UNIVERSIDADE FEDERAL DO PAMPA

TOMÁS ROSSATO BENFICA

Desenvolvimento de um Gerenciador de Contexto de Memória para Processador Python

Bagé
2024

TOMÁS ROSSATO BENFICA

Desenvolvimento de um Gerenciador de Contexto de Memória para Processador Python

Projeto de Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Engenharia de Computação como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Bruno Silveira Neves

Bagé
2024

Ficha catalográfica elaborada automaticamente com os dados fornecidos
pelo(a) autor(a) através do Módulo de Biblioteca do
Sistema GURI (Gestão Unificada de Recursos Institucionais) .

B465d Benfica, Tomás Rossato
Desenvolvimento de um Gerenciador de Contexto de Memória para Processador Python / Tomás Rossato Benfica.
93 p.

Trabalho de Conclusão de Curso(Graduação)-- Universidade Federal do Pampa, ENGENHARIA DE COMPUTAÇÃO, 2024.
"Orientação: Bruno Silveira Neves".

1. Gerenciador de Contexto de Memória. 2. Processador de Bytecode Python. 3. Execução de Funções. 4. Arquitetura em Hardware. 5. Aninhamento de Chamadas. I. Título.

SERVIÇO PÚBLICO FEDERAL
MINISTÉRIO DA EDUCAÇÃO
Universidade Federal do Pampa

TOMÁS ROSSATO BENFICA

DESENVOLVIMENTO DE UM GERENCIADOR DE CONTEXTO DE MEMÓRIA PARA PROCESSADOR PYTHON

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Computação da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Computação.

Trabalho de Conclusão de Curso defendido e aprovado em: 11 de Dezembro de 2024.

Banca examinadora:

_____

Prof. Dr. Bruno Silveira Neves
Orientador
(Universidade Federal do Pampa - UNIPAMPA)

_____

Prof. Dr. Carlos Michel Betemps
(Universidade Federal do Pampa - UNIPAMPA)

_____

Prof. Dr. Fábio Luís Livi Ramos
(Universidade Federal do Pampa - UNIPAMPA)

Assinado eletronicamente por **BRUNO SILVEIRA NEVES**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 20/12/2024, às 01:19, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **FABIO LUIS LIVI RAMOS**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 20/12/2024, às 16:00, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1633803** e o código CRC **2E4B6FED**.

Referência: Processo nº 23100.023005/2024-63 SEI nº 1633803

Dedico este trabalho à minha família, especialmente aos meus pais, que, com amor, esforço e dedicação, construíram a base de uma família unida e me inspiraram a chegar até aqui.

AGRADECIMENTO

Agradeço aos meus professores, que não apenas me ensinaram a exercer a profissão que escolhi para a minha vida, mas também foram exemplos de caráter, paciência e generosidade. Com dedicação e esforço, garantiram que eu e meus colegas realmente aprendêssemos, indo além do básico. Em especial, agradeço aos professores Bruno, Gerson e Fábio, que, com alegria, afirmo terem se tornado amigos ao longo da jornada. Mais do que mestres, vocês foram modelos de inspiração, e parte do que sou hoje devo a vocês.

Sou profundamente grato à Unipampa, uma instituição que tenho orgulho de ter como minha casa de formação. Em breve, terei o privilégio de fazer parte dos murais de formandos que adornam seus corredores. Os momentos que vivi lá marcaram minha vida de forma indelével, e levarei comigo as memórias construídas nesse espaço que me acolheu e me transformou.

Aos amigos que se tornaram irmãos, meu mais sincero agradecimento. Dizem que amigos são o sal da vida, e posso afirmar com alegria que fiz vários durante essa jornada: Ignácio, Guilherme G., Thiagão, Kelvin, Estefanie, Heitor, Gabriel, Michel, Dalmazo, Guilherme C., Renato e Samuel. Pude superar muitas adversidades graças a vocês, que tornaram os momentos ruins mais leves. Compartilhar momentos com vocês foi um presente. São pessoas incríveis, e as memórias que construímos juntos ficarão comigo para sempre. Desejo a cada um uma vida plena e feliz, de coração.

Por fim, este trabalho, que demandou tanto tempo, esforço e dedicação, é, acima de tudo, uma consequência do amor incondicional que recebo desde sempre dos meus pais. Agradeço a eles, João e Veranice, que lançaram as bases do que sou hoje de forma tão profunda que me faltam palavras para expressar. Com eles aprendi, entre tantas outras coisas, a importância do esforço e da perseverança — lições inestimáveis transmitidas com amor, paciência e dedicação. Sou eternamente grato por tudo o que fizeram e continuam fazendo por mim.

“Não ridicularizar, não lamentar, nem detestar, mas compreender.“
(Non ridere, non lugere, neque detestari, sed intelligere.) — Baruch Espinoza

RESUMO

Este trabalho apresenta o desenvolvimento de um software Gerenciador de Contexto de Memória para um processador de bytecode Python implementado em hardware, com o objetivo de otimizar o gerenciamento da execução de funções e a troca de objetos-frame entre o processador e o ambiente de execução. Em sistemas baseados em bytecode, como o Python, é fundamental garantir a integridade da pilha de execução e a correta manipulação das variáveis globais durante a execução de funções, principalmente em contextos de aninhamento profundo de chamadas. O Gerenciador foi projetado para permitir a comunicação eficiente entre o processador e o software de controle. A implementação foi realizada em C++ e testada em um ambiente controlado, onde a atualização dos objetos-frame, o envio de *payloads* e a persistência das variáveis globais foram avaliados. Os resultados dos testes confirmaram a eficácia do sistema, destacando a capacidade do Gerenciador em lidar com cenários complexos de execução e garantir a consistência e a confiabilidade do processo. Este trabalho contribui para o avanço na implementação de processadores de bytecode Python em hardware, oferecendo uma solução robusta e escalável para o gerenciamento de contexto e execução de código. Futuras melhorias incluem a implementação de testes automatizados e validações em ambientes reais de produção.

Palavras-chave: Gerenciador de Contexto de Memória. Processador de Bytecode Python. Execução de Funções. Arquitetura em Hardware. Persistência de Variáveis Globais. Aninhamento de Chamadas.

ABSTRACT

This work presents the development of a Memory Context Manager software for a Python bytecode processor implemented in hardware, with the aim of optimizing the management of function execution and the exchange of frame objects between the processor and the execution environment. In bytecode-based systems, such as Python, it is crucial to ensure the integrity of the execution stack and the correct handling of global variables during function execution, especially in contexts of deeply nested calls. The Manager was designed to enable efficient communication between the processor and the control software. The implementation was carried out in C++ and tested in a controlled environment, where the updating of frame objects, the sending of payloads, and the persistence of global variables were evaluated. Test results confirmed the system's effectiveness, highlighting the Manager's ability to handle complex execution scenarios and ensure consistency and reliability. This work contributes to the advancement of Python bytecode processor implementations in hardware, offering a robust and scalable solution for context management and code execution. Future improvements include the implementation of automated testing and validations in real production environments.

Keywords: Memory Context Manager, Python Bytecode Processor, Function Execution, Hardware Architecture, Persistence of Global Variables, Call Nesting.

LISTA DE FIGURAS

Figura 1 Código da função `FizzBuzz` 35
Figura 2 Objeto-código gerado para a função `FizzBuzz` 36
Figura 3 Lista de campos que constituem um objeto-frame. 37
Figura 4 Fluxo de execução típico num programa Python 39
Figura 5 Estrutura de um arquivo PYC 39
Figura 6 Aninhamento de objetos-código num arquivo PYC 41
Figura 7 Bytecodes para criar objeto-classe 43
Figura 8 Bytecodes para instanciação de objeto 45
Figura 9 Exemplo de bytecodes para heranca 46
Figura 10 Bytecodes para manipulação de atributos 47
Figura 11 Arquitetura do processador PicoJava-I 53
Figura 12 Estrutura de bloco do processador JOP 54
Figura 13 Arquitetura do processador jHISC 56
Figura 14 Estrutura interna do JAIP-MP 59
Figura 15 Ilustração da proposta 63
Figura 16 Comportamento do CFP 67
Figura 17 Comunicação entre CFP e processador 68
Figura 18 Exemplo de *payload* de envio (do CFP ao processador) 70
Figura 19 Exemplo de *payload* de recebimento (do processador ao CFP) 70
Figura 20 Arquivo JSON criado pela ferramenta *pyc_serializer* 72
Figura 21 CFP rodando em modo de depuração 74
Figura 22 *Payload* criado para o Caso de Teste 1 78
Figura 23 Resultado para o Caso de Teste 1 79
Figura 24 Código para o Caso de Teste 2 80
Figura 25 Trecho da saída do CFP ao executar Caso de Teste 2 81
Figura 26 Carga de instruções para realizar o Caso de Teste 3 82
Figura 27 Verificação de resultados para o Caso de Teste 3 83
Figura 28 Carga de instruções para o Caso de Teste 4 84
Figura 29 *Payload* gerado para o objeto-frame 84

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programing Interface*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
FIFO	*First In, First Out*
FPGA	*field programmable gate array*
GDM	Gerenciador Dinâmico de Memória
GPU	*Graphics Processing Unit*
HDL	*Hardware Description Language*
HECO	*High Efficiency Computing*
IaaS	*Infrastructure as a Service*
JAIP-MP	*Java Application IP - Multicore*
JHisc	*Java High Level Instruction Set Computer*
JIT	*Just-in-Time Compilation*
JOP	*Java Optimized Processor*
JVM	*Java Virtual Machine*
LIFO	*Last-In-First-Out*
LOAC	*Lock Object Access Controller*
LRU	*Least Recently Used*
MMU	*Memory Management Unity*
MVP	Máquina Virtual Python
OOP	*Object-oriented Programming*
OpenGL	*Open Graphics Library*
ORC	*Object Controllers*
PaaS	*Platform as a Service*

PMM	*Python Memory Manager*
SaaS	*Software as a Service*
WCET	*Worst-Case Execution Time*

SUMÁRIO

1 INTRODUÇÃO 14
1.1 Ambiente Operacional para execução de Python 14
1.1.1 Máquina Virtual Python (MVP) 14
1.1.2 Compilação Cruzada 15
1.1.3 JIT (Tradução dinâmica) 17
1.1.4 Execução em nuvem (*Cloud*) 18
1.1.5 Suporte por GPU 19
1.2 Motivações para o desenvolvimento de um processador Python em hardware 21
1.2.1 Benefícios semelhantes obtidos com Java 22
1.2.2 Processador Python Multiciclo 23
1.2.3 Processador Python Pipeline 24
1.3 Hipótese central e objetivos para esta pesquisa 24
1.3.1 Objetivos gerais e específicos 25
2 METODOLOGIA 27
2.1 Classificação da metodologia científica 27
2.2 Etapas do método de trabalho 27
3 REFERENCIAL TEÓRICO 30
3.1 A linguagem Python 30
3.1.1 Aplicações 32
3.2 A Máquina Virtual Python (MVP) 33
3.2.1 Lógica de funcionamento de uma VM 34
3.2.2 Formato do arquivo PYC 38
3.2.3 Conjunto de instruções baseado em arquitetura de pilha 40
3.2.4 Semântica das instruções para suporte a objetos Python 43
3.2.5 Alocação de memória para objetos Python 47
3.2.6 Reciclagem de memória na MVP 48
3.3 Trabalhos Correlatos 50
3.3.1 PicoJava-I 50
3.3.2 JOP 53
3.3.3 jHISC 55
3.3.4 JAIP-MP 57
4 DESENVOLVIMENTO DO GERENCIADOR 60
4.1 Fluxo de Funcionamento do Gerenciador 60
4.1.1 Obtenção dos dados necessários à execução de programas Python 62
4.1.2 Dinâmica de comunicação com o processador 64
4.1.3 Padrões de comunicação com o processador 66
4.2 Extração de dados do PYC 69
4.3 Software CFP 73
4.4 Utilitário de teste 75
5 RESULTADOS E DISCUSSÕES 77
5.1 Descrição dos Casos de Teste 77
5.2 Execução dos Testes 78
5.2.1 Caso de Teste 1 78
5.2.2 Caso de teste 2 79
5.2.3 Caso de teste 3 81
5.2.4 Caso de Teste 4 82

5.3 Discussão 84
5.3.1 Caso de Teste 1: Atualização de Objetos-Frame 84
5.3.2 Caso de Teste 2: Aninhamento Profundo 85
5.3.3 Caso de Teste 3: Persistência e Modificação de Variáveis Globais 85
5.3.4 Caso de Teste 4: Formatação de Payloads de Envio 85
5.4 Considerações Gerais 86
5.4.1 Possíveis Melhorias Futuras 86
6 CONSIDERAÇÕES FINAIS 87
REFERÊNCIAS 89

1 INTRODUÇÃO

Python é uma linguagem de programação que se mostra cada vez mais pervasiva em diversas áreas. Deveras, consta entre as linguagens com maior demanda pela indústria (OSES, 2023), de forma a encontrar-se desde 2011 entre as 5 linguagens mais populares do mundo (BELLAPU, 2023). Sua orientação à objetos, tipagem dinâmica, portabilidade, gramática intuitiva e bibliotecas ricas garantiram à mesma lugar de destaque numa ampla gama de domínios, emergentes ou não, tais como: desenvolvimento web, análise de dados, inteligência artificial, aprendizado de máquina, automação de tarefas, jogos, Internet das Coisas, entre outros (SAABITH; FAREEZ; VINOTHRAJ, 2019). Como afirma Srinath (2017), Python é amplamente adotada nos mais diferentes setores de desenvolvimento. Cita-se por exemplo a academia, companhias de software, instituições financeiras, desde pequenas *startups* até grandes corporações.

Python, em sua forma de concepção original, é uma linguagem interpretada, de forma que sua execução acontece por meio de uma máquina virtual instalada na arquitetura hospedeira. Apesar deste modelo de funcionamento assegurar portabilidade ao código escrito, o desempenho dos programas é comprometido devido ao alto custo do processo de interpretação, como será discutido a seguir.

Este capítulo está estruturado da seguinte forma: a seção 1.1 busca discorrer sobre o ambiente operacional para a execução dos algoritmos, isto é, as diferentes formas existentes atualmente para execução dos códigos da linguagem Python, buscando apresentar tanto os aspectos positivos quanto os negativos de cada método; a seção 1.2 disserta sobre as motivações para o desenvolvimento de um processador Python em hardware, buscando elucidar os benefícios de tal processador no contexto atual e, subsequentemente, justificar o presente trabalho, que tem por objetivo desenvolver uma parte integrante de importância fundamental para este processador. Por fim, a seção 1.3 trata da hipótese central e os objetivos de pesquisa para este trabalho.

1.1 Ambiente Operacional para execução de Python

1.1.1 Máquina Virtual Python (MVP)

Interpretada, não compilada, a linguagem Python é executada através de sua própria máquina virtual. Como define o *Dictionary of Computer Science, Engineering,*

And Technology: "Máquina virtual é um processo num sistema multi-tarefa, que por sua vez se comporta como um computador independente, e não parte de um sistema maior", e complementa que tal processo é responsável por: "executar um conjunto de instruções tipicamente executado por um interpretador"(LAPLANTE, 2017, pg. 522). Portanto, uma máquina virtual é um processo que roda sobre um sistema computacional, sendo responsável por executar a sequência de bytecodes em fluxo de emulação de arquitetura física, usando, na realidade, instruções e recursos nativos da arquitetura hospedeira.

Tal funcionamento assegura portabilidade ao código escrito, uma vez que devido ao processo de execução depender do interpretador, basta que o mesmo seja suportado pela máquina hospedeira para que os bytecodes de Python possam ser executados. Embora tal recurso seja importante no contexto da indústria atual, visto que evita gastos para promover a portabilidade dos sistemas desenvolvidos sobre diferentes plataformas computacionais, o processo de interpretação afeta consideravelmente a performance (O'CONNOR; TREMBLAY, 1997). De fato, uma única instrução da linguagem interpretada comumente desdobra-se em dezenas ou centenas de instruções no processador nativo, dado o *overhead* decorrente da execução da PVM sobre a arquitetura subjacente.

Tal decréscimo de desempenho, universal às linguagens interpretadas, motivou diversas iniciativas afim de mitigar o problema. Citam-se, por exemplo, variações da máquina virtual padrão, como Mamba (PEREIRA; AYCOCK, 2002) e Falcon (BELAID et al., 2013), alternativas ao interpretador padrão, as quais procuram diminuir o custo do processo de interpretação. Ambas apresentam modificações no funcionamento dos bytecodes, além de otimizações na geração dos mesmos durante o processo de compilação. Apesar de efetivos ganhos de desempenho, devido a fatores como pouca compatibilidade com bibliotecas importantes (POWER; RUBINSTEYN, 2013), e o avanço de técnicas de compilação cruzada e *Just-In-Time* (JIT), o emprego de tais tecnologias se mostra reduzido.

1.1.2 Compilação Cruzada

Compilação é um processo que normalmente implica em produzir código para a mesma arquitetura em que a tradução está ocorrendo. Quando o objetivo é produzir código para ser executado em outro sistema, dá-se o nome de *cross-compiling* (compilação cruzada) (LAPLANTE, 2017). No contexto do Python, e de linguagens

interpretadas de modo geral, compilação cruzada refere-se à produção de código diretamente à arquitetura do sistema hospedeiro. Isto é, em vez do processo resultar em bytecodes Python destinados à MVP, ele irá gerar instruções nativas da arquitetura do processador físico disponível na máquina hospedeira, ou seja, instruções pertencentes ao seu *Instruction Set Architecture* (ISA). Apesar de, ao primeiro olhar, a estratégia de compilação cruzada apenas reduzir os custos antes produzidos pela execução da MVP, gargalo por sua vez já discutido na seção anterior, cabem diversos questionamentos acerca deste processo.

Primeiramente a qualidade da tradução, crítica para o objetivo da técnica, depende profundamente do conhecimento das características do processador de destino, num processo a fim de transcrever um código que, salienta-se, não foi pensado para execução direta nos processadores atuais. Tal peculiaridade, igualmente, compromete uma característica cara à linguagem: sua portabilidade. Compiladores cruzados (assim como JIT, conforme será visto na seção seguinte) requerem, por natureza, esforço intenso para garantir tradução de qualidade às mais diferentes plataformas de destino possíveis, o que se traduz igualmente em custo. Dessa forma, sob a égide da compilação cruzada, a máxima *write once, run anywhere* (escreva uma vez, execute em qualquer lugar) se mostra improvável, dada a grande quantidade de arquiteturas possíveis e as constantes atualizações sofridas pela linguagem. Não obstante, tais compiladores (assim como para JIT, conforme será discutido) precisam ser revisados a cada mudança tecnológica observada nas novas gerações de processadores.

Num outro argumento, afirma-se também que a flexibilidade de execução nesta técnica fica comprometida. Supondo-se um software crítico rodando num servidor, sendo tal software compilado, qualquer eventual complemento (biblioteca por exemplo), ou modificação a este código, implicaria em necessariamente interromper seu funcionamento a fim de recompilar todo software (agora com o novo recurso). Naturalmente, tal penalidade poderia ser mitigada por infraestrutura abundante, mas a questão levantada é que ocorre perda de flexibilidade de modificação do software durante sua execução, sendo passível de problemas em aplicações críticas e intensivas.

Por último, Python, assim como boa parte das linguagens interpretadas em voga, suporta orientação a objetos (OOP), sendo este de fato o principal paradigma utilizado pelos desenvolvedores (DYER; CHAUHAN, 2022). Ao realizar uma tradução de código em alto nível para código objeto nativo do processador, estima-se que a eficiência da execução deste software sobre o processador não seja ótima, uma vez que os

processadores atuais não possuem suporte direto a OOP (daí a necessidade de uma MVP). Python, diferentemente de outras linguagens compiladas, carece de suporte nativo e mais específico por parte da arquitetura nativa.

Contudo, apesar de tal ressalva, não se encontram evidências concretas na literatura que efetivamente caracterizem a qualidade da tradução como problemática, ao menos não a ponto de invalidar o processo. Entretanto, como será discutido adiante, argumenta-se que um processador nativo a OOP não apenas poderia mostrar-se mais eficiente (uma vez que suportaria diretamente o paradigma da linguagem), como já o aconteceu no contexto da linguagem Java, a qual por sua vez compartilha diversas semelhanças relevantes para com Python.

Portanto, embora compilação cruzada possa ser uma solução para casos genéricos, reforça-se para diversos casos tal técnica não é apta como solução. Por exemplo, em situações em que a alta portabilidade e migração do código são requeridos, quando a aplicação precisa ser modificada (corrigida ou otimizada por exemplo) sem parada do código e quando a arquitetura alvo é muito nova, de forma que o compilador cruzado ainda não é capaz de gerar código para ela. Por fim, salienta-se que o compilador cruzado não oferece a mesma segurança para *exploits* de código binário (técnicas que exploram vulnerabilidades específicas em programas, manipulando diretamente seu código executável em formato binário) que outros compiladores mais maturados, de outras linguagens não interpretadas.

1.1.3 JIT (Tradução dinâmica)

Na busca pela otimização de desempenho de linguagens interpretadas, aprimorou-se a técnica *Just-in-Time Compilation* (JIT). Sendo seu maior expoente no âmbito da linguagem Python o interpretador PyPy, esta técnica consiste em uma tentativa de traduzir trechos de código para a arquitetura nativa da máquina hospedeira (realizando portanto compilação cruzada), durante sua execução. Posteriormente, tais trechos são armazenados em cache, a fim de serem reutilizados sob demanda (NEWHALL, 1999). De fato, esta técnica evita a sobrecarga de compilar código desnecessário (visto que apenas os trechos executados são compilados), o que não ocorre no processo de compilação tradicional. Deveras, é bastante observado que esta técnica apresenta vantagens perceptíveis sobre as demais, de forma que hoje é uma das principais formas de otimizar a execução dos códigos da linguagem, sendo dessa forma o interpretador PyPy

uma alternativa popular ao interpretador padrão *CPython*.

Todavia, apesar dos bons resultados, tal técnica implica em determinadas limitações. Como afirma Muller et al. (1997), JIT inibe quaisquer otimizações avançadas pois a compilação acontece em tempo de execução, de forma que quaisquer gargalos no processo de tradução penalizam diretamente o tempo de execução. Reforça-se que otimizações são importantes, principalmente, em arquiteturas que contemplem a abordagem de design *Reduced Instruction Set Computer* (RISC), as quais demandam análise mais profunda a fim de se obter desempenho satisfatório.

Igualmente, enfatiza-se que tal técnica faz uso de compilação cruzada, e portanto, todas questões levantadas na seção anterior se estendem a esta técnica. Não obstante, Schoeberl (2008) mostra que, apesar dos bons resultados, JIT demanda quantidades expressivas de memória, inclusive cache, além da necessidade de um compilador ativo no sistema hospedeiro (SCHOEBERL, 2008), características que a tornam inadequada para quaisquer sistemas limitados, a exemplo de sistemas embarcados e de sistemas de tempo real. Ademais, reforça-se que, assim como para técnicas de compilação cruzada, não há garantia de que a tradução apresente boa qualidade.

