

[11] Nikos Foutris, Dimitris Gizopoulos, Xavier Vera, and Antonio Gonzalez. Deconfigurable microprocessor architectures for silicon debug acceleration. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 631–642, 2013.

[12] Dimitris Gizopoulos, Mihalis Psarakis, Miltiadis Hatzimihail, Michail Maniatakos, Antonis Paschalis, Anand Raghunathan, and Srivaths Ravi. Systematic software-based self-test for pipelined processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(11):1441–1453, 2008.

[13] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[14] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1:1–13, 2018.

[15] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1044–1051, 2019.

[16] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation, 2018.

[17] George Papadimitriou, Athanasios Chatzidimitriou, Manolis Kaliorakis, Yannos Vastakis, and Dimitris Gizopoulos. Micro-viruses for fast system-level voltage margins characterization in multicore cpus. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 54–63. IEEE, 2018.

[18] George Papadimitriou, Dimitris Gizopoulos, Athanasios Chatzidimitriou, Tom Kolan, Anatoly Koyfman, Ronny Morad, and Vitali Sokhin. Unveiling difficult bugs in address translation caching arrays for effective post-silicon validation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 544–551. IEEE, 2016.

[19] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2020.

[20] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design Test of Computers*, 27(3):4–19, 2010.

[21] Yiannakis Sazeides, Alex Gerber, Ron Gabor, Arkady Bramnik, George Papadimitriou, Dimitris Gizopoulos, Chrysostomos Nicopoulos, Giorgos Dimitrakopoulos, and Karyofyllis Patsidis. Idld: Instantaneous detection of leakage and duplication of identifiers used for register renaming. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 799–814. IEEE, 2022.

[22] Karthik Swaminathan, Ramon Bertran, Hans Jacobson, Prabhakar Kudva, and Pradip Bose. Generation of stressmarks for early stage soft-error modeling. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S)*, pages 42–48, 2019.

[23] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3237–3254, Boston, MA, August 2022. USENIX Association.

[24] Haoyu Wang, Junjie Chen, Chuyue Xie, Shuang Liu, Zan Wang, Qingchao Shen, and Yingquan Zhao. Mlirsmith: Random program generation for fuzzing mlir compiler infrastructure. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1555–1566, 2023.