1.1.4 Execução em nuvem (*Cloud*)

Outra forma bastante popular atualmente de executar código Python é através da nuvem. Computação em nuvem refere-se ao fornecimento de recursos de TI por meio da internet, sob demanda (SURBIRYALA; RONG, 2019). Dessa forma, indivíduos e organizações não mais precisam comprar infraestrutura e recursos para usufruí-los, uma vez que podem solicitá-los através da internet e pagar somente de acordo com o uso. Algumas de suas vantagens são a escalabilidade, ou seja, a capacidade do sistema de se adaptar à demanda, e a agilidade, uma vez que infraestrutura e serviços podem ser adquiridos imediatamente através da Internet. Tal modalidade surgiu ao final dos anos 1990 e ganhou tração em meados dos anos 2000, inaugurando por sua vez uma área inovadora na indústria, estando diretamente envolvida na ascensão de empresas como Amazon Web Services, IBM, Google, entre outras. (SURBIRYALA; RONG, 2019).

Os recursos obtidos através da nuvem desdobram-se em diversos formatos. Os mais conhecidos são Software como Serviço (SaaS), Plataforma como Serviço (PaaS) e Infraestrutura como Serviço (IaaS). Desses, PaaS se mostra uma alternativa para execução de código Python. Plataforma como serviço, nesse contexto, significa que a infraestrutura

necessária à execução de determinado código é fornecida pelo provedor de nuvem. Dessa forma, o usuário apenas fornece o código, e o serviço de nuvem se encarregará de sua execução. Portanto, irá prover memória, tempo de processador e quaisquer dependências requisitadas pelo usuário (SURBIRYALA; RONG, 2019).

Dessa forma, um usuário que deseje executar código Python através da nuvem precisa apenas de um login de acesso na plataforma do provedor, fornecendo o código da aplicação a ser executada. Portanto, é necessário apenas um navegador, não havendo necessidade de preocupar-se com instalação e configuração da MVP. Esta forma de execução se mostra bastante prática, uma vez que é possível executar qualquer aplicação imediatamente, independente de sua demanda de hardware e infraestrutura em geral (já que os provedores fornecem recursos virtualmente ilimitados). Naturalmente, tal método não é adequado a todos os casos.

Embora seja uma opção interessante para vários casos, é natural que tal método implique em custo mais elevado para diversas situações. Como foi afirmado, no contexto da computação em nuvem, paga-se pelo uso. Dessa forma, a mesma pode não ser uma opção viável para aplicações que devam executar ininterruptamente (como servidores) ou que demandem processamento intensivo e extenso (como simulações). Ambos casos podem não ser vantajosos, uma vez que normalmente os serviços cobram pelo tempo de uso e quantidade de processamento necessária. Entretanto, reforça-se que para cargas extremas de serviço, a nuvem podem sim ser a melhor opção, dada que fornece infraestrutura ilimitada, mas naturalmente tal questão depende de cada caso.

Outro ponto em que a nuvem pode não ser adequada é em casos de serviços que exijam baixa latência. Isto porque este método requer necessariamente uma conexão eficiente e estável com a Internet (portanto fica sujeito à qualidade da conexão), o que pode ser um impeditivo em determinados casos. Seguindo nesta linha, aplicações interativas que envolvam processamentos complexos ficam igualmente sujeitas à qualidade da conexão disponível. Não obstante, a plataforma que de execução do servidor possivelmente roda uma PVM, de forma que sofre das mesmas questões de desempenho discutidas nas seções anteriores.

1.1.5 Suporte por GPU

Das alternativas à MVP citadas até aqui, pode-ser afirmar que o método de aceleração mais utilizado é por meio do uso de unidades de processamento gráfico

(*Graphics Processing Units* - GPUs). De fato, encontra-se na literatura uma pletora de bibliotecas para os mais diversos fins. Sendo a linguagem Python popular para aplicações científicas, dadas as ricas APIs disponíveis para a mesma, tais aceleradores visam otimizar o desempenho deste nicho específico. Estes buscam, por exemplo, otimizar o desempenho de simuladores de fenômenos físicos, como dinâmica molecular (SCHOENHOLZ; CUBUK, 2019) e ciência dos materiais (BROUGH; WHEELER; KALIDINDI, 2017), computação de séries de Fourier (BEZZAM et al., 2022), modelos de *machine learning* (MAIER et al., 2020), *deep learning* (DOGARU; DOGARU, 2017) e etc. Reforça-se, porém, que tais tecnologias visam otimização de nichos específicos da linguagem, e não do desempenho da mesma como um todo.

Apesar de seu amplo uso, o emprego de GPU não constitui uma panaceia para as questões de desempenho das linguagens interpretadas. A seguir, enumera-se alguns pontos sobre:

- Complexidade do código: devido a sua natureza paralela, código para GPU requer uma compreensão profunda de programação concorrente e paralela, o que costuma aumentar a complexidade do código e introduzir possíveis erros.
- *Overhead* de transferência de dados: transferir dados entre a CPU e a GPU pode ser um gargalo de desempenho significativo. Se a quantidade de dados transferidos entre a CPU e a GPU for grande em comparação com o tempo de cálculo real na GPU, isso pode anular os benefícios da aceleração da GPU (FERREIRA; MAGALHÃES; NACIF, 2019).
- Compatibilidade de hardware: nem todas as GPUs suportam todas as operações ou versões de bibliotecas e APIs. Isso pode levar a problemas de compatibilidade entre diferentes hardware e software, tornando a implementação e manutenção mais desafiadoras.
- Consumo de energia: GPU geralmente consomem mais energia do que CPUs (MITTAL; VETTER, 2014), o que pode ser um problema em sistemas com recursos limitados, como dispositivos móveis ou sistemas embarcados.
- Sobrecarga de paralelismo: nem todas as tarefas se beneficiam de paralelismo em GPU. Algumas tarefas são intrinsecamente sequenciais e não podem ser aceleradas significativamente em uma GPU.
- Custo: GPU dedicadas de alto desempenho podem ser caras. Além disso, a implementação de software otimizado para GPU pode exigir investimentos

significativos em termos de desenvolvimento e treinamento, uma vez que requerem uso de linguagens pouco familiares como CUDA ou OpenGL.

1.2 Motivações para o desenvolvimento de um processador Python em hardware

Embora eficaz para executar aplicações convencionais, diversas tecnologias intrinsecamente envolvidas com a linguagem, como redes neurais, *deep learning*, mineração de dados, robótica, entre outras, demandam um poder de processamento cada vez mais expressivo. Neste contexto, sua natureza interpretada não se mostra uma alternativa ótima para atender tais demandas (NEWHALL, 1999). Apesar dos métodos existentes já citados acudirem de alguma forma a demanda por otimização dos códigos da linguagem, mostrou-se que todos apresentam determinadas desvantagens, as quais atuam como motivação para este trabalho.

Apesar dos vastos esforços no sentido de otimizar o desempenho do processo de execução de bytecodes, poucas iniciativas são encontradas no âmbito do hardware. Neste domínio, encontra-se poucos projetos de aceleração para Python, ausência que contrasta com as diversas soluções desenvolvidas para a linguagem Java, em especial o desenvolvimento processadores nativos para sua respectiva máquina virtual. Assim, até onde sabemos, não há processador nativo Python.

Desenvolver uma máquina física diretamente compatível com a MVP em *Hardware Description Language* (HDL) é uma iniciativa interessante uma vez que a sobrecarga do processo de emulação dos bytecodes Python seria eliminada, assim como a sobrecarga do processo de tradução em compiladores JIT, e de sua excessiva demanda por recursos. Como argumenta O'connor e Tremblay (1997, pg. 1): "Ao executar diretamente os bytecodes, um processador poderia combinar as vantagens da interpretação e compilação JIT, enquanto que elimina suas desvantagens". Entretanto, apesar dos potenciais benefícios de tal iniciativa, não se encontram tais *softcores* (arquiteturas descritas em HDL) de processadores Python funcionais na literatura [1].

[1] Com exceção da máquina desenvolvida por BITENCOURT (2019), que foi desenvolvida no mesmo grupo deste trabalho, conforme será discutido.

1.2.1 Benefícios semelhantes obtidos com Java

Como mencionado, diversas soluções em hardware foram desenvolvidas para otimizar o desempenho na execução de programas escritos em Linguagem Java. Esta linguagem, por sua vez, também é interpretada, e portanto é necessário também um interpretador ativo num sistema computacional para executar suas instruções. Dessa forma, sofre das mesmas penalidades de desempenho já mencionadas para o Python. Tais penalidades são bem conhecidas e motivaram o desenvolvimento de diversas alternativas à *Java Virtual Machine* (JVM), de forma que uma das soluções propostas foi o desenvolvimento de um processador nativo aos bytecodes dessa linguagem. Os bons resultados obtidos, discutidos adiante, inspiram e motivam a contribuição a ser dada pelo presente trabalho de desenvolvimento do processador nativo para Python. A seguir, enumeram-se os principais trabalhos nesse sentido, expondo de forma simplificada os resultados informados por cada autor:

- PicoJava-I (O'CONNOR; TREMBLAY, 1997): um dos primeiros a ser publicado, é capaz de executar todos os bytecodes Java, e dispõe de suporte para *multithreading*, coletor de lixo, entre outros recursos. Assim como os demais, apresenta arquitetura otimizada para a orientação à pilha [2], característica das linguagens Java e Python. Segundo o autor, apresentou um ganho de velocidade 5x maior em relação a um Intel Pentium, com compilador JIT.
- JOP (SCHOEBERL, 2003): consiste em um processador de sistema de tempo real (em que o tempo de processamento das tarefas é pré-definido) que realiza uma tradução dos bytecodes Java para um conjunto próprio de instruções, chamado de *microcode*, sendo tal conjunto de instruções definido pelos autores como uma versão RISC dos bytecodes da JVM. Apesar de não haverem sido utilizados *benchmarks* para comparação direta de performances, os dados fornecidos pelo autor indicam ganho considerável de eficiência em contraste com a máquina virtual.
- JHisc (YIYU et al., 2006): voltado para sistemas embarcados, esta arquitetura foi projetada para realizar operações de tempo real e baixo consumo energético. Segundo o autor, 94% dos bytecodes são implementados diretamente em hardware. A fim de mensurar o desempenho, o autor informa um ganho de velocidade de 30%

[2] Uma arquitetura de processador orientada à pilha é um tipo de design de CPU em que as operações são realizadas empilhando e desempilhando dados em uma estrutura de pilha. Nesse tipo de arquitetura, os dados são armazenados e recuperados da pilha, que é uma região de memória dedicada para esse fim.

sobre o *PicoJava I* e 183% em relação do *JOP*.

- JAIP-MP (TSAI; WU; SU, 2015): uma das primeiras arquiteturas desse tipo genuinamente *multicore* (abrange 4 núcleos), é otimizada para tarefas paralelas, seguindo o padrão de *threads* nativo da linguagem Java. Igualmente apresenta ganhos expressivos de desempenho em relação à uma execução convencional (por máquina virtual) dos programas testados pelo autor.

Uma vez introduzidos tais trabalhos correlatos, argumenta-se que resultados semelhantes possam vir a ser obtidos para programas Python, através de um processador nativo. Não apenas como nenhum impedimento para tal é encontrado na literatura, como uma versão multiciclo rudimentar (discutida adiante) já existe e mostra-se funcional. Ademais, dado o vasto uso da linguagem pela indústria, e sua importância para as tecnologias já mencionadas, argumenta-se que o desenvolvimento de um processador Python nativo mostra-se relevante, uma vez que possibilita conciliar as desvantagens dos métodos de execução disponíveis atualmente. Doravante, conjectura-se que os benefícios de um hardware específico poderiam potencializar o alcance da linguagem para áreas em que no momento encontra-se não satisfatoriamente cobertas pela infraestrutura em uso disponível para execução de Python, como sistemas embarcados, sistemas de tempo real e servidores.

1.2.2 Processador Python Multiciclo

Dada a complexidade de desenvolvimento de um processador, tal tarefa não poderia ser contemplada num único trabalho. De fato, um protótipo funcional foi desenvolvido por BITENCOURT (2019), e é usado como ponto de partida para este trabalho. Esta arquitetura consiste numa implementação multiciclo, ou seja, as instruções são executadas em múltiplos ciclos de *clock*, em vez de serem concluídas em um único ciclo. Tal arquitetura é capaz de executar instruções básicas da linguagem, como as de aritmética e manipulação de variáveis simples. Entretanto, o código a ser executado deve ser convertido a um formato específico antes de ser aceito pela arquitetura.

Apesar de suportar algoritmos elementares, o processador se mostra insuficiente para execução de algoritmos reais, que por sua vez envolvam matrizes, vetores, e, principalmente, objetos (construção fundamental na linguagem). Dessa forma, a arquitetura encontra-se em formato de protótipo, e carece de melhorias a fim de poder

ser empregada em contextos reais. Este trabalho, por sua vez, tem como um dos objetivos complementar a mesma, a fim de habilitá-la à manipulação de objetos.

1.2.3 Processador Python Pipeline

Entre as melhorias previstas, outra evolução encontra-se em andamento paralelamente ao desenvolvimento deste presente trabalho. Uma versão *pipeline* do processador está sendo atualmente concebida no contexto do grupo de pesquisa HECO (*High Efficiency Computing Group*), estando registrado sob o nº 2023.PE.BG.2067 na Pró-Reitoria de Pesquisa e Inovação da UNIPAMPA. O objetivo desta versão é habilitar o processador ao *pipeline* de instruções, a fim de que o desempenho para a execução dos bytecodes de Python seja mais competitivo com o das arquiteturas de processadores contemporâneos, que dispõem de *pipeline* de instrução. A conclusão do núcleo básico do processador é prevista até o final do primeiro semestre de 2025.

Neste contexto, este trabalho está alinhado ao desenvolvimento do *pipeline*, e trabalha-se para que o Gerenciador de Contexto de Memória seja diretamente compatível com tal processador. Assim, espera-se que através do Gerenciador aqui desenvolvido o processador seja capaz de suportar as trocas de contexto de memória (ou seja, chamadas e retornos de funções), fundamentais em programas Python. Não obstante, é prevista compatibilidade direta com o arquivo interpretado ("PYC") gerado pelo interpretador, alcançando assim suporte abrangente da linguagem.

1.3 Hipótese central e objetivos para esta pesquisa

A hipótese central para este trabalho é que há significativos benefícios a serem proporcionados pelo desenvolvimento de um Gerenciador de Contexto de Memória (GCM) para a arquitetura Python nativa. Tal afirmação se baseia na ideia de que o gerenciamento da pilha de execução Python é uma tarefa complexa e computacionalmente custosa que pode ser delegada a um coprocessador, tal como nas arquiteturas Java e nos processadores modernos, que dispõem de Unidades de Gerenciamento de Memória (MMU), as quais não apenas aprimoram o desempenho, como contribuem para a proteção dos dados do usuário. Ao transferir essa responsabilidade, o processador principal fica livre para executar os demais bytecodes da aplicação alvo. Portanto, um gerenciador de

contexto de memória é importante no sentido de auxiliar o processador desenvolvido a alcançar um desempenho superior.

O sistema, por sua vez, será ligado ao processador principal através de uma interface e estará ao serviço deste, intermediando a gestão do contexto de execução e a troca de objetos-frame durante a execução de funções. Embora o foco deste trabalho seja o Gerenciador de Contexto, espera-se que, no futuro, o sistema possa ser expandido para incluir funcionalidades como controle completo do fluxo de execução, gestão da pilha de objetos-frame e manipulação de variáveis globais de forma integrada. Dessa forma, o trabalho estabelece uma base sólida para o desenvolvimento de um processador dedicado à execução de bytecode Python, que possa gerenciar objetos de maneira eficiente e confiável, aproximando-o de seu objetivo final.

Salienta-se que o software desenvolvido neste trabalho não é restrito a um protótipo específico de processador Python. Como discutido, apenas protótipos de processadores Python nativos existem atualmente, e embora este projeto busque ser acoplado à versão pipeline discutida na seção anterior, o intuito é que ele seja adaptável a qualquer processador Python nativo. Portanto, o "processador" será referido como uma arquitetura hipotética, reforçando o objetivo do gerenciador de ser compatível com qualquer processador Python nativo. Para isso, a interface entre o gerenciador e o processador hipotético é bastante genérica, não abrangendo qualquer detalhe específico dos protótipos de processadores mencionados.

1.3.1 Objetivos gerais e específicos

- Objetivo geral: Desenvolver um gerenciador de contexto de memória para um processador Python nativo.
- Objetivos específicos:
    - Aprimoramento das técnicas para projeto de gerenciadores de contexto de memória.
    - Estudo avançado do projeto de processadores e software básico de sistemas computacionais.
    - Experiência no desenvolvimento de um sistema computacional completo em âmbito acadêmico dentro do curso de Engenharia de Computação da UNIPAMPA.

Esta monografia está organizada em cinco capítulos que abordam os principais aspectos do desenvolvimento deste trabalho. O Capítulo 2 apresenta a metodologia utilizada, descrevendo o tipo de pesquisa empregado e as etapas realizadas para o desenvolvimento do Gerenciador de Contexto de Memória. No Capítulo 3, é explorado o referencial teórico, com uma visão geral sobre a linguagem Python, sua Virtual Machine e os trabalhos correlatos que embasaram a solução proposta. O Capítulo 4 detalha o processo de desenvolvimento da solução, incluindo as decisões técnicas, os desafios enfrentados e as estratégias adotadas para implementar o sistema. No Capítulo 5, são apresentados os resultados obtidos e discutidas as implicações desses achados, analisando o funcionamento do Gerenciador em diferentes cenários. Por fim, o Capítulo 6 conclui o trabalho, destacando as principais contribuições, limitações e possíveis direções para futuras evoluções deste projeto.

2 METODOLOGIA

Dada a natureza metódica de um trabalho científico, procura-se aqui expor a metodologia utilizada para o desenvolvimento desta monografia. Assim, a seção 2.1 busca estabelecer as classificações de metodologias científicas as quais este trabalho se encaixa. Após, buscando detalhar o processo de desenvolvimento, expõe-se as etapas definidas para a concretização deste trabalho.

2.1 Classificação da metodologia científica

Dada a característica do tema abordado, entende-se que este trabalho se enquadra nas diferentes classificações:

- Pesquisa aplicada: uma vez que possui objetivo prático, sendo por sua vez a implementação do coprocessador.
- Quali-Quanti: devido a ser uma pesquisa mista, no sentido de envolver métodos qualitativos e quantitativos, a fim de obter uma compreensão mais abrangente do tópico pesquisado.
- Exploratória: dada a necessidade inicial de investigação ampla do tópico de pesquisa, sem necessidade, num primeiro momento, de testar hipóteses específicas.
- Experimental: uma vez que irá requerer implementação e avaliação experimental (empírica) de uma solução para coprocessamento.
- Bibliográfica: considerando que envolve larga análise e síntese de informações disponíveis em fontes bibliográficas, como livros, artigos acadêmicos, teses e outras publicações.

2.2 Etapas do método de trabalho

A seguir, elenca-se as etapas definidas para conclusão do objetivo geral deste trabalho, comentando as etapas concluídas até o momento:

- Etapa 1: **leitura dos documentos preliminares sobre arquitetura da MVP e semântica do seu conjunto de instruções**. De fato se encontrou bastante conteúdo disponível, dada a popularidade da linguagem e sua natureza *open-source*. Desta

forma, diversos livros e artigos foram encontrados, sendo o mais completo o livro *Inside the Python Virtual Machine*, de Ike-Nwosu (2018), que se tornou a principal referência acerca da arquitetura da MVP neste trabalho.

- Etapa 2: **experimentação com o *decompiler* Python para compreender melhor como se dá a interação entre os bytecodes para executar as operações descritas pelo programador Python em linguagem de alto nível**. Esta etapa é importante no sentido de assimilar a construção de um programa Python em nível de bytecodes. Assim, foi possível inclusive estimar quais as instruções mais frequentes, assim como compreender quais instruções são responsáveis pela troca de contexto de objetos-código (como será visto no capítulo 3), e quais são responsáveis pela manipulação da memória.
- Etapa 3: **levantamento e leitura preliminar de material bibliográfico relacionado ao escopo deste trabalho (busca no Google Acadêmico, ACM Portal, IEEE xplore, Citeseer, entre outros)**. Aqui se deu início à pesquisa bibliográfica necessária ao embasamento das ideias expostas neste trabalho. Portanto, buscou-se referências acerca dos principais assuntos que norteiam o trabalho. Informações sobre gerenciamento de memória em linguagens orientadas a objetos, linguagem Python de forma mais específica e trabalhos correlatos foram definidas nesta etapa.
- Etapa 4: **leitura dos documentos levantados e selecionados na Etapa 1, buscando identificar aspectos e informações que balizem ou inspirem para o desenvolvimento de um suporte em hardware para gerenciamento de objetos Python**. Aqui foi necessário condensar os conhecimentos adquiridos a partir da primeira etapa, a fim de esboçar estratégias para o desenvolvimento do coprocessador. Foi dada atenção especial ao gerenciador de memória da MVP, dada sua pertinência para este trabalho.
- Etapa 5: **escrita dos capítulos iniciais da monografia, seguindo estrutura específica**. Nesta etapa iniciou-se efetivamente a escrita. Antes de começar definiu-se a estrutura dos capítulos e seções, a fim de se obter uma sequência lógica de informações.
- Etapa 6: **Desenvolvimento das rotinas de gerenciamento de contexto de memória, buscando cobrir as instruções de gerenciamento geradas pelo compilador**. Esta etapa consiste na estruturação geral das rotinas que deverão atender às demandas do processador. Desta forma, definiu-se quais instruções

devem ser atendidas pelo gerenciador, e então se desenvolve o código em C++ responsável pela sua execução.

- Etapa 7: **desenvolvimento e seleção de programas para teste do gerenciador com conjunto de instruções expandido.**
- Etapa 8: **testes e depuração do gerenciador frente a instruções de gerenciamento de contexto.**
- Etapa 9: **finalização dos capítulos da monografia.**

3 REFERENCIAL TEÓRICO

Este capítulo está organizado da seguinte forma: a seção 3.1 desenvolve a questão acerca do gerenciamento de memória. Assim, busca num primeiro momento contextualizá-la de forma a expor seu surgimento e problemática, para então discorrer sobre os principais aspectos acerca do tópico: hierarquia clássica de memória, gerenciamento dinâmico e estático, memória contígua e paginada, e finaliza por expor técnicas importantes de coleta de lixo. Após, a seção 3.2 discorre sobre a linguagem Python como um todo, desde seu surgimento a suas características, popularidades e aplicações. Na seção 3.3, expõe-se o funcionamento da MVP, dissertando desde o funcionamento geral até o detalhamento acerca da manipulação de memória pela mesma. Por fim, a seção 3.4 apresenta os trabalhos correlatos a este, e desenvolve acerca do suporte para Gerenciador de Contexto de Memória (GCM) nos processadores nativos para linguagens orientadas a objetos.

3.1 A linguagem Python

A linguagem de programação Python foi concebida ao longo dos 1980, de forma que seu desenvolvimento iniciou em dezembro de 1989 (TULCHAK; MARCHUK, 2016). Criada por Guido Van Rossum nos Países Baixos, foi pensada como uma evolução da linguagem de programação ABC, a qual tinha fins educacionais e se mostrava popular à época. Desde o início, foi elaborada a fim de priorizar código limpo e permitir que programadores implementem ideias em menos linhas de código que o necessário, em comparação a outras linguagens. Para tal, definiu-se que sua sintaxe dispensaria diversos elementos, quase onipresentes noutras linguagens, tais como ponto e vírgula ao final de declarações e atribuições de variáveis, uso de colchetes para delimitar blocos particulares de código, como `if`, `else`, `for`, etc. utilizando em vez disso indentação para demarcar tais blocos. Dispensa igualmente a palavra-chave `new` ao instanciar novos objetos, além de utilizar tipagem dinâmica, a qual desobriga o programador a declarar o tipo das variáveis utilizadas, igualmente permitindo alterar o tipo de dado o qual elas referenciam.

Apresentando sólida evolução, recebe incrementos constantes de forma a suportar cada vez mais funções. Atualmente, é uma linguagem rica em recursos, que atende demandas de diversas áreas da indústria de software. Cita-se, por exemplo, o suporte

a diversos paradigmas de programação, como orientação a objetos, funcional, estruturado e imperativo; a tipagem dinâmica, celebrada pela comunidade, e cada vez mais comum em linguagens recentes como Elixir, Clojure e Hack; apresenta eficiente gerenciamento dinâmico de memória, o que permite uso ótimo da mesma sem a necessidade de intervenção do programador. Tal recurso é importante e, juntamente com tipagem dinâmica, mostra-se cada vez mais comum em novas linguagens, dado o expressivo aumento de produtividade por parte da equipe de desenvolvimento. Outra característica altamente relevante é a sua rica biblioteca. Considerada pela comunidade uma das características mais importantes da linguagem (TULCHAK; MARCHUK, 2016), a considerável quantidade de ferramentas contidas na biblioteca foi e ainda é primordial na sua ampla adoção. Tal característica é conhecida como "*batteries included*", e faz parte da filosofia da linguagem desde o começo.

Destaca-se igualmente sua ampla popularidade, tal que, a exemplo, Srinath (2017) cita pelo menos quatro indicadores de popularidade a fim de atestar o amplo uso da linguagem, sendo eles: 5ª maior comunidade no StackOverflow [1], contando com 85.900 seguidores, e mais de 500.000 perguntas relacionadas à linguagem. 4ª linguagem mais utilizada no GitHub [2] e a 3ª maior comunidade no Meetup Community [3]. A outra métrica utilizada pelo autor é a quantidade de ofertas de vagas para desenvolvedores da mesma: em 2017, Python era a 2ª linguagem mais requisitada pelo mercado, e, devido a fatores como a ascensão do *Big Data* na época, apresentava a maior média salarial entre as fontes pesquisadas pelo autor. Não obstante, Srinath (2017) argumenta que Python é a linguagem com crescimento mais rápido do mercado.

A linguagem Python é interpretada, não compilada (IKE-NWOSU, 2018). Isto significa que o código escrito em alto nível nesta linguagem não é compilado (traduzido) para o conjunto de instruções (ISA) nativo da máquina hospedeira. Em vez disso, tal código é convertido para os bytecodes específicos da linguagem, os quais constituem uma ISA própria. Para executar tais bytecodes, necessita-se de uma Máquina Virtual Python (MVP), a qual tem a responsabilidade de concretizar as instruções fornecidas em bytecode através da ISA e do sistema operacional nativos da máquina hospedeira.

A decisão pelo processo de interpretação em detrimento ao de compilação assegura benefícios importantes à linguagem, com destaque para sua portabilidade. Uma

[1] Plataforma online popular de perguntas e respostas focada no desenvolvimento de software.
[2] Plataforma popular de hospedagem e colaboração para desenvolvimento de software baseado em controle de versão Git.
[3] Plataforma de mídia social a fim de organizar eventos presenciais.

vez que a execução efetiva dos bytecodes fica a cargo da máquina virtual, basta que a MVP seja suportada por determinada arquitetura. Tal princípio é conhecido como *write once, run everywhere* (escreva uma vez, execute em qualquer lugar), expressão cunhada no advento da linguagem Java, mas que hoje se estende às linguagens multiplataformas. Efetivamente, a portabilidade é um dos destaques da linguagem (SAABITH; FAREEZ; VINOTHRAJ, 2019), sendo um recurso importante na indústria, uma vez que permite a equipes de desenvolvimento reduzirem seus custos com a mesma. Entretanto, como será discutido posteriormente, do processo de interpretação decorrem penalidades de memória e desempenho as quais restringem a eficiência dos sistemas computacionais envolvidos, e limitam, se não coíbem, o alcance da linguagem a determinados nichos.

3.1.1 Aplicações

Em virtude das características de Python mencionadas anteriormente, mostra-se pertinente reforçar a ampla utilização da linguagem nos mais diferentes meios. Como afirma Saabith, Fareez e Vinothraj (2019), Python é largamente utilizado na academia, companhias de software, grandes corporações e instituições financeiras, e, por ser uma linguagem de propósito geral, adapta-se aos mais diversos domínios da área de desenvolvimento de software. Também de acordo com a visão de Saabith, "Python como um todo pode ser utilizado em qualquer esfera de desenvolvimento", tendo este autor apresentado uma breve enumeração dos principais domínios onde a Linguagem Python vem sendo aplicada:

- Aplicações *Desktop* baseadas em interface gráfica: através de sua arquitetura modular e interoperabilidade para com diversos sistemas operacionais, Python se mostra um recurso pertinente para o desenvolvimento de tais aplicações, uma vez que apresenta bibliotecas sólidas para desenvolvimento de interfaces gráficas.
- Processamento de imagens e design gráfico: a linguagem é parte, em diferentes proporções, de importantes *softwares* de processamento gráfico encontrados na indústria. Cita-se Inkscape, GIMP, Blender, 3ds Max, Cinema 4D, Maya e Houdini.
- Aplicações científicas e numéricas: Python provê alta disponibilidade e produtividade para tal nicho através de importantes bibliotecas como *Scientific Pyhon* (SciPy), *Numerical Python* (NumPy) e *Panel Data* (Pandas). Não obstante, tais bibliotecas estão profundamente envolvidas na ascensão recente da inteligência

artificial, área emergente cada vez mais relevante na indústria e na sociedade.

- Indústria de jogos eletrônicos: diversos jogos "AAA" (*video-games* conhecidos por serem de alta qualidade e alto orçamento), empregam Python em seu desenvolvimento, sendo citados pelo autor: Civilization-IV, Disney's Toontown Online, Vega Strike, etc.
- Desenvolvimento de software em geral: dotada de estrutura plena para atender as demandas da indústria, a linguagem fornece recursos para integração com bancos de dados, interface gráfica, testes, depuração, *profiling*, módulos para criptografia, redes, expressões regulares, interação com arquivos e pastas do sistema operacional nativo, para além de uma miríade de recursos à disposição da equipe de desenvolvimento.
- Influência em outras linguagens: por fim, cita-se a direta influência de Python em linguagens mais recentes, como Swift, CoffeScript, Cobra e OCaml.

3.2 A Máquina Virtual Python (MVP)

Como afirma Craig (2005), existem basicamente duas formas de implementar uma linguagem de programação: compilação ou interpretação. A compilação consiste essencialmente em converter o código escrito em código de máquina da arquitetura destino. Compiladores normalmente são construídos tendo apenas uma arquitetura destino em mente, embora naturalmente hajam exceções, como o GNU C Compiler que permite compilar para diversos sistemas. De fato, esta etapa se desdobra em outras como: pré-processamento, compilação, montagem, vinculação, etc. (STALLINGS, 2010). O resultado é um arquivo executável específico para a arquitetura hospedeira, e que executa diretamente através das instruções nativas do processador.

Já o processo de interpretação consiste na execução imediata do código-fonte por um interpretador. Comumente, tal interpretador executa as instruções uma por uma até o final do programa. Tal processo também desdobra-se em diversas fases, executadas para cada instrução do programa: análise léxica, sintática, semântica e etc. (IKE-NWOSU, 2018). Segundo Craig (2005), uma máquina virtual encontra-se no intermédio entre compilação e interpretação, uma vez que consiste num processo de "compilação" (conversão de um código em alto nível para um de baixo nível) direcionado a uma arquitetura bastante específica: a especificada pela linguagem. Ou seja, o resultado

do processo de compilação não mais é direcionado a uma arquitetura *física*, existente num determinado processador ou sistema computacional. Tal resultado agora é destinado ao software capaz de executar o conjunto de instruções estabelecido pela linguagem, denominado Máquina Virtual. Ainda de acordo com Craig (2005), máquinas virtuais são um método eficiente de assegurar portabilidade, uma vez que basta que a arquitetura subjacente suporte uma MVP para que o código seja executado de forma idônea.

Como introduzido previamente, a MVP consiste numa máquina virtual, que por sua vez é um processo num sistema operacional hospedeiro, e que tem por finalidade executar os bytecodes Python na máquina em que se encontra (IKE-NWOSU, 2018). Portanto, trata-se essencialmente de um ambiente que permite a execução de código Python em diferentes sistemas operacionais. Ela atua como uma camada intermediária entre o código-fonte Python e o hardware do computador. Quando um código Python é fornecido como entrada para a MVP, o código é primeiro interpretado para bytecodes pela mesma. Como será discutido, esse bytecodes é uma representação de baixo nível do código Python original e é independente da arquitetura do hardware. Em seguida, a máquina virtual Python interpreta e executa esse bytecode.

O termo MVP não se refere necessariamente a uma implementação específica. Na verdade, existem diversas implementações, tais como as introduzidas no capítulo 1: Mamba, Falcon e, principalmente, PyPy. Embora estas alternativas apresentem variações na forma como executam as instruções, todas possuem comportamento similar e suportam necessariamente o mesmo conjunto de bytecodes da linguagem. Dentre as diversas variações, a principal é conhecida como CPython. Esta por sua vez é a versão desenvolvida e mantida pelo criador da linguagem, e é considerada a implementação de referência atualmente (IKE-NWOSU, 2018), sendo, portanto, ela apresentada de forma mais aprofundada nesta seção.

3.2.1 Lógica de funcionamento de uma VM

Antes de efetivamente adentrar no funcionamento da MVP, é fundamental discorrer acerca de duas estruturas de dados: objeto-código e objeto-frame. Tais estruturas são fundamentais na linguagem, e toda execução de código Python acontece a partir delas (IKE-NWOSU, 2018). Objeto-código, por sua vez, é uma estrutura gerada no momento da compilação, e é inserida pelo interpretador no arquivo PYC (conforme será visto na seção 3.2.2). O arquivo PYC, por sua vez, contém todas as instruções e dados necessários

à execução do que a documentação da linguagem chama de *bloco de código* (ROSSUM; DRAKE, 2009). Um bloco de código consiste em quaisquer trechos de código que sejam executados como uma unidade. Dessa forma, explicita-se: "os seguintes são blocos: um módulo, o corpo de uma função e uma definição de classe"(IKE-NWOSU, 2018, pg. 62). Deveras, a documentação é taxativa, e afirma que um programa Python é literalmente constituído de blocos de código (IKE-NWOSU, 2018, pg. 62).

Dessa forma, todo *script* (arquivo que contém código Python e que não necessariamente faz parte de uma aplicação maior), função, método e objeto, possui um objeto-código próprio. Esta estrutura contém as instruções (bytecodes) e os dados (nomes de variáveis e constantes) pertencentes ao bloco específico e que são necessários a sua execução. De fato, é a partir do objeto-código que se dá a execução do código. A Figura 1 ilustra o código da função `FizzBuzz`, uma função simples que implementa a lógica do jogo FizzBuzz. Ela retorna "FizzBuzz" se `n` for divisível por 3 e 5, "Fizz" se divisível apenas por 3, "Buzz" se divisível apenas por 5, e converte `n` em uma *string* se não atender a nenhuma das condições anteriores. A Figura 2, contém o objeto-código gerado para esta função.

Figura 1 – Código da função `FizzBuzz`

```
1    def fizzbuzz(n):
2        if n % 3 == 0 and n % 5 == 0:
3            return 'FizzBuzz'
4        elif n % 3 == 0:
5            return 'Fizz'
6        elif n % 5 == 0:
7            return 'Buzz'
8        else:
9            return str(n)
```

Fonte: Ike-Nwosu (2018)

É possível observar na Figura 2 diversos campos. A maioria de fato é relativo a questões organizacionais da MVP. De modo geral, os campos mais relevantes à execução do código são os seguintes:

- *co_code*: contém efetivamente os bytecodes produzidos no processo de interpretação. Este campo contém a sequência de *bytes* que constituem as instruções e os respectivos operandos (que por sua vez serão detalhados na seção 3.2.3) de um bloco de código.

Figura 2 – Objeto-código gerado para a função `FizzBuzz`

```
    co_argcount = 1
    co_cellvars = ()
    co_code = b'|\x00d\x01\x16\x00d\x02k\x02r\x1e|\x00d\x03\x16\x00d\x02k\x02r\x1ed\\
x04S\x00n,|\x00d\x01\x16\x00d\x02k\x02r0d\x05S\x00n\x1a|\x00d\x03\x16\x00d\x02k\x02r\
Bd\x06S\x00n\x08t\x00|\x00\x83\x01S\x00d\x00S\x00'
    co_consts = (None, 3, 0, 5, 'FizzBuzz', 'Fizz', 'Buzz')
    co_filename = /Users/c4obi/projects/python_source/cpython/fizzbuzz.py
    co_firstlineno = 6
    co_flags = 67
    co_freevars = ()
    co_kwonlyargcount = 0
    co_lnotab = b'\x00\x01\x18\x01\x06\x01\x0c\x01\x06\x01\x0c\x01\x06\x02'
    co_name = fizzbuzz
    co_names = ('str',)
    co_nlocals = 1
    co_stacksize = 2
    co_varnames = ('n',)
```

Fonte: Ike-Nwosu (2018)

- *co_varnames*: consiste na lista de nomes de variáveis locais do bloco de código. Quaisquer nomes de variáveis definidas dentro de um bloco de código estão contidas aqui. Sempre que uma instrução manipula uma variável local, o identificador da variável está contido nesta lista.
- *co_names*: é a lista que contém os nomes, ou identificadores, das variáveis globais utilizadas no bloco de código. Portanto, em operações que manipulem este tipo de variável, a instrução fará uma referência a um item contido nesta lista.
- *co_consts*: lista que contém as constantes do bloco de código. Instruções que fazem uso de constantes acessam os itens desta lista. Uma constante num bloco de código costuma ser uma *string*, constante numérica ou referência a outra função ou método.

Outra estrutura importante é o objeto-frame. Tal estrutura é criada a partir de um objeto-código durante a execução da MVP, de forma que consiste numa expansão deste. Durante a execução, à medida que os objetos-código vão sendo lidos pelo interpretador, vão igualmente sendo anexados a um objeto-frame, que por sua vez contém informações referentes apenas à execução do código. Dessa forma, o objeto-frame armazena referências à *thread* de execução atual, às informações de depuração, à documentação do bloco de código (se houver), ao *frame* de execução anterior, etc.

Não obstante, o objeto-frame contém referências importantes para os *namespaces* acessados no bloco de código. Em Python, um *namespace* refere-se a um contexto no qual nomes para variáveis são armazenados e podem ser usados para acessar valores. É uma

espécie de mapeamento entre nomes e objetos. Cada função, módulo ou classe em Python tem seu próprio *namespace*. Esta estrutura portanto consiste num dicionário que mapeia o nome de uma função à sua implementação, ou o nome de uma variável a seu valor, por exemplo. Quando uma variável é utilizada em Python, o interpretador procura o nome no *namespace* local primeiro, depois no *namespace* global e, finalmente, no *namespace* embutido. Essa ordem é conhecida como a "*LEGB Rule*" (*Local, Enclosing, Global, Built-in*).

Reforça-se que o objeto-frame mantém inalterado o conteúdo do objeto-código correspondente. Na Figura 3, expõe-se os campos que constituem o objeto-frame:

Figura 3 – Lista de campos que constituem um objeto-frame.

```
>>> dir(f)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__ne__', '__new__','__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'clear',
'f_back', 'f_builtins', 'f_code', 'f_globals', 'f_lasti', 'f_lineno',
'f_locals', 'f_trace']
```

Fonte: Ike-Nwosu (2018)

Dos campos mostrados na Figura 3, os mais relevantes são:

- *f_back*: contém a referência ao *frame* de execução anterior. Tais *frames* por sua vez constituem a pilha de chamadas (*stack trace*). Quando um bloco de código (uma função, por exemplo) finaliza sua execução, o interpretador utiliza a referência contida neste campo para retornar ao bloco de código anterior (por sua vez o *caller* da função finalizada).
- *f_code*: contém a referência ao objeto-código do bloco de código corrente, tal como visto na Figura 2. Como comentado acima, este permanece inalterado.
- *f_globals*: armazena a referência ao *namespace* global, ou seja, à estrutura que contém as variáveis globais utilizadas pelo código. Observa-se que, diferentemente do campo *co_names* de um objeto-código (Figura 2), o *namespace* reflete o estado atual das variáveis em tempo de execução. Isto é, enquanto *co_names* representa uma lista estática de identificadores gerada durante a compilação, *f_globals* referencia a estrutura que de fato contém os valores que são manipulados durante a execução. *co_names*, portanto, armazena apenas o nome das variáveis, de forma que durante a execução o interpretador usa este nome para buscar a variável

real no *namespace* referenciado por *f_globals*.

- *f_locals*: armazena referência ao *namespace* local. Este campo é similar ao item acima, com exceção apenas de que as variáveis referenciadas são locais. Não obstante, a relação entre *co_varnames* e *f_locals* é idêntica à relação entre *f_globals* e *co_names*.
- *f_builtins*: armazena a referência ao *namespace* embutido, isto é, a estrutura dicionário que contém as funções disponíveis para todos os objetos durante a execução: `print, enumerable, eval, filter, hash`, etc.

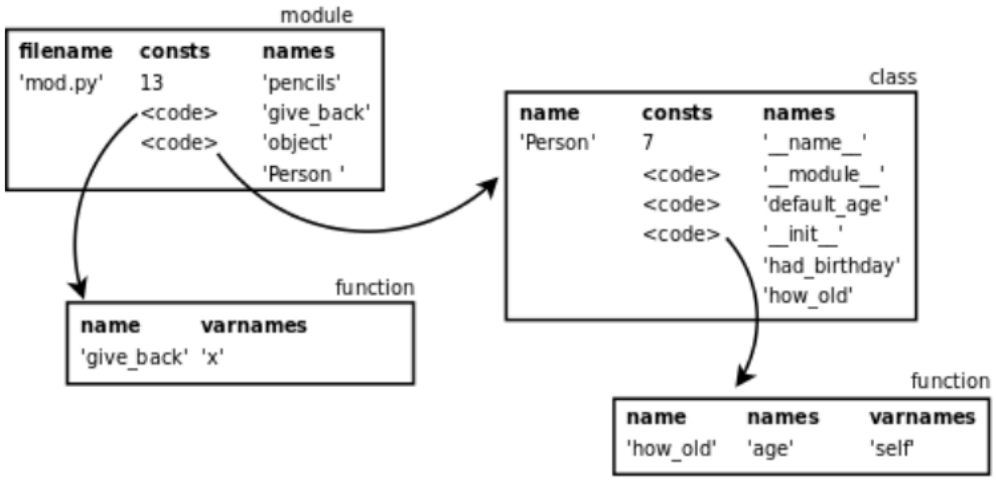
Como já introduzido, objetos-frame e objetos-código estão no cerne da execução do código. Quando a execução é iniciada, o interpretador lê o primeiro objeto-código e acopla-o a um objeto-frame. Neste objeto-frame criado, são atribuídos aos campos *f_locals*, *f_globals* e *f_built-in*, as referências para, respectivamente, o *namespace* local, global e embutido. Após, as instruções contidas no campo *co_code* começam a ser executadas. Em determinado momento, uma função (ou qualquer outro bloco de código) será chamada. Neste momento, o objeto-código dessa função estará inserido na lista de constantes (campo *co_consts*) do objeto-código corrente, e deverá ser carregado para a pilha de execução. Uma vez feito isto, antes da sua primeira instrução ser executada, o mesmo será acoplado a um novo objeto-frame e então o processo começará novamente. Uma vez que todas as instruções de tal função forem executadas, o fluxo de execução volta para o *frame* anterior (através da referência contida em *f_back*, do objeto-frame corrente).

É assim, portanto, que se dá a execução de um código Python, através da troca constante de objetos-frame durantes as chamadas de blocos de código. A Figura 4 ilustra esse comportamento: a partir do módulo (primeiro bloco de código que entra em execução), um primeiro objeto-código contido na lista de constantes é puxado para a pilha, e sua execução começa. Após terminar a execução, o fluxo volta para o bloco de código original (o módulo). Então, uma segunda chamada é feita a outro objeto-código, dessa vez se tratando de uma instanciação de objeto, que por sua vez invoca o construtor da classe. Uma vez no construtor, o fluxo retorna à classe e então de volta ao módulo.

3.2.2 Formato do arquivo PYC

O arquivo PYC é produzido após o processo de compilação. Tipicamente, após o interpretador produzi-lo, ele o insere na pasta *__pycache__*, no mesmo diretório

Figura 4 – Fluxo de execução típico num programa Python

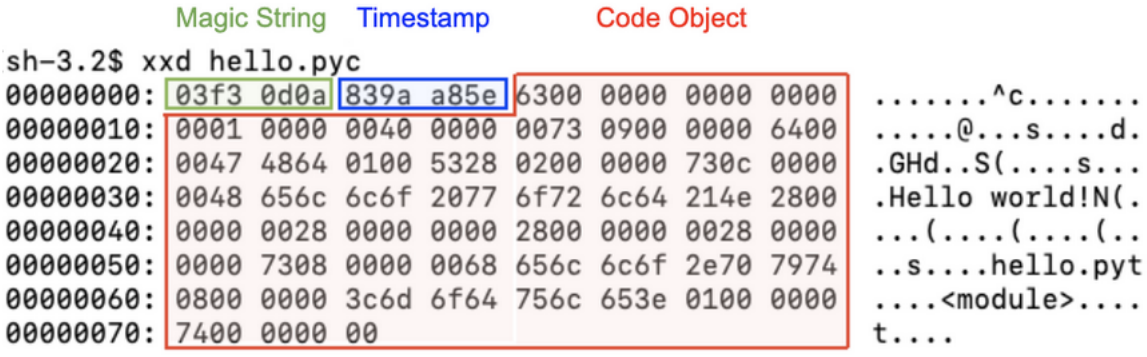


Fonte: Matusiak (2023)

do arquivo cujo código foi compilado (ROSSUM; DRAKE, 2009). Ele consiste nos bytecodes compilados do programa a ser executado. O objetivo desse arquivo é acelerar a inicialização do programa em eventuais reexecuções, uma vez que o código não precisará ser compilado novamente. Assim, na próxima vez que o programa for executado, o interpretador irá procurar este arquivo na pasta para executá-lo, de forma a agilizar o processo e evitar processamento desnecessário.

O mesmo consiste num arquivo binário, e tem estrutura bem definida. A Figura 5 expõe a organização interna do arquivo:

Figura 5 – Estrutura de um arquivo PYC



Fonte: Lyne (2023)

Como pode ser observado na Figura, o arquivo é constituído por 3 partes:

- *Magic Number*: é uma sequência de bytes que identifica a versão do bytecodes e a plataforma para a qual o arquivo foi compilado. No caso específico do Python 3, o número mágico é uma maneira de garantir que o arquivo PYC seja compatível

com a versão correta do interpretador Python. O número mágico é geralmente uma sequência de 4 bytes no início do arquivo PYC. Cada versão do Python tem seu próprio número mágico.

- *Timestamp*: refere-se a um valor que representa a data e a hora específicas em que a compilação ocorreu. Consiste num número inteiro que representa a quantidade de milissegundos decorridos desde a interpretação. É importante porque o interpretador compara a data de modificação do arquivo com este campo, a fim de determinar se o PYC é válido ou precisa ser atualizado.
- Objeto-código: é a estrutura discutida na seção 3.2.1, e que por sua vez contém as instruções e os dados necessários à execução do programa. Como visto na seção supracitada, é criado um objeto-código para cada bloco de código, sendo que um programa Python é constituído por diversos blocos de código. Assim, todos os objetos-código necessários à aplicação alvo precisam estar contidos no arquivo PYC. Como será discutido a seguir, eles são aninhados pelo compilador no arquivo.

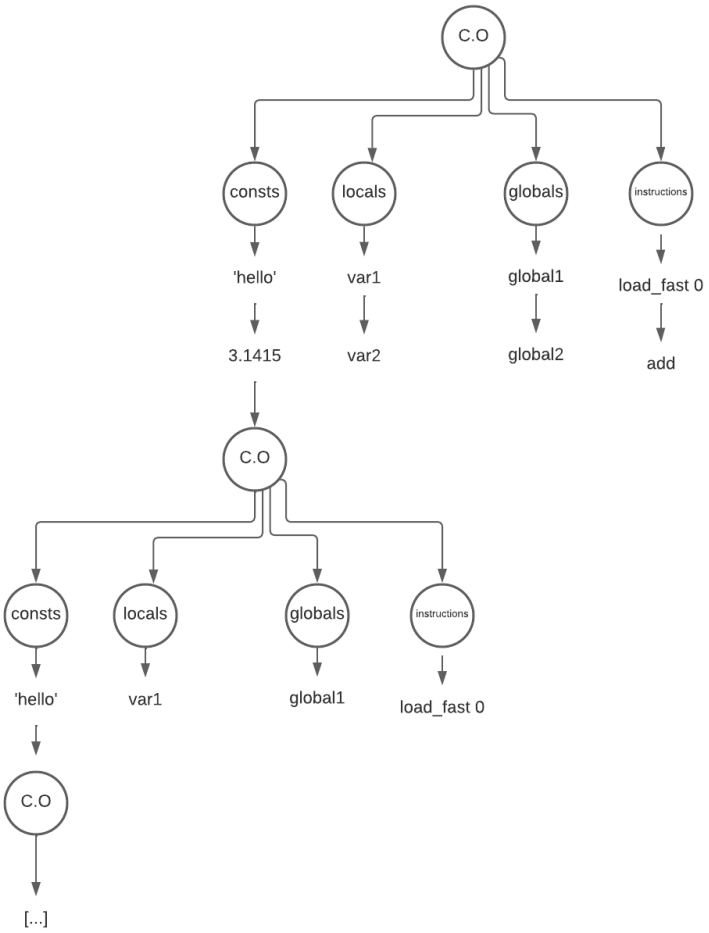
Das partes discutidas, objeto-código é naturalmente a mais importante. Como afirmado, dentro de um PYC estão todos os objetos-código necessários a execução do programa, assim, um arquivo comumente terá dezenas a centenas de objeto-código aninhados. Nesse cenário, fica a questão de como ocorre esse aninhamento. Ike-Nwosu (2018) explica que objetos-código se encontram dentro da lista de constantes de cada objeto-código.

A Figura 6 ilustra esse comportamento. Os diversos objetos-código (C.O) são inseridos pelo interpretador no campo *co_consts* de cada objeto-código. Dessa forma, quando um bloco de código chamar uma função, o objeto-código dela estará em tal lista. Assim, a mesma é inserida na pilha e sua execução começa. Reforça-se que um bloco de código se encontra na lista de constantes de outro apenas se houver referência a este no código, isto é, apenas se houver possibilidade de ser chamado.

3.2.3 Conjunto de instruções baseado em arquitetura de pilha

A pilha de execução na PVM é um componente fundamental responsável pela realização de operações aritméticas e lógicas durante a execução dos programas Python. Como expõe Craig (2005), trata-se de uma estrutura organizada como uma pilha, ou seja, que segue o modelo *Last-In-First-Out* (LIFO), onde operandos, isto é, constantes ou

Figura 6 – Aninhamento de objetos-código num arquivo PYC



Fonte: Autor (2023)

variáveis copiados das suas respectivas localizações na memória, são empilhados antes da execução de operações e desempilhados mais tardar após a sua conclusão (IKE-NWOSU, 2018). Sendo assim, toda a manipulação de variáveis acontece através desta estrutura. O conjunto de elementos essenciais associados à pilha de execução inclui operandos, instruções, um apontador de pilha, e mecanismos para empilhamento e desempilhamento de valores. Operações aritméticas e lógicas são realizadas mediante a manipulação desses elementos, de forma que a MVP busca manter a coerência dos diferentes tipos de dados a serem processados.

No contexto da PVM, cada instrução corresponde a uma operação específica, como adição, subtração, comparação, carregamento e armazenamento de variáveis, etc. A eficiência da pilha de execução reside na sua capacidade de manipular dinamicamente diferentes tipos de dados, permitindo a execução correta das operações. Já o apontador de pilha indica seu topo, determinando a próxima posição disponível para empilhar um

operando. Esse apontador é importante para manter a consistência da pilha durante a execução, mantendo a ordem correta de empilhamento e desempilhamento.

Dado que a PVM é uma máquina virtual orientada à pilha, seu conjunto de instruções busca manipular a pilha, a fim de executar as operações necessárias. Essas instruções são projetadas para realizar uma variedade de operações, incluindo manipulação de operandos, controle de fluxo e chamadas de função. A seguir, são descritas algumas instruções relevantes que desempenham um papel crucial na manipulação da pilha, de acordo com Rossum e Drake (2009):

1. **Instruções de Manipulação da Pilha:**
    - `POP_TOP`: desempilha o topo da pilha.
    - `ROT_TWO`: troca os dois elementos do topo da pilha.
    - `ROT_THREE`: gira os três elementos do topo da pilha.
2. **Instruções Aritméticas e Lógicas:**
    - `BINARY_ADD`, `BINARY_SUBTRACT`, `BINARY_MULTIPLY`, etc.: realizam operações aritméticas sobre os dois elementos do topo da pilha.
    - `BINARY_AND`, `BINARY_OR`, `BINARY_XOR`: realizam operações lógicas sobre os dois elementos do topo da pilha.
3. **Instruções de Controle de Fluxo:**
    - `JUMP_FORWARD`, `JUMP_ABSOLUTE`: controlam o fluxo de execução, alterando o apontador de instrução.
    - `POP_JUMP_IF_TRUE`, `POP_JUMP_IF_FALSE`: realizam saltos condicionais baseados na verdade ou falsidade do topo da pilha.
4. **Instruções de Chamadas de Função:**
    - `CALL_FUNCTION`, `CALL_FUNCTION_KW`: realizam chamadas de função (troca de contexto de memória), empilhando argumentos e desempilhando resultados.
    - `RETURN_VALUE`: desempilha o valor de retorno de uma função.
5. **Instruções de Manipulação de Variáveis:**
    - `STORE_FAST`, `LOAD_FAST`: armazenam e carregam valores em variáveis locais.

Essas instruções bytecodes formam a base para a execução de programas Python na máquina virtual. A variedade de operações oferecidas permite a manipulação dinâmica de tipos de dados, a realização de cálculos aritméticos e lógicos, o controle de fluxo de execução e a chamada de funções.

3.2.4 Semântica das instruções para suporte a objetos Python

A semântica das instruções bytecodes na PVM desempenha um papel crítico na execução dos programas que fazem uso dos recursos de orientação a objetos. De fato, toda instanciação, manipulação de atributos e acesso a métodos acontece por meio das instruções de suporte a objetos. Estas, por sua vez, são bem definidas, sendo abordadas a seguir, de acordo com Rossum e Drake (2009):

Criação de objeto-classe

Antes de efetivamente instanciar novos objetos de determinada classe, é necessário criar um objeto desta classe. Rossum e Drake (2009) afirmam que "criar uma nova classe cria um novo tipo de objeto, permitindo que novas instâncias desse tipo sejam produzidas". E complementa que objetos-classe suportam 2 tipos de operação: instanciação e referência a atributos. A primeira é a criação de objetos a partir do objeto-classe, operação comum às linguagens orientadas a objetos. A segunda, por sua vez, refere-se à possibilidade de acessar atributos e métodos da classe de forma estática, sem a necessidade de instanciar um objeto. Dessa forma, antes de instanciar um objeto da classe `MinhaClasse`, as instruções ilustradas na Figura 7 devem ser executadas.

Figura 7 – Bytecodes para criar objeto-classe

```
LOAD_BUILD_CLASS
LOAD_CONST               0 ( <class code-object> )
LOAD_CONST               1 ('MinhaClasse')
MAKE_FUNCTION            0
LOAD_CONST               1 ('MinhaClasse')
CALL_FUNCTION            2
STORE_NAME               1 ('MinhaClasse')
```

Fonte: Autor(2023)

As instruções da Figura 7 são formatadas de forma similar à utilizada pela documentação para representar código bytecode. A primeira coluna contém o mnemônico da instrução, enquanto que a segunda o operando, e na terceira, entre parênteses, o conteúdo em si. Como discutido na seção 3.2.1, cada tipo de instrução referencia um campo específico do objeto-código corrente. Algumas referenciam a lista de constantes (*co_consts*), outras a de variáveis locais (*co_varnames*) e outras a de variáveis globais (*co_names*). Os operandos de tais instruções são portanto os índices de tais listas. A seguir, especifica-se cada instrução separadamente:

- **LOAD_BUILD_CLASS**: carrega a função incorporada `__build_class__`. A função `__build_class__` é uma função interna do Python que tem a responsabilidade de construir objetos-classe durante a execução do programa.
- **LOAD_CONST**:
  A instrução `LOAD_CONST 0 (<class code-object>)` carrega o objeto-código da classe para a pilha.
  A instrução `LOAD_CONST 1 ('MinhaClasse')` empilha o nome da classe.
- **MAKE_FUNCTION**: cria uma nova função a partir do objeto-código da classe carregado anteriormente. O número zero indica que não há argumentos para a função.
- **LOAD_CONST 1**: carrega novamente o nome da classe no topo da pilha. Isso ocorre porque a instrução anterior, `MAKE_FUNCTION`, removeu o nome da classe da pilha.
- **CALL_FUNCTION**: chama a função criada pela instrução MAKE_FUNCTION. O argumento 2 indica que há dois elementos na pilha para serem passados como argumentos para a função. Esses argumentos são o objeto-código da classe e o nome da classe.
- **STORE_NAME**: armazena o resultado da chamada da função no namespace atual com o nome especificado ("MinhaClasse").

Em resumo, essas instruções em Python são responsáveis por carregar o nome e o objeto-código da classe, criar um objeto-classe com essas informações, chamar esse objeto com o nome da classe e, finalmente, armazenar o resultado no *namespace* com o nome da classe ("MinhaClasse"). Esse processo efetivamente cria uma classe durante a execução do programa.

Instanciação de Objeto

Uma vez criado o objeto-classe, é possível instanciar objetos dessa classe. A instanciação de objetos na linguagem Python ocorre invocando o nome da classe entre parênteses, da seguinte forma:

```
variavel = MinhaClasse('parametro_1', 'parametro_2')
```

Assim, os bytecodes gerados para esta instrução estão ilustrados na Figura 8. A seguir, discorre-se sobre cada instrução.

Figura 8 – Bytecodes para instanciação de objeto

```
LOAD_NAME        0 (MinhaClasse)
LOAD_CONST       2 ('MinhaClasse')
LOAD_CONST       3 ('parametro_1')
LOAD_CONST       4 ('parametro_2')
CALL_FUNCTION    3
STORE_NAME       1 (variavel)
```

Fonte: Autor(2023)

- **LOAD_NAME 0 (MinhaClasse)**: carrega o objeto-classe, discutido na seção anterior, para o topo da pilha. A criação da instância acontece a partir dele. O interpretador irá buscar pelo objeto-classe de nome "MinhaClasse" no *namespace* local.
- **LOAD_CONST 2 ('MinhaClasse')**: carrega a constante cujo índice na lista de constantes do objeto-código corrente é 2, sendo a *string* "MinhaClasse". Isso coloca a *string* no topo da pilha.
- **LOAD_CONST 3 ('parametro 1')**: empilha a constante de índice 3, que é a *string* "parametro 1".
- **LOAD_CONST 4 ('parametro 2')**: empilha a constante de índice 4, que é a *string* "parametro 2".
- **CALL_FUNCTION 3**: chama a função no topo da pilha (que deve ser um objeto-classe, pois estamos lidando com uma instanciação). O número 3 indica que há 3 argumentos no topo da pilha para serem passados para a função. A chamada da função resultará em uma instância da classe "MinhaClasse".
- **STORE_NAME 1 (variavel)**: armazena o resultado da chamada da função (a

instância da classe) na variável com nome "variavel".

Essas instruções formam um trecho de código que instancia um objeto da classe "MinhaClasse" com os parâmetros "parametro_1" e "parametro_2", e armazena essa instância na variável local chamada "variavel".

Herança e Polimorfismo

A semântica de herança na PVM é suportada pelo bytecodes associado a chamadas de métodos e construtores de classes base, como `CALL_FUNCTION` e `BUILD_CLASS`. O polimorfismo é inerente à natureza dinâmica da linguagem, onde a verificação de tipos ocorre em tempo de execução. A Figura 9 ilustra os bytecodes gerados para herança.

Figura 9 – Exemplo de bytecodes para heranca

```
LOAD_CONST          0 ('SubClasse')
LOAD_NAME           1 (MinhaClasse)
CALL_FUNCTION       2
STORE_NAME          2 (SubClasse)
```

Fonte: Autor(2023)

Na Figura 9, os bytecodes representam a criação de uma subclasse chamada `SubClasse` que herda de `MinhaClasse`, a seguir a explicação passo a passo do código:

- **`LOAD_CONST 0 ('SubClasse'):`** carrega a constante `'SubClasse'` na pilha, representando o nome da classe que será instanciada.
- **`LOAD_NAME 1 (MinhaClasse):`** carrega o nome da classe base `MinhaClasse` na pilha.
- **`CALL_FUNCTION 2:`** chama a função associada à classe `MinhaClasse` com dois argumentos.
- **`STORE_NAME 2 (SubClasse):`** armazena a instância da classe criada com o nome `SubClasse` no namespace.

Manipulação de Atributos

A semântica de manipulação de atributos em Python é expressa por meio de bytecodes específicos, como `LOAD_ATTR` e `STORE_ATTR`. Essas instruções permitem o

acesso e a atribuição dinâmica de atributos em objetos. A Figura 10 exemplifica um caso típico de manipulação de atributos.

Figura 10 – Bytecodes para manipulação de atributos

```
LOAD_NAME        0 (instancia)
LOAD_CONST       0 ('novo_atributo')
LOAD_CONST       1 (42)
STORE_ATTR       2 (novo_atributo)
```

Fonte: Autor(2023)

Na Figura 10, os bytecodes representam a atribuição do valor 42 ao atributo `novo_atributo` de uma instância. A seguir a explicação:

- **`LOAD_NAME 0 (instancia)`:** carrega o nome `instancia` na pilha, referindo-se à instância do objeto.
- **`LOAD_CONST 0 ('novo_atributo')`:** carrega a constante `'novo_atributo'` na pilha, representando o nome do atributo que será modificado.
- **`LOAD_CONST 1 (42)`:** carrega a constante `42` na pilha, representando o valor a ser atribuído ao atributo.
- **`STORE_ATTR 2 (novo_atributo)`:** atribui o valor ao atributo especificado na instância do objeto.

3.2.5 Alocação de memória para objetos Python

De acordo com Rossum e Drake (2009), o gerenciamento de memória em Python é realizado através de um mecanismo conhecido como *Python Memory Manager* (PMM), que inclui o coletor de lixo para lidar com a desalocação de objetos não referenciados.

Objetos Python e Estrutura Interna

Cada objeto Python é representado por uma estrutura interna que contém informações fundamentais, tal que Ike-Nwosu (2018) cita principalmente:

- **Tipo do Objeto:** indica o tipo do objeto (inteiro, string, lista, etc.).

- **Referência ao Tipo:** ponteiro para a estrutura que descreve o tipo do objeto.
- **Contagem de Referências:** número de referências ao objeto para gerenciamento de coleta de lixo.
- **Dados do Objeto:** o conteúdo real do objeto, que pode incluir dados e referências adicionais.

Alocação de Memória

A alocação de memória para objetos Python é realizada através de funções como `PyObject_Malloc` que, por sua vez, utiliza a alocação de memória do sistema operacional subjacente. Alocar memória para um novo objeto envolve a determinação do tamanho necessário, levando em consideração a estrutura interna do objeto e os dados que ele armazenará.

Inicialização do Objeto

Após a alocação de memória, a inicialização do objeto ocorre. Dessa forma, a PVM configura o campo responsável pela contagem de referências do objeto para 1 (já que o objeto acabou de ser instanciado), e inicializa os atributos do novo objeto para valores iniciais.

3.2.6 Reciclagem de memória na MVP

A reciclagem de memória na Máquina Virtual Python (MVP) é um processo essencial para garantir o uso eficiente dos recursos do sistema. Esta fase envolve a identificação e liberação de memória alocada para objetos que não são mais acessíveis pelo programa em execução. A MVP emprega diferentes estratégias de coleta de lixo (ROSSUM; DRAKE, 2009), sendo a principal a abordagem baseada na contagem de referências. Outras estratégias, como a coleta de lixo cíclica, são utilizadas para identificar e coletar objetos que formam ciclos de referências, os quais não seriam adequadamente gerenciados pela abordagem baseada em contagem de referências.

Estratégias de Coleta de Lixo

A seguir, discorre-se sobre as principais estratégias de coleta de lixo empregadas pela MVP:

- **Contagem de Referências:** é o principal algoritmo de coleta de lixo empregado pela MVP. A premissa central deste método consiste em rastrear quantos contextos distintos mantêm uma referência a um objeto específico. Esses contextos podem englobar outros objetos, variáveis globais, estáticas e locais. Quando o contador de referências de um objeto atinge zero, indica que não existem mais referências válidas para o objeto em questão, possibilitando assim a desalocação de sua memória. Caso o objeto removido contenha referências a outros objetos, os contadores de referência desses objetos referenciados são decrementados. Estes objetos adicionais podem, por conseguinte, ser desalocados se a redução de seus contadores de referência atingir zero, desencadeando um processo sucessivo. É oportuno salientar que Python fornece meios de monitorar a contagem de referências a um objeto em tempo de execução, bastando que o usuário interrompa a execução de uma aplicação (através da inserção de um *breakpoint* durante uma sessão de depuração por exemplo), e use a chamada:

  - `sys.getrefcount(OBJECT_NAME)`

  Assim, é possível conhecer a quantidade de referências a um objeto específico em tempo real.

- **Coleta de Lixo Cíclica:** este método atua em conjunto com a contagem de referências, e foi desenvolvido para lidar com a problemática dos ciclos de referência. Esses ciclos ocorrem quando uma coleção de objetos mantém referências recíprocas, formando uma estrutura circular, o que coíbe a liberação de memória por meio da contagem de referências. O Coletor de Lixo Cíclico inicialmente identifica os ciclos de referência, nos quais objetos se referenciam circularmente, e, em seguida, procede-se à marcação dos objetos alcançáveis, num processo semelhante à contagem de referências.

  Além da desalocação de memória, o Coletor de Lixo Cíclico invoca os métodos `del` dos objetos sendo coletados, permitindo a liberação de recursos não relacionados à memória, como conexões de rede ou arquivos abertos. Por fim, o Coletor de Lixo Cíclico é configurável por meio de parâmetros específicos, como o limiar de geração, que determina o momento em que o coletor de lixo

é acionado, além de outros parâmetros relacionados ao desempenho. Assim, em síntese, o Coletor de Lixo Cíclico do Python representa uma extensão ao coletor de lixo convencional, abordando de maneira específica as complexidades inerentes às referências cíclicas entre objetos, assegurando uma eficiente liberação de memória em cenários mais intricados do que aqueles tratados pela contagem de referências isolada.

3.3 Trabalhos Correlatos

Como afirmado no capitulo 1, apesar dos potenciais benefícios de uma arquitetura nativa para os bytecodes da linguagem Python, tal iniciativa de desenvolvimento não é encontrada na literatura para esta linguagem. Ausência que, por sua vez, contrasta com os diversos processadores desenvolvidos para Java. Como introduzido no capítulo supracitado, existem 4 principais publicações neste sentido: PicoJava-I (O'CONNOR; TREMBLAY, 1997), JOP (SCHOEBERL, 2003), JHisc (YIYU et al., 2006) e JAIP-MP (TSAI; WU; SU, 2015). Considerando que as características de cada um, assim como os resultados positivos, foram comentados no capítulo 1, o objetivo desta seção é elucidar as soluções propostas para os gerenciadores de memória destas arquiteturas, haja vista que as soluções propostas para suporte nativo ao Java mantém forte potencial inspirador para o projeto de uma arquitetura nativa análoga para a Linguagem Python.

Apesar da importância do gerenciamento de memória para o funcionamento desses sistemas, poucos trabalhos detalham a implementação desse gerenciamento. Todas as arquiteturas aqui discutidas abordam o gerenciador dinâmico de memória em seu processador, mas diversas não informam sobre questões básicas, como qual algoritmo de alocação foi utilizado, ou como a cache e o coletor de lixo foram implementados. Assim, a profundidade das informações fornecidas nesta seção fica restrita ao detalhamento fornecido pelos respectivos autores, os quais deram diferentes enfoques às apresentações das suas arquiteturas, de acordo com a natureza e foco da publicação de cada trabalho.

3.3.1 PicoJava-I

O'connor e Tremblay (1997) reforçam que a JVM não fornece uma especificação rigorosa de como o gerenciamento de memória deve ocorrer. Especifica apenas que

quaisquer espaços alocados para novos objetos devem ser pré-inicializados com zeros, e que deve haver, necessariamente, coleta de lixo automática. Por ser automática, o programador não deve se preocupar com os objetos não mais utilizados na aplicação, isto é, objetos que por sua vez não sejam mais referenciados por nenhuma variável (portanto inacessíveis). Como comenta o autor, existem diversos métodos de coleta de lixo disponíveis na literatura, de forma que boa parte implica em, após intervalos regulares de tempo, percorrer a *heap* (memória reservada para alocação dinâmica de objetos) em busca de objetos não mais referenciados, para então excluí-los, e devolver o espaço antes ocupado ao *pool* de memória disponível.

Para esta implementação, o autor optou por um método no qual a penalidade de desempenho durante a varredura da *heap* fosse reduzida: coletores geracionais. Esta classe de coletor por sua vez se baseia na observação empírica de que a maioria dos objetos tem uma curta vida útil e, portanto, são mais propensos a se tornarem inutilizados rapidamente. Assim, a memória é dividida em diferentes gerações, geralmente representadas por espaços distintos: jovens, médios e antigos. A ideia é que a coleta de lixo seja realizada com frequência nos espaços mais jovens, onde a maioria dos objetos se torna obsoleta, enquanto os espaços mais antigos são coletados com menos frequência (WILSON, 1992).

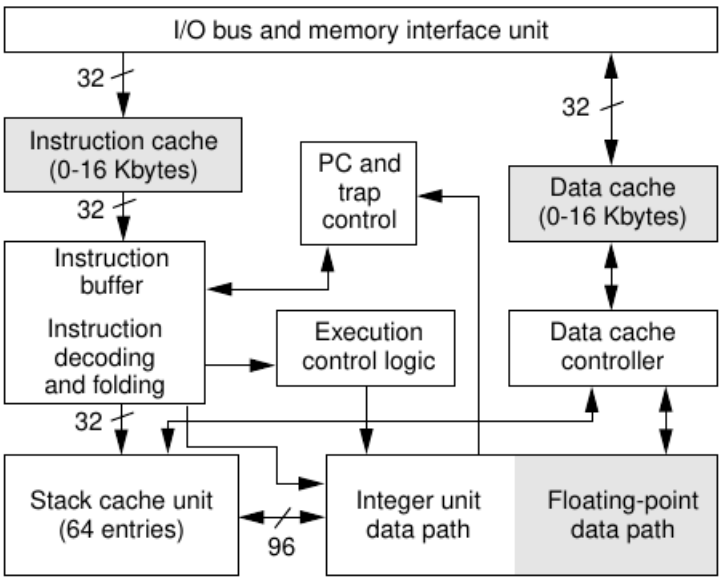
O funcionamento básico de um coletor de lixo geracional pode ser descrito em termos de dois principais algoritmos: o algoritmo de marcação e o algoritmo de varredura. Durante a fase de marcação, o coletor identifica todos os objetos alcançáveis a partir de raízes de referência, como variáveis locais, globais e pilhas de execução. Os objetos alcançáveis são marcados como ativos, enquanto os não alcançáveis são marcados como candidatos à coleta. Assim, a estratégia geracional aproveita a observação de que a maioria dos objetos se torna inutilizada rapidamente. Portanto, ao realizar a coleta com mais frequência nos espaços mais jovens, é possível reduzir o custo computacional associado à coleta de objetos obsoletos. Essa abordagem aumenta a eficiência do coletor de lixo, proporcionando um melhor equilíbrio entre a coleta de lixo e o desempenho do programa em execução (WILSON, 1992). É esta portanto a abordagem implementada pela *Memory Management Unit* (MMU) da arquitetura. É importante mencionar que o processo de marcação é realizado inteiramente por hardware, e o de varredura (remoção do objeto) através de chamadas de software.

O autor adota uma estratégia específica a fim de balancear a complexidade de determinadas tarefas com a frequência de utilização das mesmas. Por exemplo,

invocar métodos é uma tarefa complexa, e passível de ser realizada por software (o que facilitaria sua implementação e eventual manutenção). Entretanto, como é crítica para o desempenho, uma vez que chamadas a métodos e funções são bastante frequentes, o autor opta por implementá-las através de microcódigo (sequência de instruções elementares que são interpretadas e executadas pelo controle interno do processador). Já em relação à criação de objetos, atividade igualmente complexa, mas por ser bastante infrequente segundo o autor, opta-se por implementá-la em software.

Quanto à questão das caches, o autor informa que implementa caches para dados e para instruções, como é possível observar na Figura 11, a qual ilustra a arquitetura geral do processador. Assim, tanto os dados como as instruções ficam disponíveis de forma eficiente quando houver necessidade de acessá-las novamente. O'connor e Tremblay (1997) ainda detalham o seguinte: cache de 16Kbytes, 2-vias, associativa e que implementa *write-back* como estratégia de atualização de dados. Uma cache associativa é uma estrutura de memória cache que permite que blocos de dados sejam armazenados em qualquer posição da cache, sem uma correspondência fixa entre blocos e linhas específicas. Isso oferece flexibilidade na alocação de dados, reduzindo a probabilidade de conflitos em comparação com o mapeamento direto, mas pode envolver uma complexidade adicional no hardware devido à necessidade de comparadores de correspondência em cada linha da cache (STALLINGS, 2010). Quanto ao mecanismo de *write-back*, é uma política de atualização na RAM de dados modificados existentes nas caches. Quando ocorre uma operação de escrita que modifica um dado na cache, a modificação é inicialmente feita apenas na cache, e a linha de cache é marcada como "suja"(*dirty*). A escrita de volta para a memória principal acontece apenas quando a linha de cache é substituída, adiando a atualização para otimizar a eficiência, reduzindo o tráfego na memória principal (STALLINGS, 2010).

Figura 11 – Arquitetura do processador PicoJava-I



Fonte: O'connor e Tremblay (1997)

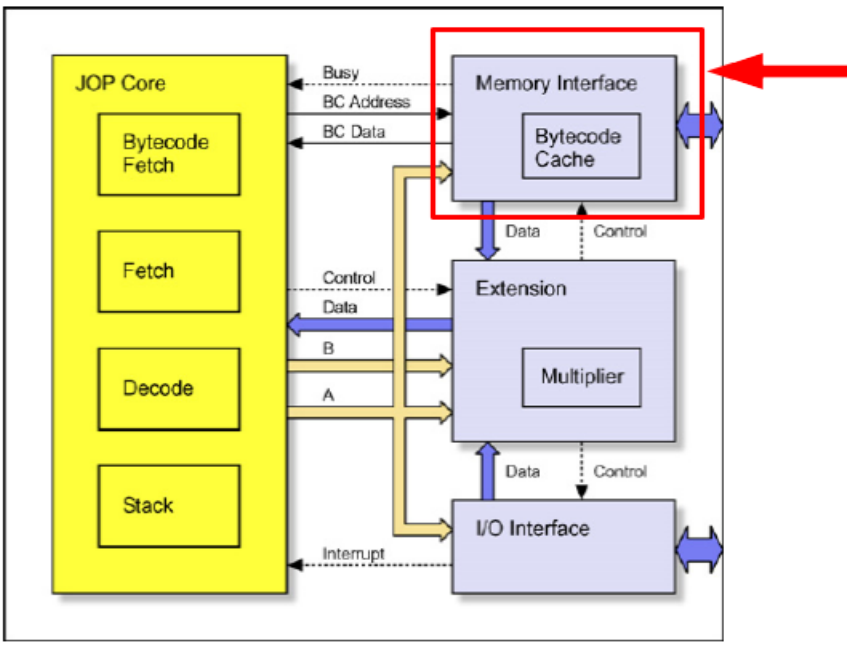
3.3.2 JOP

Como previamente introduzido, *Java Optimized Processor* (JOP) consiste num processador nativo para Java, com foco em sistemas embarcados. Nesta arquitetura, Schoeberl (2003) discorre sobre a interface de memória do sistema, responsável pelo gerenciamento de memória do processador. A Figura 12 ilustra esta estrutura, destacada em vermelho:

Esta interface de memória do sistema estabelece uma conexão entre a memória principal e o núcleo do processador. Além disso, essa interface inclui a cache de bytecode. Um módulo de extensão é responsável por controlar as operações de leitura e escrita de dados. O sinal "*busy*"é utilizado por uma instrução de microcódigo própria (`wait`) para sincronizar o núcleo do processador com a unidade de memória. O núcleo do processador realiza a leitura de instruções de bytecode por meio de barramentos dedicados (endereço de bytecode e dados de bytecode) provenientes do subsistema de memória. O pedido para que um método seja armazenado em cache é executado por meio do módulo de extensão, porém, a detecção e o carregamento de um acerto no cache são conduzidos pela interface de memória de forma independente do núcleo do processador (e, portanto, de maneira concorrente).

Tal qual as demais implementações discutidas neste trabalho, instruções

Figura 12 – Estrutura de bloco do processador JOP



Fonte: Schoeberl (2003)

demasiadamente complexas (como `new`) e infrequentes, são implementadas em software. Da mesma forma, instruções complexas porém frequentes, devem ser implementadas através de microcódigo. Quanto ao recurso de coleta de lixo, embora o autor mencione como presente na implementação (considerando ser um requisito da JVM), o mesmo não entra em detalhes, e não explicita o algoritmo utilizado.

O JOP propõe duas caches previsíveis em termos temporais: uma *stack cache* (que alimenta a pilha de execução), por sua vez substitui a cache de dados, e uma *method cache*, destinada a armazenar instruções. A *stack* cache é dividida em dois níveis, sendo os dois elementos superiores implementados como registradores, enquanto o nível inferior consiste em uma memória *on-chip* extensa. O preenchimento e esvaziamento entre esses níveis são realizados em hardware, enquanto as operações entre a memória *on-chip* e a principal são controladas por microcódigo, garantindo previsibilidade temporal. A troca entre a *stack cache on-chip* e a memória principal pode ocorrer durante a invocação/retorno de método ou a troca de *thread*.

A *method cache*, por sua vez, adota uma abordagem inovadora, armazenando métodos completos. O preenchimento da cache ocorre somente em casos de falha na invocação ou retorno de método, assegurando que todos os outros bytecodes tenham um acerto de cache garantido. A organização da cache por blocos, semelhante a linhas de cache, é notável, com a substituição de blocos dependendo da árvore de

chamadas, em contraste com endereços de instruções. Essa abordagem facilita a análise do comportamento no pior caso.

A eficácia dessas estratégias é apoiada pelo fato de que, embora o JOP busque uma previsibilidade temporal para a análise do *Worst-Case Execution Time* (WCET), ainda proporciona ganhos substanciais de desempenho, sendo sua *method cache* comparável em desempenho, no caso médio, a uma cache direta mapeada. O tamanho máximo do método é restrito pela capacidade da *method cache*, sendo verificado por uma ferramenta de pré-link para garantir conformidade com essa restrição. Essa abordagem específica visa atender às demandas de aplicações críticas em sistemas de tempo real, oferecendo um equilíbrio entre previsibilidade temporal e desempenho médio.

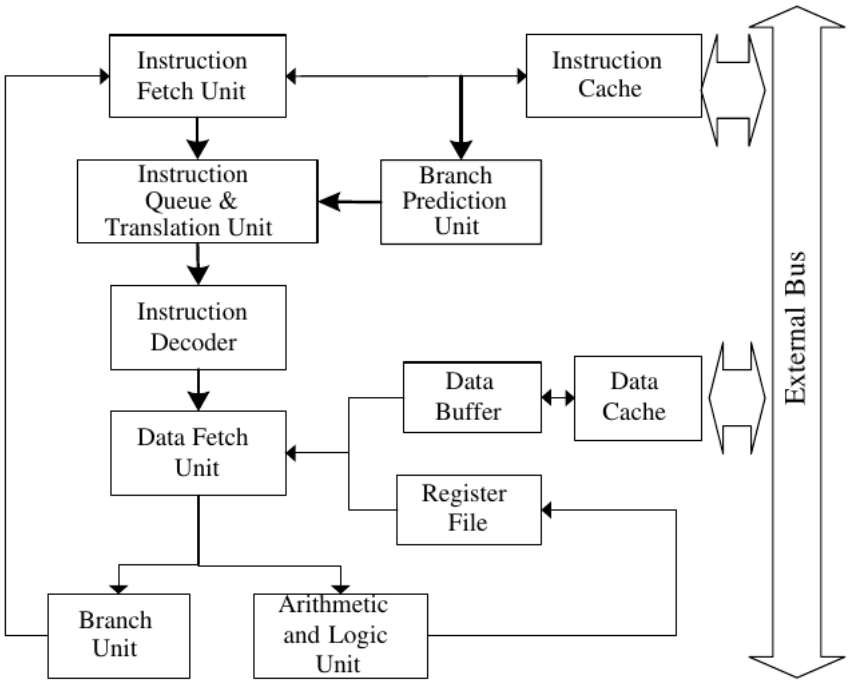
3.3.3 jHISC

jHISC é uma alternativa de processador nativo para bytecode Java, voltado para o nicho de sistemas embarcados. Portanto, seu objetivo é executar as instruções da JVM com tempo previsível e baixo consumo de energia. Como informa Yiyu et al. (2006), 83% dos bytecodes relacionados à orientação a objetos são implementados por hardware diretamente. Esta arquitetura, da mesma forma que a anterior, adota uma cache para dados e outra para instruções. É possível observar ambas na Figura 13, a qual expõe a arquitetura do processador:

Neste trabalho, o autor aborda a importância da representação de objetos em sistemas de programação orientada a objetos, destacando seu impacto na velocidade de acesso aos objetos. Uma representação eficiente maximiza a eficiência das operações com objetos e minimiza a sobrecarga de armazenamento. Para atender a esses requisitos, o modelo de objeto deve seguir algumas regras, como ter um cabeçalho de objeto pequeno que contenha informações suficientes para reduzir a sobrecarga de acesso à memória. Assim, é proposto um cabeçalho de objeto de três palavras para a Máquina Virtual Java, composto por sete campos: Tipo de Objeto, Tipo de *Array*, Trava, Informações de Coleta de Lixo, Tamanho de Dados, Classe e Tamanho de *Array*. Cada campo tem uma função específica, como definir o tipo de objeto, especificar o tamanho do espaço de dados e lidar com a coleta de lixo.

O autor comenta sobre três tipos de contextos de objeto mapeados para a arquitetura de hardware do processador: contexto de instância, contexto de classe e contexto de método. Cada contexto tem uma estrutura específica e inclui cabeçalhos

Figura 13 – Arquitetura do processador jHISC



Fonte: Yiyu et al. (2006)

e espaços de dados relevantes. Além disso, o texto destaca a eficiência do acesso a objetos no jHISC em comparação com processadores Java baseados em pilha, como PicoJava I, devido à organização contínua dos componentes e à realização de verificações relacionadas ao objeto durante o acesso, o que é feito por hardware.

No contexto de sistemas orientados a objetos, a invocação de métodos desempenha um papel crucial na performance do sistema. Isso se deve à comunicação entre objetos por meio de métodos, sendo que cada invocação de método pode consumir várias dezenas de ciclos de *clock* por meio de uma interrupção de software. Portanto, o processador orientado a objetos deve usufruir de manipulação rápida e segura de métodos diretamente no hardware. Na Máquina Virtual Java, existem dois tipos básicos de métodos: método de instância e método estático, que são invocados por dois bytecodes diferentes. O bytecode `invokevirtual` é usado para invocar um método de instância, enquanto o bytecode `invokestatic` é utilizado para invocar um método estático. No entanto, no contexto do processador jHISC, é fornecida uma instrução chamada `ivkclass` para invocar um método estático.

Para métodos de instância, se estiverem no mesmo contexto de classe que o invocador, a instrução interna de invocação de método `ivkinternal` é utilizada; caso contrário, a instrução de invocação de método de instância `ivkinstance` é empregada.

Durante cada invocação, o processador precisa localizar o código do método, verificar o controle de acesso, empurrar os dados do contexto do objeto atual para a pilha do sistema e transferir o controle para o método invocado. Quanto mais contextos de objetos são alternados, mais dados são necessários para serem empurrados na pilha do sistema. Uma invocação interna de método geralmente requer apenas a troca dos contextos de métodos envolvidos, enquanto uma invocação de método de classe requer um contexto de classe envolvido adicional. Uma invocação de método de instância exige a troca de todos os três tipos de contextos: contexto de método envolvido, contexto de classe e contexto de instância. Essas são as razões pelas quais o bytecode `invokevirtual` foi dividido nas instruções `invokevirtual` e `ivkinstance` para acelerar sua execução.

Ao encontrar uma instrução de revogação de método, o processador restaura os contextos do objeto retirando os conteúdos do contexto previamente armazenados num registrador específico, chamado de registrador de estágio. Se assumirmos que todos os dados estão na cache de dados e que o objeto está resolvido, a invocação interna do método consumirá cinco ciclos de *clock*. Esses ciclos envolvem a leitura do descritor do operando da tabela de descritores do operando, obtenção do endereço direto do Espaço de Dados de Classe (a parte da memória que armazena as variáveis estáticas de uma classe) da classe atual, acesso ao cabeçalho do objeto do método invocado, armazenamento do contexto do método atual e alocação do novo quadro de objeto para o método invocado, respectivamente.

No entanto, em PicoJava I, a invocação do método é realizada pelo bytecode `invokevirtual` por meio de uma chamada de software inicialmente. Isso consumirá 195 ciclos de *clock* no caso da mesma suposição. Dentre esses ciclos, 31 são dedicados à leitura da referência do objeto e à verificação se o objeto está resolvido; 29 são consumidos pela extração do ponteiro para a entrada da tabela de informações do campo e verificação da *flag* de acesso; 107 são gastos no acesso ao método invocado e verificação do índice do método para determinar o tipo de instrução; 13 são usados para reconstruir o quadro de armadilha e retornar; 15 são tomados pela instrução rápida.

3.3.4 JAIP-MP

Java Application IP - Multicore (JAIP-MP) (TSAI; WU; SU, 2015), consiste num processador nativo para bytecodes Java, e que por sua vez é voltado a oferecer uma arquitetura genuinamente *multicore*. O sistema em um chip (SoC) é composto por um

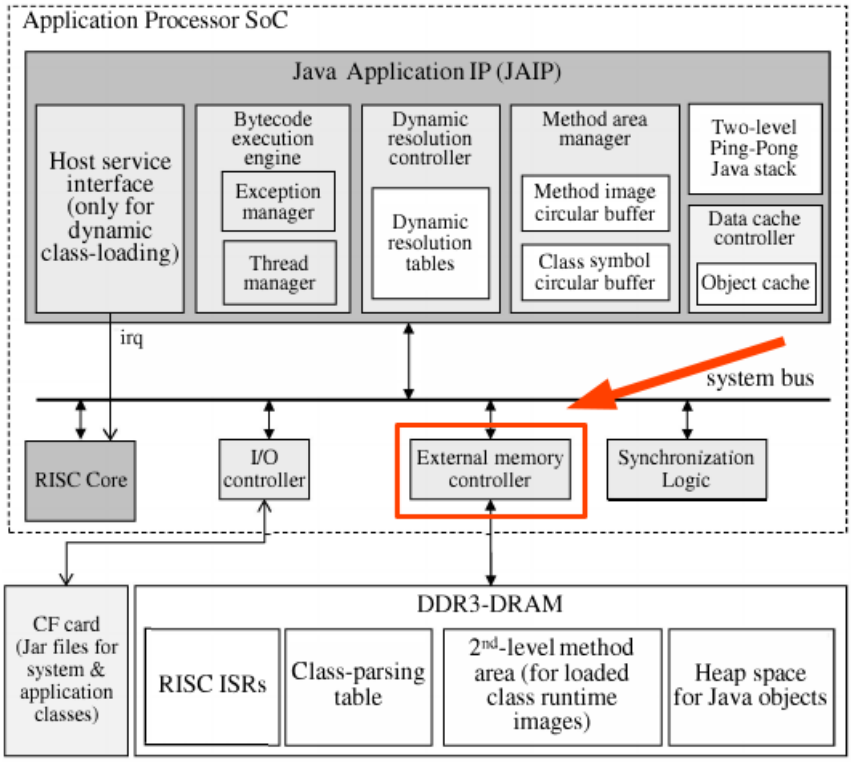
núcleo RISC e um núcleo JAIP. No contexto da execução de um programa Java, o núcleo RISC assume a responsabilidade pelo carregamento dinâmico e análise dos arquivos de classe armazenados em um arquivo JAR no cartão *Compact Flash* (CF). O analisador RISC carrega os arquivos de classe Java padrão para a RAM em tempo de execução de maneira dinâmica, possibilitando a execução direta pelo núcleo JAIP. Além disso, o analisador RISC mantém a tabela de referência cruzada de símbolos armazenada na memória principal para todas as classes carregadas. O núcleo JAIP é completamente responsável pela execução de bytecodes, resolução dinâmica, *caching* de métodos e objetos, gerenciamento de memória e escalonamento de múltiplas *threads*.

O JAIP adota um design de área de método em dois níveis. Todas as classes carregadas durante a execução são armazenadas na DDR3 SDRAM (ou seja, a área de método de segundo nível) usando a política de resolução tardia. No entanto, um método Java (e suas informações de símbolos relacionadas) deve ser carregado na memória cache de método *on-chip* (a área de método de primeiro nível) antes de poder ser executado pelo mecanismo de execução de bytecodes. Em resumo, as imagens completas de classe das aplicações Java são armazenadas na DDR3 SDRAM, enquanto os métodos e informações de símbolos mais recentemente utilizados são armazenados na memória cache de método *on-chip* de maneira FIFO (*First In, First Out*).

De fato, o gerenciamento de memória desta arquitetura se mostra mais complexo em relação às demais, uma vez que é fundamental garantir a coerência dos dados entre os diferentes núcleos de processamento (sendo 4 no total). Para isto, Tsai, Wu e Su (2015) implementa o *Controlador de Coerência de Cache* (DCC). Cada núcleo por sua vez contém uma unidade deste módulo.

A arquitetura detalhada do Controlador de Coerência de Cache (DCC) é apresentada na Figura 14, composta por quatro sub-módulos distintos. O Controlador de Coerência de Cache (CCC) tem a responsabilidade de manter a consistência dos dados entre os Controladores de Objetos (ORC) de cada núcleo. O ORC adota a política *Least Recently Used* (LRU) e a estratégia *write-through* para o armazenamento em cache de objetos Java na *heap*. O Controlador de Mutex decodifica sequencialmente as solicitações enviadas pelos núcleos JAIP e ativa o submódulo correspondente. O Controlador de Atribuição de *Threads* (TAC) realiza o balanceamento de carga entre todos os núcleos JAIP, enviando registros especiais para os núcleos com o menor número de threads prontas ao ser invocado o método `Thread.start()`. O Controlador de Acesso a Objetos de Bloqueio (LOAC) mantém informações sobre threads em espera associadas a cada objeto

Figura 14 – Estrutura interna do JAIP-MP



Fonte: Tsai, Wu e Su (2015)

de bloqueio ocupado.

4 DESENVOLVIMENTO DO GERENCIADOR

Este capítulo discorre acerca do desenvolvimento da solução proposta. Dessa forma, a seção 4.1 expõe o funcionamento geral do sistema proposto, buscando explicar a solução como um todo. As seções seguintes, por sua vez, aprofundam as etapas discutidas na seção 4.1, detalhando cada elemento da solução proposta. Assim, a seção 4.2 detalha o processo de extração de dados do PYC, a seção 4.3 detalha o funcionamento do Software Gerenciador e a seção 4.4 aborda o utilitário de teste criado para auxiliar na conferência do funcionamento do Gerenciador.

4.1 Fluxo de Funcionamento do Gerenciador

O papel do Gerenciador de Contexto proposto é fornecer parte do suporte necessário para a execução de programas em um processador hipotético de bytecode Python. Esse suporte consiste em gerenciar o contexto de execução, incluindo a troca de objetos-frame, a persistência de variáveis globais e o controle da pilha de execução durante chamadas de funções. Além disso, o Gerenciador assegura que o processador possa acessar corretamente as instruções, constantes e variáveis que compõem o programa, garantindo o suporte às operações fundamentais associadas ao modelo de execução Python. Qualquer programa Python, por mais simples que seja, exige a manipulação precisa desses elementos para funcionar adequadamente. Daí a importância de um Gerenciador de Contexto, pois ele estabelece as bases necessárias para a execução correta e eficiente de programas Python em hardware dedicado. Tendo em vista esses pontos, deu-se ao gerenciador desenvolvido o nome de CFP (*Code Frame Processor*).

Antes de adentrar na solução, a partir deste capítulo, é necessário ajustar a nomenclatura empregada. No capítulo anterior, distinguiu-se entre objetos-código e objetos-frame: o primeiro, gerado durante a compilação, é um elemento estático que contém os dados e instruções necessários à execução de uma função; o segundo, criado quando o objeto-código entra em execução, é uma estrutura dinâmica que o encapsula no contexto da PVM, tornando-o apto para execução. Neste trabalho, simulam-se essas estruturas para dar suporte ao processador hipotético, mas não é viável nem desejável replicar integralmente um objeto-frame da PVM no CFP. Assim, define-se que, no contexto do CFP, um objeto-frame corresponde a um objeto-código em execução,

transformado em uma entidade dinâmica, com listas de variáveis também dinâmicas, capazes de receber valores calculados durante a execução dos bytecodes, e que não goza de atributos pertencentes à PVM, como referências a *namespaces* abstratos e identificadores de *threads*, que não se aplicam ao gerenciador proposto.

Além do efetivo desenvolvimento do CFP, este trabalho se propõe a estabelecer padrões de funcionamento para o processador hipotético destino. Tais padrões são necessários, uma vez que diversas decisões de planejamento precisam ser tomadas a fim de se chegar a um modelo funcional futuramente. Assim, ao longo dessa seção serão mencionados formatos específicos de transferência de dados, ações que o processador deverá tomar ao processar determinadas instruções, entre outros. Assim, o processador destino precisará aderir a tais decisões de projeto, a fim de que a integração entre ambos funcione como o esperado. Os requisitos esperados pelo processador são abordados ao longo desta seção, encontrando-se de forma resumida na Tabela 1.

Tabela 1 – Requisitos esperados pelo processador para determinadas instruções e situações

Instrução ou Situação	**Requisito esperado pelo processador**
Ínicio da execução	Sinalizar CFP; aguardar o primeiro objeto-frame do CFP
MAKE_FUNCTION	Carregar índice do objeto-código na variável destino da operação
CALL_FUNCTION	Armazenar os parâmetros para o novo objeto-frame (ou seja, parâmetros da função invocada); codificar o objeto-frame atual e enviá-lo ao CFP; aguardar *payload* do próximo frame
RETURN	Armazenar resultado; sinalizar CFP; aguardar novo objeto-frame do CFP

Expõe-se que o modelo aqui proposto não atende a todas as funcionalidades da linguagem Python, limitação decorrida da incapacidade de recriar recursos disponíveis na Máquina Virtual Python, a qual goza de toda abstração que o processo de interpretação proporciona. As funcionalidades não suportadas, identificadas na versão atual, encontram-se na Tabela 2:

Apesar de tais limitações, reforça-se que o modelo proposto cobre considerável parcela de programas e recursos da linguagem, e serve como ponto inicial de um trabalho que possa ser expandido no futuro. Programas com funções complexas, e aninhadas, que façam uso de variáveis globais, são, teoricamente, suportados pelo CFP, desde que não façam uso dos recursos citados na Tabela 2.

Tabela 2 – Funcionalidades da linguagem Python não suportadas pelo modelo proposto

Funcionalidade não suportada	**Causa**
Orientação à objetos	Não foi identificada uma forma viável de implementar o bytecode *LOAD_BUILD_CLASS*, fundamental à OOP
Funções lambda	A forma proposta de lidar com a instrução *MAKE_FUNCTION*, que será vista a seguir, parte de premissas que não se aplicam a este tipo de função

4.1.1 Obtenção dos dados necessários à execução de programas Python

Como abordado anteriormente, um programa Python é compilado para uma sequência de bytecodes da respectiva linguagem. O resultado da compilação do programa é um arquivo de extensão PYC. Este, por sua vez, consiste num arquivo binário, onde estão armazenados todos os dados e instruções necessários à execução de código Python. Um código compilado em uma máquina funciona exatamente da mesma forma em qualquer outra que suporte o interpretador da linguagem, bastando transferir o arquivo PYC à mesma.

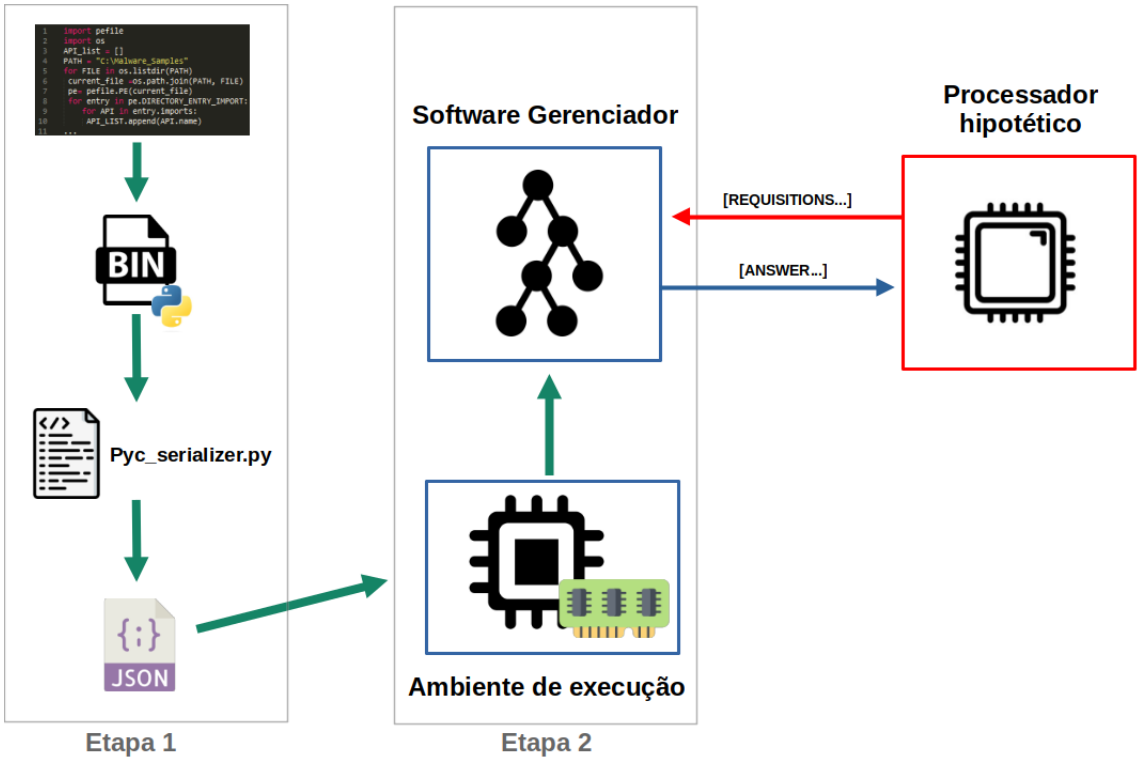
Tal arquivo é, portanto, a chave para a obtenção dos dados e instruções que devem ser fornecidos ao processador. Sendo um arquivo binário, sua interpretação depende de informações herméticas, propositalmente omitidas pela documentação da linguagem. Felizmente, não é necessário extrair tais dados manualmente, pois a biblioteca padrão da linguagem fornece uma função que permite essa decodificação nativamente. Esta função permite acesso aos dados e instruções do programa escrito. Através dela, criou-se uma ferramenta para realizar tal extração, a qual extrai os dados necessários à execução, converte-os num formato legível, e os escreve num arquivo JSON. Após este processo, tem-se todas as informações necessárias à execução do programa num arquivo transferível e legível a outros programas. O arquivo JSON será consumido pelo CFP, o qual processará seu conteúdo, e após, estará pronto para servir ao processador.

O processo descrito até o momento é ilustrado pela Figura 15. Salienta-se que as etapas principais do processo serão esmiuçadas nas seções seguintes, de forma que esta seção busca apenas fornecer uma visão geral do processo.

Na Figura 15, é possível observar 2 etapas principais:

- Etapa 1: esta etapa é executada inteiramente por uma ferramenta desenvolvida pelo autor, chamada de *pyc_extractor*. É uma ferramenta escrita em Python, e invocada

Figura 15 – Ilustração da proposta



Fonte: Autor (2024)

através de linha de comando. Ao ser executada, deve-se informar como parâmetro o nome do arquivo a ser processado. O arquivo deve conter o código Python que será executado no processador hipotético. A ferramenta compila o código presente no arquivo para bytecodes Python, gerando o arquivo PYC no sistema de arquivos do usuário. Após, este arquivo é aberto pela ferramenta e uma função de serialização roda sobre o mesmo, de forma que todos os dados necessários à execução (vide a seção 3.2.1) são extraídos e copiados num arquivo JSON. Este arquivo fica à disposição do usuário, e seu conteúdo deverá posteriormente ser carregado para o CFP. O arquivo JSON resultante reflete a estrutura dos objetos-códigos aninhados no arquivo PYC.

- Etapa 2: Após, o conteúdo do arquivo JSON deve ser transferido para o ambiente de execução (um sistema operacional, *FPGA*, etc.), e então ser carregado para o CFP. Assim como os objetos-códigos estão aninhados no arquivo PYC e, portanto, no arquivo JSON, o CFP processará cada objeto-código presente na memória, convertendo-os portanto, a estruturas semelhantes aos objetos-frames discutidos na seção 3.2.1. Devido ao aninhamento dos objetos-códigos no arquivo, o processo é realizado de forma recursiva. Após o processamento de todos os objetos-códigos,

uma estrutura contendo o objetos-frame inicial é obtida. A partir desta estrutura, é possível dar início à comunicação com o processador Python.

4.1.2 Dinâmica de comunicação com o processador

Um processador que busque implementar a Máquina Virtual Python em hardware deve suportar inúmeros bytecodes da linguagem, dos quais uma parcela é forçosamente referente à troca de contextos da memória. Como mostrado no capítulo anterior, a execução de um programa Python, por mais simples que seja, consiste numa troca frequente de objetos-frame, ou contextos, de execução. Cada objeto-frame possui, como mencionado, instruções bytecode, listas de variáveis e constantes. Durante a execução de um objeto-frame, à medida que as instruções vão sendo executadas, as variáveis e constantes nas listas vão sendo diretamente manipuladas, até o fim do bloco de instruções corrente. Para que essa manipulação seja possível, faz-se uso de diversas instruções de manipulação de memória.

Neste contexto, idealizou-se inicialmente que, assim que o processador identificasse tais instruções de manipulação de memória, o mesmo deveria se comunicar com o CFP, para que ele manipulasse as listas de variáveis conforme necessário, de forma que o objeto-frame em si seria mantido exclusivamente do lado do CFP, e enviaria-se apenas as instruções ao processador. Esta concepção inicial, entretanto, foi preterida. Apesar de funcional, isto exigiria inúmeras requisições ao CFP, o que deveras prejudicaria o desempenho do sistema em geral, uma vez que o processador seria forçado a ficar ocioso com uma frequência excessiva, à espera das respostas do CFP.

Posteriormente, chegou-se a um formato mais eficiente: identificou-se que a maioria das instruções de busca e atribuição de variáveis apresenta baixa complexidade, permitindo que o processador as execute diretamente. Com o objetivo de minimizar a necessidade constante de comunicação com o CFP, determinou-se que o objeto-frame deveria ser enviado ao processador, garantindo-lhe acesso às constantes e variáveis indispensáveis para a computação. Após a execução completa das instruções de um objeto-frame, o processamento retorna ao frame anterior. Essa é, portanto, a dinâmica definida para a comunicação com o processador, sendo imprescindível que qualquer processador desenvolvido para operar com este gerenciador adote tal modelo de funcionamento.

Em resumo, ao identificar a instrução *CALL_FUNCTION* (indicando uma troca

de objeto-frame), o processador deve enviar o objeto-frame atual ao CFP, assim como os parâmetros para a troca de contexto (uma vez que é necessário informar ao gerenciador qual objeto-código da lista *CO_CONSTS* deve entrar em execução). O CFP, por sua vez, receberá o objeto-frame juntamente com os argumentos da instrução, atualizará sua versão local do objeto-frame, localizará o próximo frame requisitado e o devolverá ao processador para que a execução continue.

Os argumentos da instrução *CALL_FUNCTION* consistem em dois bytes numéricos. O primeiro byte indica a lista do objeto-frame, enquanto o segundo especifica o índice dentro dessa lista. O índice obtido aponta para um elemento da lista *CO_CONSTS*, que contém o próximo objeto-frame a ser carregado para execução. Como o primeiro byte é numérico, atribuiu-se números às listas de variáveis de um objeto-frame, relação que se encontra na Tabela 3. Assim, quando o processador identificar que o índice para o próximo objeto-frame se encontra na lista *CO_VARNAMES*, por exemplo, deve enviar o valor 2 como primeiro parâmetro da requisição *CALL_FRAME*.

Tabela 3 – Relação entre o valor numérico e lista de variáveis do objeto-frame

Valor numérico	**Lista de variáveis do objeto-frame**
0	*GLOBALS*
1	*CO_NAMES*
2	*CO_VARNAMES*
3	*CO_FREEVARS*
4	*CO_CELLVARS*

O segundo argumento reflete diretamente a posição desejada no vetor indicado, com o índice inicializado em zero, representando o primeiro elemento da lista. Para garantir o funcionamento correto desse mecanismo, o índice do próximo objeto-frame deve, de fato, corresponder à posição especificada pelo processador.

Contudo, esse vínculo não é inerente ao comportamento dos bytecodes. Na Máquina Virtual Python, o objeto-código é inserido diretamente na pilha e, em seguida, transformado em um objeto-frame dinâmico pela instrução *MAKE_FUNCTION*. Após essa conversão, o frame é armazenado em um vetor pré-definido, por meio do bytecode *STORE_NAME*.

Esse modelo, no entanto, não é viável em um processador Python implementado em hardware, devido à ausência das abstrações oferecidas pela máquina virtual. Como solução alternativa, ao encontrar a instrução *MAKE_FUNCTION*, o processador deverá armazenar apenas o índice do objeto-código na lista de constantes atual, em vez de criar imediatamente o objeto-frame. Essa abordagem simplificada fornece os argumentos

necessários para a instrução *CALL_FUNCTION* no CFP, garantindo a continuidade da execução sem perdas significativas de funcionalidade (vide Tabela 2).

De forma complementar ao empilhamento de objetos-frame resultante da execução da instrução *CALL_FUNCTION*, é necessário voltar ao objeto-frame anterior quando a execução do frame atual finaliza. O momento em que isto acontece é identificado pela instrução *RETURN*. Quando o processador identificar esta instrução, ele deve enviá-la ao CFP, para que o mesmo destrua o objeto-frame corrente, e volte ao anterior. Ao voltar ao anterior, o CFP o envia ao processador.

É pertinente destacar que a troca de objetos-frame durante a execução é gerenciada por meio de uma estrutura de pilha, que representa a pilha de execução das funções do programa. Dessa forma, quando se afirma que um novo objeto-frame foi carregado e está em execução, isso significa que ele foi empilhado no topo da pilha. Analogamente, ao retornar ao objeto-frame anterior, o CFP realiza uma operação de *POP*, removendo o objeto-frame do topo da pilha de execução. Mais sobre esse funcionamento será comentado na seção 4.3, onde se entrará em mais detalhes sobre o funcionamento do CFP.

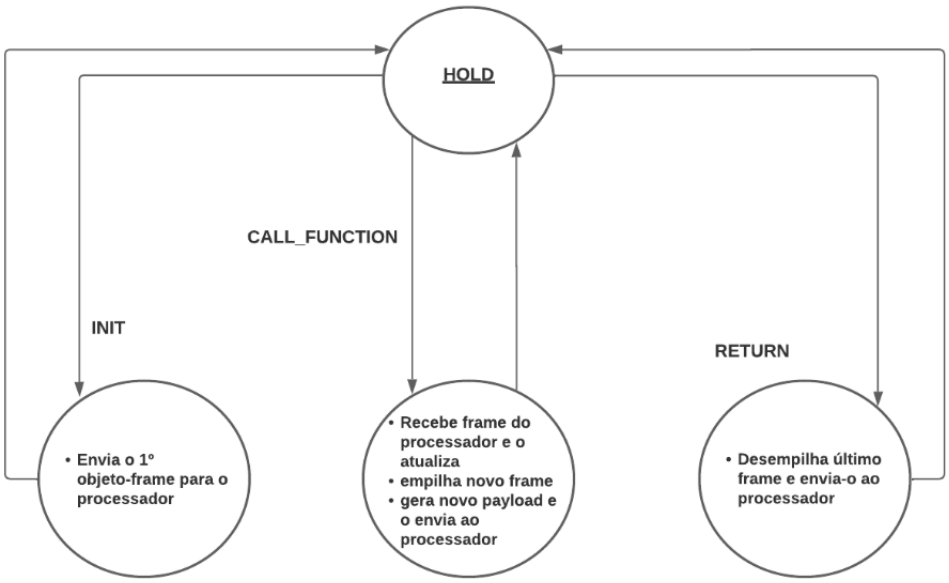
Além de tratar os bytecodes *CALL_FUNCTION* e *RETURN*, é necessário que o processador informe ao CFP quando o processamento irá iniciar. Assim, o processador deve enviar uma requisição *INIT* ao CFP, para que este envie o primeiro objeto-frame do programa. Observa-se que esta requisição, contudo, não faz parte do conjunto de bytecodes Python, sendo portanto uma instrução idealizada apenas no contexto do processador e do CFP.

É pertinente informar que o envio e recebimento de objetos-frame entre o processador e o CFP são realizados mediante codificação e decodificação. O formato exato desses processos será descrito a seguir. Ademais, o comportamento do CFP, conforme descrito até o momento, é ilustrado na Figura 16.

4.1.3 Padrões de comunicação com o processador

A proposta de comunicação entre o processador Python e o CFP é mais simples possível: o CFP mantém-se em espera ocupada, aguardando instruções do processador. Ao receber uma instrução, envia um sinal *ASCII ACK (ACKNOWLEDGE)*, sinalizando que está disponível e irá processar a requisição, e então realiza o processamento interno, enviando o resultado ao processador. Este processo é ilustrado na Figura 17.

Figura 16 – Comportamento do CFP



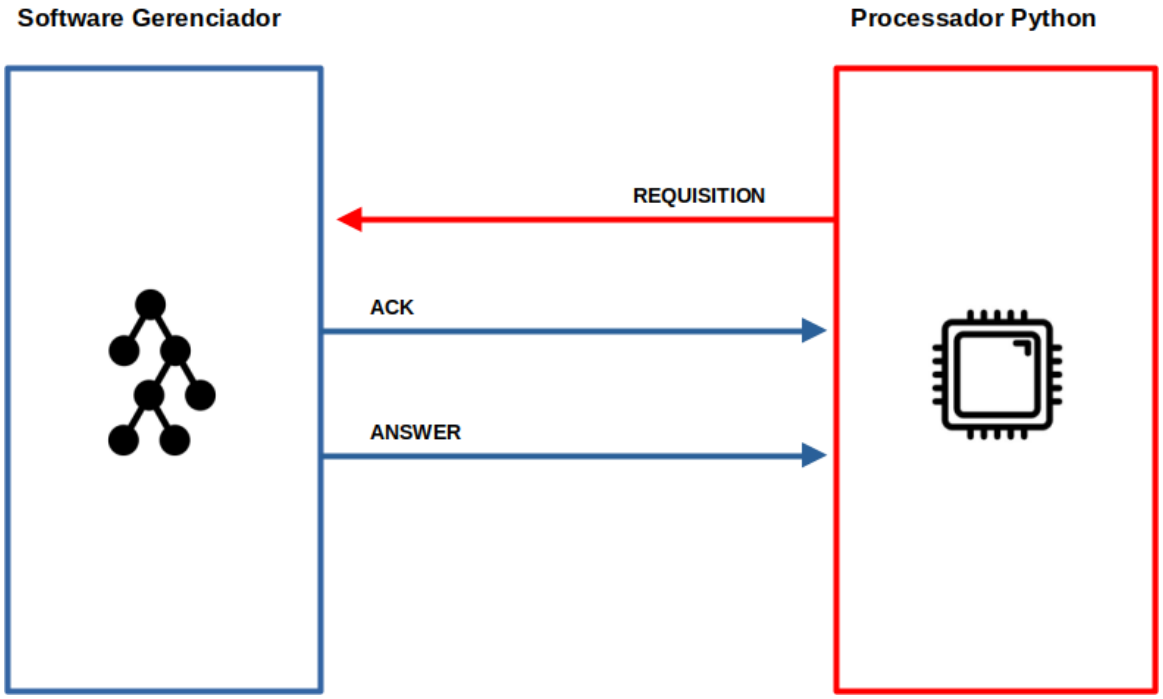
Fonte: Autor (2024)

Como mencionado na seção anterior, a troca de objetos-frame entre o CFP e o processador envolve processos de codificação e decodificação. Para desenvolver esses formatos, considerou-se a importância de preservar a ordem das variáveis e das listas de constantes, já que os argumentos dos bytecodes de manipulação de memória dependem dos índices dessas listas. Além disso, os tipos dos dados precisam ser enviados ao processador, pois são fundamentais para a execução correta das operações, haja vista que Python é uma linguagem fracamente tipada (IKE-NWOSU, 2018).

Por exemplo, a operação de soma varia conforme os operandos: no caso de strings, ocorre uma concatenação, enquanto que, com valores numéricos, realiza-se uma soma aritmética. O bytecode do Python não diferencia essas situações, utilizando a mesma instrução, *BINARY_ADD*, para ambos os casos. Assim, é indispensável que os tipos básicos estejam disponíveis durante a execução das instruções. Tais tipos estão presentes nos objetos-código gerados pelo compilador, de forma que são transferidos para o JSON final. Para poder informar o tipo do dado ao processador, elaborou-se a Tabela 4, em que consta a padronização de um byte para cada tipo básico de dados, relação que será necessária para o tópico a seguir.

Para fins de nomenclatura, um objeto-frame codificado para comunicação entre o CFP e o processador, ou vice-versa, será denominado, doravante, como *payload*. O *payload* será considerado válido se atender às seguintes regras e convenções, definidas

Figura 17 – Comunicação entre CFP e processador



Fonte: Autor (2024)

Tabela 4 – Relação entre tipo de dado e código binário no *payload*

Tipo de dado	**Código binário**
Code	100
null	000
int	001
float	010
string	011
bool	101
custom	110
unknown	111

para cada situação:

- Envio: do CFP ao processador

    1. Devem constar, na seguinte ordem, as listas do objeto-frame: *CO_CODE, GLOBALS, CO_NAMES, CO_VARNAMES, CO_FREEVARS, CO_CELLVARS* e *CO_CONSTS*.
    2. Cada lista é separada pelo caractere de controle *ASCII* GS *(GROUP SEPARATOR)*.
    3. GS deve ser o primeiro e o último byte do *payload*.
    4. Após cada caractere GS, com exceção do primeiro e do último, segue-se um

inteiro informando a quantidade de elementos da lista atual.

5. Os elementos de cada lista devem ser precedidos por um byte que indica o tipo do elemento, conforme a Tabela 4.
6. Dentro de cada lista, o byte de tipo e o elemento devem ser separados pelo caractere de controle *ASCII* US *(UNIT SEPARATOR)*.
7. Elementos do tipo *CODE* não são incluídos no *payload*. Em seu lugar, utiliza-se o caractere de controle *ASCII NULL*, mantendo, contudo, o tipo *CODE*, conforme especificado na Tabela 4[4].
8. Listas sem elementos devem consistir apenas em um inteiro com o valor zero.
9. Inteiros são representados por 4 bytes.

- Recebimento: do processador ao CFP

1. Valem as mesmas regras para o envio, porém as listas *CO_CODE* e *CO_CONSTS* não devem estar presentes, pois são imutáveis e são pertinentes apenas ao processador.
2. Após cada caractere GS, o inteiro informando a quantidade de elementos do vetor não deve estar presente, uma vez que o CFP já possui esta informação.

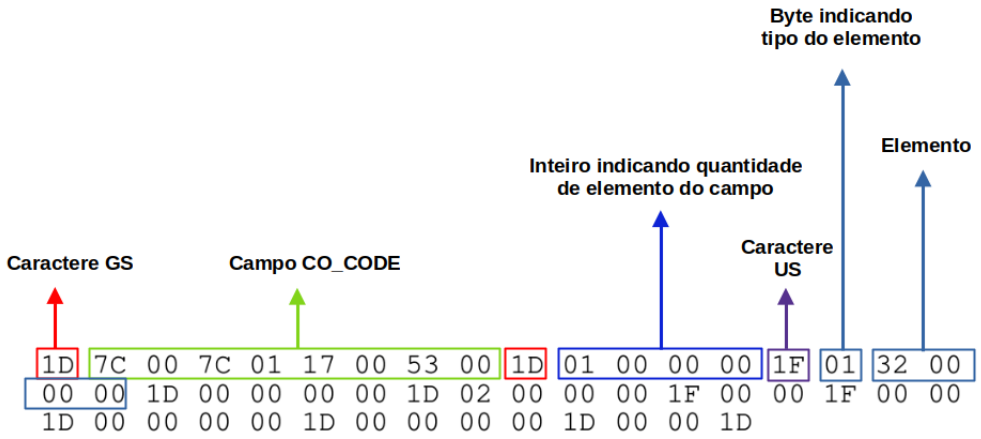
A Figura 18 apresenta um exemplo de *payload* de envio (fluxo do CFP ao processador) válido, destacando os caracteres de controle, os inteiros que indicam a quantidade de elementos em cada lista e os próprios elementos. Nela, o trecho sem marcações indica que a estrutura se repete para as demais listas. De forma análoga, a Figura 19 ilustra um *payload* de recebimento (fluxo do processador ao CFP) válido, enfatizando as listas relevantes.

4.2 Extração de dados do PYC

Como comentado, o primeiro passo para viabilizar um gerenciador de contexto de memória Python é extrair as informações que compõem o programa: seus dados e instruções, os quais encontram-se no arquivo PYC. Este arquivo é fruto da codificação realizada pelo módulo `Marhsal`, da biblioteca padrão da linguagem (ROSSUM; DRAKE, 2009). O módulo possui dois principais métodos: `dump` e `load`. O primeiro

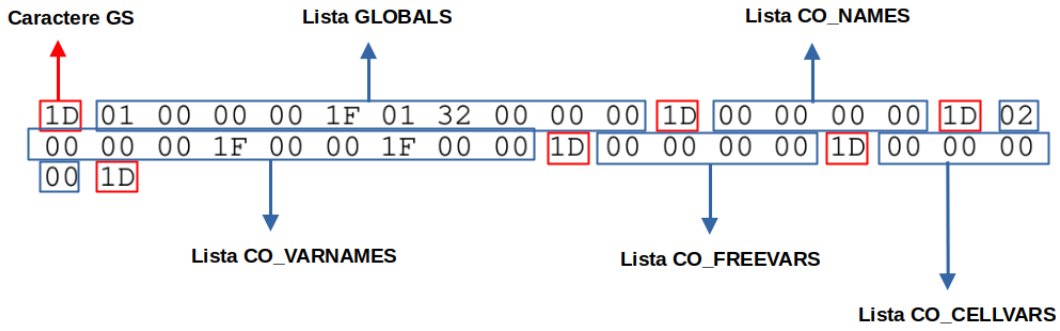
[4]Esta especificação é motivada porque elementos *CODE* costumam ter aninhamentos profundos, o que tornaria o *payload* excessivamente extenso, e também porque a presença desses elementos não se faz, realmente, necessária no âmbito do processador, apenas do CFP.

Figura 18 – Exemplo de *payload* de envio (do CFP ao processador)



Fonte: Autor (2024)

Figura 19 – Exemplo de *payload* de recebimento (do processador ao CFP)



Fonte: Autor (2024)

é chamado ao final do processo de compilação, e tem como objetivo codificar os objetos-códigos aninhados gerados durante o processo, escrevendo o resultado final no arquivo PYC. De fato, esta codificação é um tanto obscura, sendo que a documentação explicita que a sua especificação não é divulgada propositalmente. Entretanto, a função `load` do mesmo módulo realiza o trabalho de decodificação. Ao invocá-la, a codificação é revertida, e se tem os objetos-códigos novamente, com suas estruturas manipuláveis. É esta função, portanto, que permite a extração dos dados necessários à execução do programa.

Portanto, através da função `load` foi possível escrever o utilitário *pyc_serializer.py*, ferramenta desenvolvida pelo autor, que implementa a etapa 1 da solução proposta (vide a Figura 15). Assim, o utilitário é capaz não apenas de serializar o arquivo PYC num arquivo JSON, como também exibe informações relevantes para o estudo dos bytecodes, mostrando a estrutura aninhada de objetos-code gerados pelo compilador, assim como decodifica suas instruções e exibe-as ao usuário. Estas opções

estão disponíveis ao usuário através dos parâmetros que o utilitário disponibiliza, por meio de linha de comando. A seguir, as funcionalidades proporcionadas pelo utilitário:

- **-p**: Ativa a exibição dos bytecodes contidos no arquivo de entrada, permitindo que o usuário visualize os objetos-códigos presentes do arquivo. Este parâmetro é útil para análise inicial do código compilado.
- **-d**: Detalha os opcodes e seus parâmetros associados, gerando uma saída desassemblada do código. Deve ser usado em conjunto com o parâmetro **-p**, pois depende da exibição dos bytecodes.
- **-s**: Ativa a funcionalidade de serialização, que converte o objeto de código extraído do arquivo `PYC` para um formato estruturado JSON. O arquivo gerado será salvo no local especificado pelo parâmetro **-o**.
- **-o output**: Define o nome e o caminho do arquivo onde o JSON gerado pela serialização será salvo. Caso não seja especificado, o padrão é `output.json`.
- **-h**: Exibe uma descrição dos parâmetros aceitos pela aplicação e como utilizá-los. Este parâmetro não executa nenhuma outra funcionalidade.
- **-i input_file.py**: Especifica o arquivo de código-fonte Python (`.py`) que será processado. É um parâmetro obrigatório e deve conter o caminho de um arquivo válido.

A Figura 20 mostra um trecho de um arquivo JSON, resultado do processamento de *pyc_serializer*.

`pyc_serializer.py` utiliza bibliotecas padrão do Python, como ‘marshal‘ para desserialização dos objetos de código e ‘json‘ para a serialização. A execução começa com a definição de variáveis globais que controlam o comportamento do utilitário, como o nome dos arquivos de entrada e saída e opções para serializar ou imprimir informações detalhadas do código. A função principal, ‘main‘, interpreta os argumentos de linha de comando usando ‘getopt‘, permitindo configurar o arquivo de entrada, o arquivo de saída e as opções de operação. Caso o arquivo de entrada seja válido e tenha a extensão .py, ele é compilado para gerar o correspondente PYC, que é então processado pela função ‘initialize_parser‘.

A função ‘initialize_parser‘ realiza a leitura do arquivo PYC, identificando inicialmente o *magic number*, que indica a versão do Python usada na compilação. Em versões mais recentes, ela também verifica o cabeçalho de hash e outros metadados do arquivo. Após processar essas informações, os dados binários do arquivo PYC são

Figura 20 – Arquivo JSON criado pela ferramenta *pyc_serializer*

Fonte: Autor (2024)

convertidos em um objeto de código Python utilizando ‘marshal.load‘. Dependendo das configurações especificadas, este objeto pode ser serializado para JSON, usando a função ‘create_dict‘, ou analisado detalhadamente para exibir os bytecodes e os elementos associados, como constantes, variáveis e nomes livres.

Para representar a estrutura do código de forma hierárquica no JSON, a função ‘create_dict‘ transforma recursivamente o objeto de código em um dicionário, incluindo informações como os bytecodes (convertidos em hexadecimal), as variáveis declaradas, as constantes, e quaisquer sub-objetos de código. Caso a opção de decodificação seja ativada, a função ‘show_code‘ utiliza o módulo ‘dis‘ para desassemblar os bytecodes, detalhando os opcodes e seus parâmetros. O utilitário também inclui uma função de ajuda (‘print_help‘), que explica os argumentos mencionados na anteriormente.

A execução completa segue um fluxo modular que combina a análise do arquivo PYC com a manipulação eficiente dos dados extraídos, oferecendo flexibilidade para estudos ou transformações dos objetos de código em Python.

4.3 Software CFP

O software foi escrito na linguagem de programação C++, escolhida pelas convenientes abstrações que oferece, como a manipulação de vetores e o suporte a estruturas de dados prontas, como pilhas e listas encadeadas, simplificando a implementação de funcionalidades complexas. Além disso, o C++ é uma linguagem reconhecida por sua alta performance e pela capacidade de ser compilada para um amplo leque de arquiteturas.

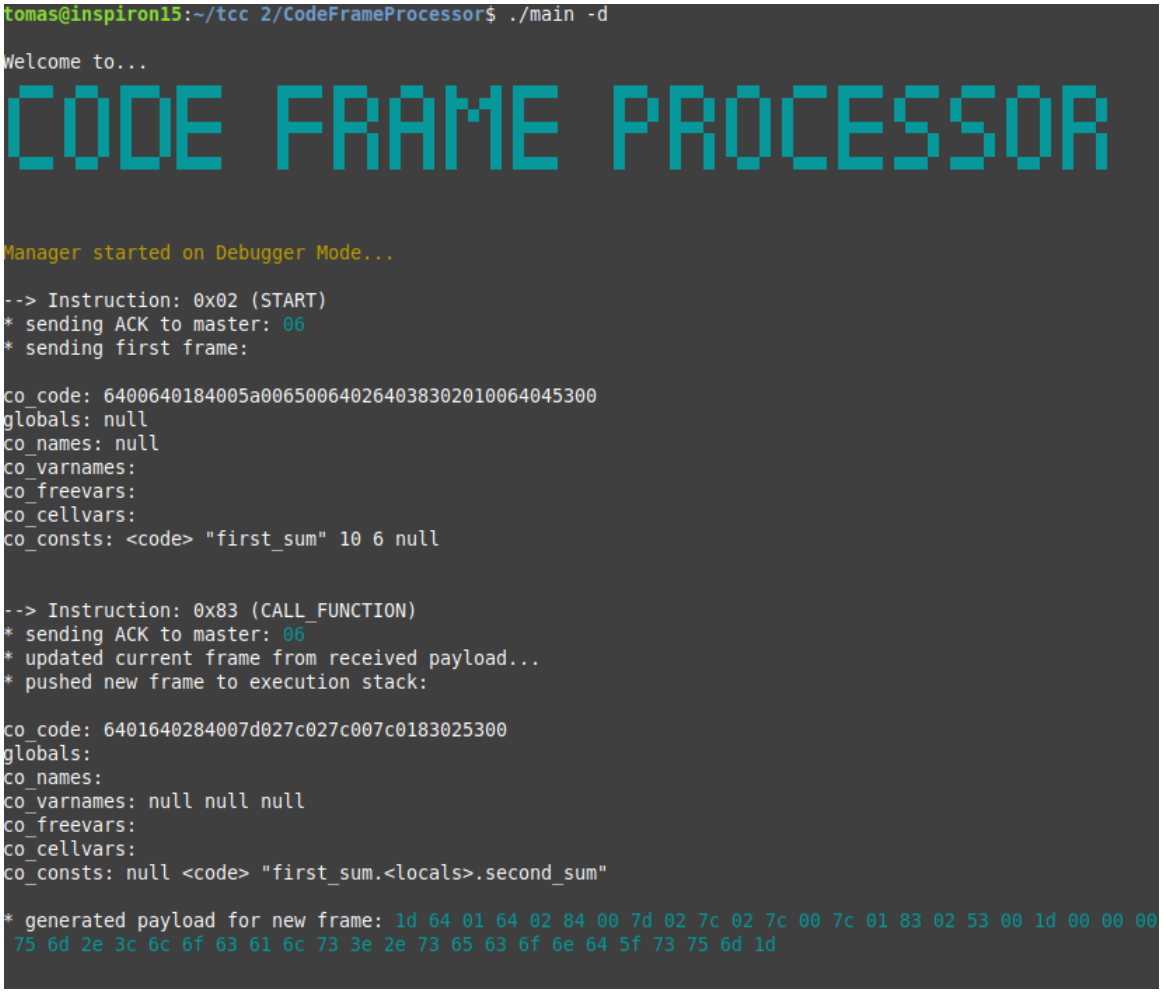
O CFP busca simular o funcionamento interno da Máquina Virtual Python, especificamente o gerenciamento dos objetos-frame provenientes do arquivo PYC. Para isso, foi criada a classe ‘Code‘, que representa os objetos-frame utilizados pela linguagem para armazenar informações sobre o estado de execução de uma função, incluindo variáveis, constantes e instruções. A implementação adota uma abordagem simplificada para reproduzir a manipulação de frames, permitindo executar instruções e alternar entre diferentes contextos de execução, de forma análoga ao comportamento da máquina virtual Python, conforme discutido no capítulo 3.

A Figura 21 mostra o CFP rodando em modo de depuração.

O código implementa uma aplicação destinada a interpretar e executar instruções provenientes de um arquivo binário, empregando um modelo de execução baseado nos objetos-frame da linguagem Python, os quais são representados pela classe ‘Code‘. O processo começa com a leitura de um arquivo JSON que descreve a estrutura do primeiro objeto-frame do programa. A partir desse arquivo, são criados objetos ‘Code‘ que armazenam informações relevantes, como constantes, nomes de variáveis e outros metadados. Durante a leitura, os vetores internos do objeto, como ‘co_names‘ e ‘co_varnames‘, são inicializados com tamanho fixo, e preenchidos com valores nulos, enquanto a lista ‘co_consts‘ é populado de acordo com os tipos dos valores presentes no JSON. Caso esses valores sejam objetos, a função faz uma chamada recursiva para construir objetos-frame aninhados. Salienta-se que os vetores internos (que correspondem às listas de variáveis de um objeto-frame) devem ser inicializados com valores nulos pois serão preenchidos à medida que as instruções do objeto-frame forem executada no processador.

Uma vez criado o objeto inicial, o programa carrega um arquivo binário contendo instruções organizadas em segmentos delimitados por sequências específicas de bytes (conforme será visto na seção seguinte). Cada segmento é interpretado como uma

Figura 21 – CFP rodando em modo de depuração



Fonte: Autor (2024)

instrução do processador que manipula os frames gerenciados pela aplicação. O processamento dessas instruções utiliza uma pilha de navegação, implementada pela classe ‘CodeNavigator‘, que mantém o controle sobre os frames em execução. Durante a execução, três tipos de instruções principais são suportados: inicialização de execução (‘INIT‘), chamada de função (‘CALL_FUNCTION‘) e retorno (‘RETURN‘). No caso de uma instrução de chamada, o programa atualiza o frame atual a partir de um *payload* recebido, que contém as informações necessárias para modificar os campos do objeto (que equivalem às listas de variáveis de um objeto-frame), e empilha um novo frame para execução. Quando uma instrução de retorno é recebida, o objeto-frame atual é desempilhado, e o programa retoma a execução do objeto-frame anterior.

Além disso, o código inclui funcionalidades de depuração que permitem acompanhar a execução das instruções em detalhes, exibindo o estado atual dos objetos-frame e os dados processados em formato hexadecimal. Por fim, o fluxo se

encerra após todas as instruções do arquivo binário serem processadas, garantindo que os frames sejam gerenciados de forma consistente ao longo da execução.

4.4 Utilitário de teste

Devido à dificuldade de lidar com os *payloads* binários descritos na seção 4.1.3 e à necessidade de criar entradas válidas para o CFP de maneira sistemática, desenvolveu-se um programa utilitário chamado `testPayload`. Esse programa não automatiza a criação de casos de teste, mas auxilia o usuário ao converter instruções especificadas diretamente no código em um arquivo binário pronto para ser utilizado como entrada pelo CFP.

O `testPayload` é um programa em C++ projetado para simplificar o processo de geração de arquivos binários formatados conforme a especificação do CFP. Ele processa instruções fornecidas pelo usuário na função ‘main‘ e as converte em *payloads* binários estruturados que simulam as requisições do processador. Este programa é particularmente útil devido à natureza específica e sensível do formato de entrada, reduzindo a probabilidade de erros na criação manual dos binários.

Para utilizá-lo, o usuário deve instanciar objetos da classe ‘Code‘ diretamente na função ‘main‘, configurar os valores desejados para cada objeto de acordo com os cenários de teste e invocar métodos específicos para gravar as instruções no arquivo binário. O programa cuida de toda a lógica de serialização, adicionando os delimitadores e opcodes necessários para garantir que o formato esteja de acordo com o protocolo definido na seção 4.1.3.

O fluxo típico do programa começa com a criação de um arquivo de saída, chamado `master_instructions.bin`. Inicialmente, o programa grava uma instrução de inicialização (‘INIT‘) no arquivo. Em seguida, o usuário especifica os objetos-frame e utiliza o método ‘generateInputTestPayload‘ para serializá-los no formato binário correto. Funções auxiliares como ‘generateCallFn‘ e ‘generateReturn‘ são usadas para gravar instruções específicas, como chamadas a funções (‘CALL_FUNCTION‘) ou retornos (‘RETURN‘).

Essas funções realizam a seguinte lógica:

- **‘generateCallFn‘**: Grava uma instrução ‘CALL_FUNCTION‘ no arquivo binário. Cada instrução inclui um opcode fixo, dois bytes que indicam o vetor de destino

e seu índice, o *payload* serializado do objeto ‘Code‘ e um delimitador fixo representado pelo vetor de caracteres ‘recordSeparator‘ `[0x03, 0x02]`.

- **‘generateReturn‘**: Grava uma instrução de retorno (‘RETURN‘), composta por um opcode fixo seguido pelo mesmo delimitador.

Ao final do programa, instruções de retorno adicionais podem ser adicionadas para completar o fluxo. Após gravar todos os dados, o utilitário verifica a integridade do arquivo binário gerado e exibe uma mensagem indicando o sucesso da operação.

A utilização de opcodes fixos e delimitadores bem definidos garante que o formato do arquivo seja interpretado corretamente pelo CFP, promovendo consistência e evitando ambiguidades. O design modular do código, com funções especializadas para cada tipo de instrução, facilita sua extensão e manutenção. Apesar de o programa não criar casos de teste automaticamente, sua função principal é fornecer uma ferramenta prática para gerar binários válidos de forma confiável e eficiente, permitindo ao usuário se concentrar nos cenários de teste sem se preocupar com os detalhes de formatação.

O resultado final é um arquivo binário estruturado que pode ser usado para testar o CFP em diversos cenários, garantindo a validade e a funcionalidade do sistema em conformidade com os padrões estabelecidos.

É importante reforçar que a verificação dos casos de teste gerados pelo utilitário `testPayload` é realizada de forma manual, cabendo ao usuário analisar os resultados e validar o comportamento do CFP com base nos cenários especificados. O papel do utilitário limita-se a auxiliar na conversão das instruções escritas pelo usuário para o formato binário exigido pelo CFP, automatizando apenas essa etapa do processo e reduzindo a possibilidade de erros na criação dos arquivos de entrada.

5 RESULTADOS E DISCUSSÕES

Este capítulo apresenta os resultados obtidos a partir da execução do teste do CFP e discute o comportamento observado. A seção 5.1 descreve o processo de execução do teste, detalhando os casos de teste e as etapas seguidas para validar o funcionamento do sistema. Em seguida, a seção 5.2 apresenta os resultados obtidos.

5.1 Descrição dos Casos de Teste

Para avaliar o funcionamento correto do CFP em relação aos comportamentos apresentados no capítulo anterior, foram elaborados casos de teste, descritos na Tabela 5.

Tabela 5 – Descrição dos casos de teste definidos

ID	Descrição do caso de teste
1	Verificar a atualização correta do objeto-frame atual a partir do *payload* recebido pelo processador.
2	Testar o comportamento do CFP em cenários de aninhamento profundo, com chamadas e retornos envolvendo 10 funções aninhadas, verificando a navegação e execução em estruturas complexas.
3	Avaliar o funcionamento de variáveis globais entre dois objetos-frame, verificando que os dados compartilhados são acessados e manipulados de forma consistente.
4	Verificar se o *payload* enviado está correto e formatado de acordo com a especificação vista na seção 4.1.3.

Em todos os casos de teste, o processo de execução segue o mesmo fluxo: a partir dos códigos-fonte, os dados e instruções do programa são extraídos para um arquivo JSON por meio da ferramenta *pyc_serializer* (vide seção 4.2). Em seguida, o *payload* de recebimento é gerado utilizando a ferramenta *testPayload* (vide seção 4.4), sendo gravado no arquivo que servirá como fonte de instruções para o CFP. No formato atual, o CFP é configurado para localizar o arquivo de instruções automaticamente. Assim, sua execução foi realizada por meio de linha de comando, utilizando o parâmetro de depuração (*-d*), permitindo o acompanhamento dos estados dos objetos-frame no código. Cabe ressaltar que as verificações realizadas são de natureza manual, e o ambiente de teste sobre o qual o CFP foi executado consiste no sistema operacional Linux.

5.2 Execução dos Testes

Os programas Python desenvolvidos para os casos de testes objetivam ser o mais diretos possível, dado que a meta principal é avaliar os pontos destacados na Tabela 5. Programas mais extensos ou com manipulações mais complexas, embora tecnicamente viáveis, não contribuiriam para o objetivo proposto de verificar o correto funcionamento do CFP.

A subseções seguintes comentam cada caso de teste e indicam os resultados obtidos para cada um.

5.2.1 Caso de Teste 1

Este teste avalia se o CFP está atualizando o objeto-frame atual de forma correta, com base no *payload* de recebimento. O objetivo é verificar se os tipos básicos (vide Tabela 4) estão sendo manipulados da forma correta para as listas de variáveis do objeto-frame atual.

Assim, através do utilitário *testPayload*, criou-se um *payload* de recebimento com valores arbitrários, com tipos variados. Espera-se que os valores sejam carregados para o objeto-frame corrente de forma correta, com os tipos especificados. A Figura 22 ilustra os valores inseridos no *Payload*.

Figura 22 – *Payload* criado para o Caso de Teste 1

```
// primeiro frame
Code codeObj, childCode;

Code::globals = {10, 20, "two", childCode, true};
codeObj.setCoNames(std::vector<VarType>{10, 20, "two", childCode, true});
codeObj.setCoVarnames(std::vector<VarType>{10, 0, "two", childCode, true});
codeObj.setCoFreevars(std::vector<VarType>{50, 60, "three", childCode, false});
codeObj.setCoCellvars(std::vector<VarType>{70, 80, "four", childCode, true});
generateCallFn(outfile, codeObj, 2, 1);

// retorna ao original
generateReturn(outfile);
```

Fonte: Autor (2024)

O resultado do teste está na Figura 23. É possível observar que o objeto-frame foi atualizado com os valores corretos, conforme os estabelecidos na Figura 22. Reforça-se que, conforme visto na seção 4.1.2, objetos-frame aninhados não são enviados ao

processador, de forma que, em seu lugar, envia-se o valor *null* (embora mantenha-se o tipo *Code* no *payload* de envio, para que o processador tenha conhecimento do tipo correto). Da mesma forma, valores booleanos são diretamente convertidos para numéricos.

Figura 23 – Resultado para o Caso de Teste 1

Fonte: Autor (2024)

5.2.2 Caso de teste 2

Para avaliar o comportamento do CFP em cenários de aninhamento profundo — isto é, quando várias funções estão aninhadas umas dentro das outras — foi elaborado o código apresentado na Figura 24. Esse código contém 10 chamadas de função e, consequentemente, 10 retornos. O objetivo deste teste é verificar se o CFP processa todas as instruções de *CALL_FUNCTION* e *RETURN* corretamente, sem erros, executando as ações necessárias durante o processo, como o envio do sinal *ACK* ao receber a instrução e a navegação precisa pelos objetos-frame.

A Tabela 6 apresenta os resultados do respectivo teste. A Figura 25 mostra um trecho da saída do CFP durante a execução do teste, a qual foi utilizada para obter os resultados do teste.

Figura 24 – Código para o Caso de Teste 2

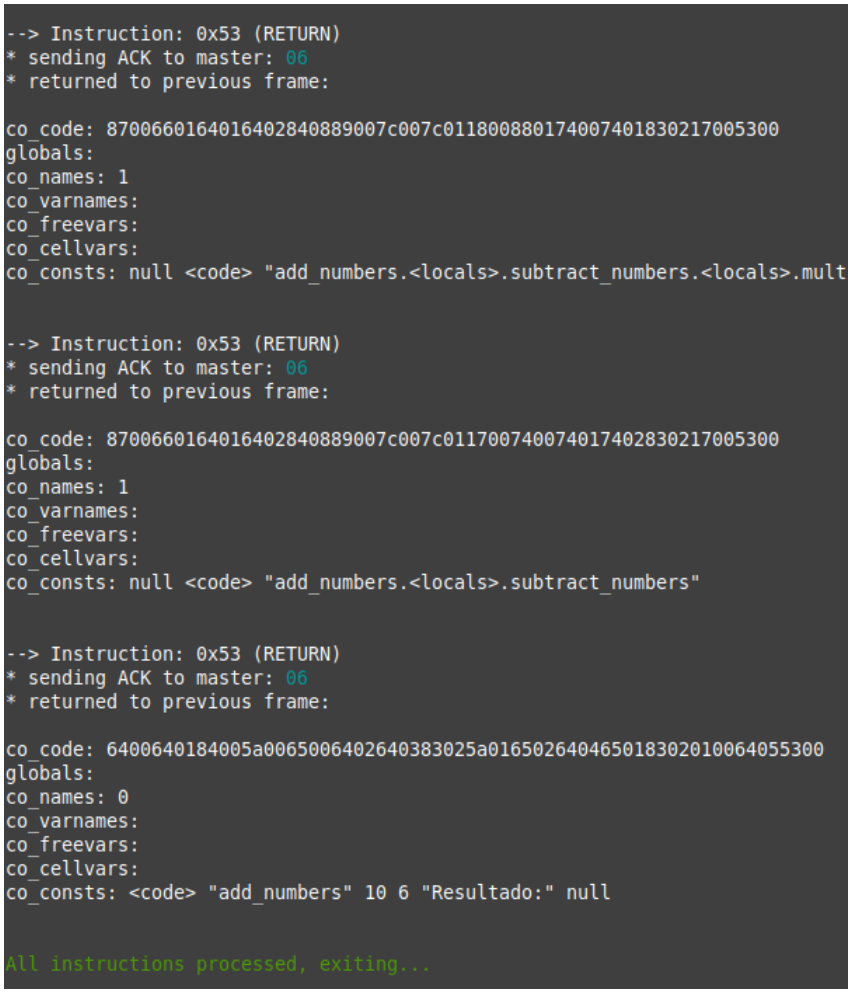
```python
def add_numbers(x, y):
    def subtract_numbers(a, b):
        def multiply_numbers(c, d):
            def divide_numbers(e, f):
                def modulus_numbers(g, h):
                    def power_numbers(i, j):
                        def floor_divide_numbers(k, l):
                            def absolute_difference(m, n):
                                def min_numbers(o, p):
                                    def max_numbers(q, r):
                                        return max(q, r)
                                    return min(o, p) - max_numbers(q, r)
                                return abs(m - n) + absolute_difference(o, p)
                            return k // l + floor_divide_numbers(m, n)
                        return i ** j + power_numbers(k, l)
                    return g % h + modulus_numbers(i, j)
                return e / f + divide_numbers(g, h)
            return c * d + multiply_numbers(e, f)
        return a - b + subtract_numbers(c, d)
    return x + y + add_numbers(a, b)


result = add_numbers(10, 6)
print("Resultado:", result)
```

Fonte: Autor (2024)

Tabela 6 – Resultados para o Caso de Teste 2

Índice	**Instrução Executada**	**ACK Enviado**	**Resultado Esperado**
1	*CALL_FUNCTION*	Sim	Sim
2	*CALL_FUNCTION*	Sim	Sim
3	*CALL_FUNCTION*	Sim	Sim
4	*CALL_FUNCTION*	Sim	Sim
5	*CALL_FUNCTION*	Sim	Sim
6	*CALL_FUNCTION*	Sim	Sim
7	*CALL_FUNCTION*	Sim	Sim
8	*CALL_FUNCTION*	Sim	Sim
9	*CALL_FUNCTION*	Sim	Sim
10	*CALL_FUNCTION*	Sim	Sim
11	*RETURN*	Sim	Sim
12	*RETURN*	Sim	Sim
13	*RETURN*	Sim	Sim
14	*RETURN*	Sim	Sim
15	*RETURN*	Sim	Sim
16	*RETURN*	Sim	Sim
17	*RETURN*	Sim	Sim
18	*RETURN*	Sim	Sim
19	*RETURN*	Sim	Sim
20	*RETURN*	Sim	Sim

Figura 25 – Trecho da saída do CFP ao executar Caso de Teste 2



Fonte: Autor (2024)

5.2.3 Caso de teste 3

Este caso de teste tem como objetivo verificar se o comportamento das variáveis globais está correto. Em programas Python, as variáveis globais permanecem constantes para os diferentes objetos-frame. Assim, durante uma troca de objeto-frame, a lista *GLOBALS* deve permanecer inalterada (nos casos em que não há instruções que a modifiquem). Contudo, caso um objeto-frame altere a lista, essa modificação deve ser refletida nos demais objetos-frame da aplicação.

Assim, para executar o teste, seguiu-se o seguinte fluxo:

1. Parte-se de um objeto-frame com todos os valores para *GLOBALS* nulos.
2. *GLOBALS* é atualizada com uma lista de valores.
3. Insere-se uma instrução *CALL_FUNCTION* na carga de instruções, implicando numa troca de objetos-frame.

4. Verifica-se que *GLOBALS*, no novo objeto-frame possui os mesmo valores que para o objeto-frame anterior.
5. Volta-se ao objeto-frame anterior, através da instrução *RETURN*.
6. Verifica-se que *GLOBALS* contém os mesmos valores configurados no início do teste.

A Figura 26 mostra a carga de instruções utilizada para implementar este caso de teste, e os resultados são mostrados na Figura 27. Verifica-se que o resultado obtido está correto, de acordo com o esperado.

Figura 26 – Carga de instruções para realizar o Caso de Teste 3

```
Code codeObj;

// Configura payload de recebimento e injeta CALL_FUNCTION
Code::globals = {10, 0, "two", "three", true};
codeObj.setCoNames(std::vector<VarType>{});
codeObj.setCoVarnames(std::vector<VarType>{});
codeObj.setCoFreevars(std::vector<VarType>{});
codeObj.setCoCellvars(std::vector<VarType>{});
generateCallFn(outfile, codeObj, 0, 1);

// retorna ao original
generateReturn(outfile);
```

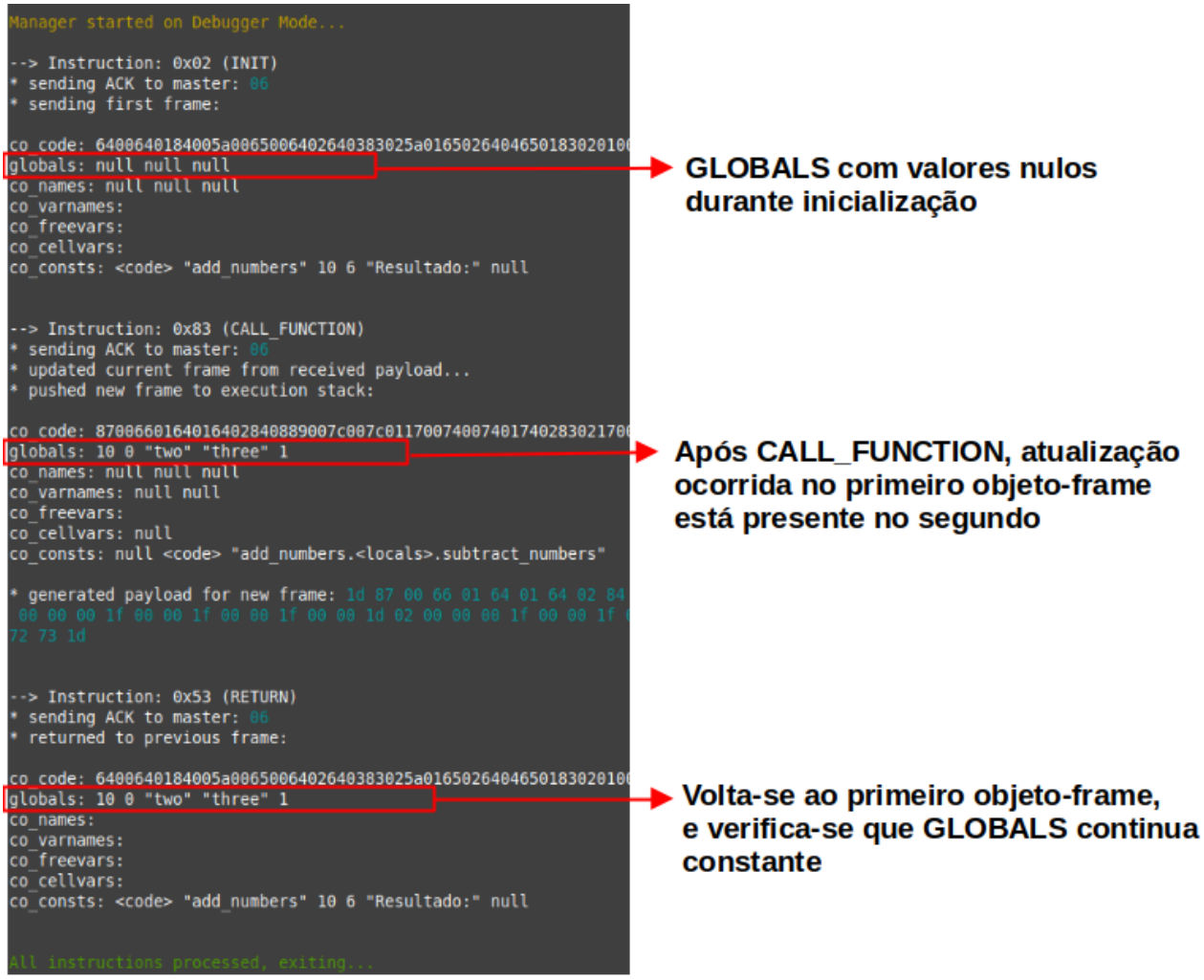
Fonte: Autor (2024)

5.2.4 Caso de Teste 4

O objetivo deste caso de teste é verificar se o *payload* de envio ao processador está correto. Assim, a Figura 28 ilustra a carga de instruções utilizada para verificar o *payload*, o qual espera-se que esteja de acordo com as regras estabelecidas na seção 4.1.3.

A Figura 29 mostra o *payload* gerado para o objeto-frame mostrado na Figura 28, e a Tabela 29 mostra os resultados do teste. Verifica-se que todas as especificações esperadas foram atingidas com sucesso, e que o CFP está, de fato, gerando os *payloads* de envio corretos.

Figura 27 – Verificação de resultados para o Caso de Teste 3



Fonte: Autor (2024)

Tabela 7 – Resultados para o Caso de Teste 4

Regra de *payload* de envio	**De acordo**
1	Sim
2	Sim
3	Sim
4	Sim
5	Sim
6	Sim
7	Sim
8	Sim
9	Sim

Figura 28 – Carga de instruções para o Caso de Teste 4

```
// Configuração de payload de recebimento e injeção de CALL_FUNCTION
Code::globals = {0, 20};
codeObj.setCoNames(std::vector<VarType>{"two", true});
codeObj.setCoVarnames(std::vector<VarType>{nullptr, 3});
codeObj.setCoFreevars(std::vector<VarType>{});
codeObj.setCoCellvars(std::vector<VarType>{});
generateCallFn(outfile, codeObj, 0, 0);

// retorna ao original
generateReturn(outfile);
```

Fonte: Autor (2024)

Figura 29 – *Payload* gerado para o objeto-frame

```
* generated payload for new frame: 1d 87 00 66 01 64 01 64 02 84 08 89 00 7c 00 7c 0
1 17 00 74 00 74 01 74 02 83 02 17 00 53 00 1d 02 00 00 00 1f 01 00 00 00 00 1f 01 1
4 00 00 00 1d 03 00 00 00 1f 00 00 1f 00 00 1f 00 00 1d 02 00 00 00 1f 00 00 1f 00 0
0 1d 00 00 00 00 1d 01 00 00 00 1f 00 00 1d 00 00 1f 04 00 1f 03 61 64 64 5f 6e 75 6
d 62 65 72 73 2e 3c 6c 6f 63 61 6c 73 3e 2e 73 75 62 74 72 61 63 74 5f 6e 75 6d 62 6
5 72 73 1d
```

Fonte: Autor (2024)

5.3 Discussão

Os resultados obtidos nos testes realizados demonstram que o CFP desenvolvido atende corretamente aos requisitos propostos, exibindo o comportamento esperado em todos os cenários testados. A seguir, discutimos os principais pontos observados em cada caso de teste e suas implicações para a funcionalidade do sistema como um todo.

5.3.1 Caso de Teste 1: Atualização de Objetos-Frame

No primeiro caso de teste, verificou-se que o CFP atualiza corretamente os valores do objeto-frame atual com base no *payload* recebido. A correspondência entre os valores definidos no *payload* e os armazenados no objeto-frame foi consistente em todas as execuções.

- Essa funcionalidade é essencial para garantir que a manipulação dos dados seja confiável.
- Os objetos-frame refletem fielmente os valores fornecidos pelo processador.

O sucesso deste teste valida a implementação dos mecanismos de decodificação e atribuição de dados no CFP.

5.3.2 Caso de Teste 2: Aninhamento Profundo

Os testes em cenários de aninhamento profundo mostraram que o CFP é capaz de lidar com chamadas e retornos de funções de forma robusta, mesmo em estruturas complexas com múltiplos níveis de aninhamento.

- O envio correto dos sinais *ACK* foi observado.
- A manipulação correta da pilha de objetos-frame confirmou a estabilidade e a integridade do sistema.

Esse comportamento é fundamental para garantir a escalabilidade do CFP, especialmente em aplicações que demandam suporte a fluxos de execução mais intrincados.

5.3.3 Caso de Teste 3: Persistência e Modificação de Variáveis Globais

A consistência das variáveis globais entre diferentes objetos-frame foi validada. Os resultados indicaram que:

- As mudanças realizadas em um objeto-frame são corretamente refletidas nos demais.
- Na ausência de alterações explícitas, os valores das variáveis globais permanecem inalterados ao longo da execução.

Este resultado demonstra que o CFP respeita as regras de escopo e compartilhamento de dados inerentes à execução de programas Python, garantindo compatibilidade com o comportamento esperado.

5.3.4 Caso de Teste 4: Formatação de Payloads de Envio

A conformidade dos *payloads* de envio com as especificações estabelecidas foi comprovada.

- O CFP segue corretamente os padrões de formatação exigidos pelo processador.
- A codificação de dados para diferentes tipos de variáveis foi validada.

Esse resultado é crucial, pois assegura que os dados enviados ao processador são

interpretáveis e consistentes, minimizando a possibilidade de erros de comunicação entre os sistemas.

5.4 Considerações Gerais

Os resultados obtidos demonstram que o CFP atende aos requisitos funcionais propostos, apresentando comportamento consistente e alinhado às expectativas. Todos os testes realizados foram concluídos com sucesso, indicando que o sistema é robusto e capaz de operar de forma confiável em cenários variados, desde estruturas simples até configurações mais complexas.

5.4.1 Possíveis Melhorias Futuras

- **Automatização de Testes:** Atualmente, as verificações são realizadas manualmente, o que pode ser propenso a erros humanos e consome tempo. A implementação de um framework de testes automatizados poderia aumentar a eficiência e a confiabilidade do processo de validação.
- **Teste em Ambientes Reais:** Embora os testes tenham sido realizados em ambiente controlado no Linux, seria interessante verificar o comportamento do CFP em condições reais de uso, como a integração com o processador em hardware.
- **Avaliação de Desempenho:** Não foram realizados testes para medir desempenho e eficiência, a fim de auferir parâmetros como tempo de execução, consumo de memória ou capacidade de resposta em cenários de alta demanda. A ausência desses testes representa uma lacuna importante, uma vez que o desempenho é um critério crítico para a viabilidade do processador em hardware. A adoção de ferramentas específicas para benchmarking e monitoramento de recursos poderia auxiliar na identificação de gargalos e otimização de algoritmos, garantindo que o CFP atenda aos requisitos do sistema final.

Conclui-se que o CFP cumpre os objetivos estabelecidos, representando uma contribuição significativa para a gestão de contexto em sistemas baseados em bytecode Python. No entanto, melhorias em áreas como automação de testes, avaliação em ambientes reais e análise de desempenho podem consolidar ainda mais sua aplicabilidade em cenários práticos e sua robustez em sistemas críticos.

6 CONSIDERAÇÕES FINAIS

Este trabalho apresentou o desenvolvimento e a validação de um Gerenciador de Contexto de Memória para um processador de bytecode Python, nomeado de CFP (*Code Frame Processor*), com o objetivo de gerenciar a troca de objetos-frame e o fluxo de execução em sistemas baseados nesse modelo. A implementação foi projetada para operar de forma eficiente e robusta em um ambiente de comunicação, atendendo aos requisitos funcionais e estruturais estabelecidos.

Os resultados obtidos nos testes realizados indicaram que o CFP é capaz de:

- Atualizar corretamente os objetos-frame com base nos *payloads* recebidos, garantindo a consistência dos dados.
- Manipular cenários de aninhamento profundo de chamadas e retornos de funções, preservando a integridade da pilha de execução.
- Gerenciar variáveis globais de forma consistente, respeitando as regras de escopo e compartilhamento inerentes à execução de programas Python.
- Gerar *payloads* de envio em conformidade com as especificações do processador, assegurando uma comunicação confiável.

Além disso, o sistema foi projetado para operar em espera ocupada, permitindo integração futura com um processador físico em hardware, como requerido pelo escopo do projeto. Embora os testes tenham sido realizados em ambiente controlado no Linux, os resultados obtidos são promissores para a integração com ambientes reais.

Entre as principais contribuições deste trabalho, destaca-se o desenvolvimento de um sistema capaz de gerenciar com precisão e eficiência a troca de contexto em um ambiente de execução de bytecode Python. Isso demonstra a viabilidade de aplicar soluções de software para suportar arquiteturas específicas de hardware, expandindo as possibilidades de integração entre essas áreas.

No entanto, algumas limitações foram identificadas. O trabalho não incluiu a aferição de performance do sistema, o que seria essencial para avaliar sua eficiência em cenários de produção. Além disso, o suporte atual a instruções é limitado e não contempla funcionalidades avançadas, como aquelas relacionadas à orientação a objetos, a exemplo da instrução `LOAD_BUILD_CLASS`. A comunicação com o CFP também apresenta restrições: no estado atual, as instruções são recebidas exclusivamente através de arquivos binários, enquanto o sistema deveria operar com entrada e saída mapeadas

em memória, como previsto inicialmente. Por fim, a ausência de um *framework* de testes automatizados e a falta de validações em um ambiente de produção limitam o alcance das análises realizadas.

Esses pontos representam oportunidades claras para trabalhos futuros. Caso este projeto seja levado adiante, recomenda-se priorizar a inclusão de suporte a instruções de orientação a objetos, a integração de entrada e saída mapeadas em memória, e a implementação de um *framework* de testes automatizados para assegurar maior robustez. Além disso, uma análise detalhada da performance do sistema, tanto em termos de tempo de execução quanto de utilização de recursos, seria essencial para aprimorar a solução e garantir sua viabilidade em ambientes reais.

Por fim, conclui-se que o CFP alcançou os objetivos estabelecidos, apresentando um comportamento alinhado às expectativas e fornecendo uma base sólida para futuras evoluções e integrações.

REFERÊNCIAS

BELAID, M. et al. Potential and technical basis for utilising coal beneficiation discards in power generation by applying circulating fluidised bed boilers. **2nd international conference on chemical, ecology and environmental sciences**, p. 6, 2013.

BEZZAM, E. et al. pyffs: A python library for fast fourier series computation and interpolation with gpu acceleration. **SIAM Journal on Scientific Computing**, SIAM, v. 44, n. 4, p. C346–C366, 2022.

BITENCOURT, T. P. ImplementaÇÃo de um processador, em vhdl, para executar algoritmos escritos em python. Unipampa, v. 1, n. 8, p. 159, 2019.

BROUGH, D. B.; WHEELER, D.; KALIDINDI, S. R. Materials knowledge systems in python—a data science framework for accelerated development of hierarchical materials. **Integrating materials and manufacturing innovation**, Springer, v. 6, p. 36–53, 2017.

CRAIG, I. D. **Virtual Machines**. 1st edition.. ed. Springer, 2005. ISBN 1852339691; 9781852339692. Disponível em: libgen.li/file.php?md5=8dc93614a5010b100c5608dffce18bfc.

DOGARU, R.; DOGARU, I. Optimization of gpu and cpu acceleration for neural networks layers implemented in python. In: IEEE. **2017 5th International Symposium on Electrical and Electronics Engineering (ISEEE)**. [S.l.], 2017. p. 1–6.

DYER, R.; CHAUHAN, J. An exploratory study on the predominant programming paradigms in python code. In: **Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2022. (ESEC/FSE 2022), p. 684–695. ISBN 9781450394130. Disponível em: https://doi.org/10.1145/3540250.3549158.

FERREIRA, R.; MAGALHÃES, S. V. G. de; NACIF, J. A. Métricas e números: Desmistificando a programação de alto desempenho em gpu. **Sociedade Brasileira de Computação**, p. 30, 2019.

IKE-NWOSU, O. **Inside The Python Virtual Machine**. 1st. ed. [S.l.]: O'Reilly Media, 2018. v. 1. (Learning Python, v. 1). ISBN 978-1491921977.

LAPLANTE, P. A. **Dictionary of Computer Science, Engineering and Technology**. 1st ed. ed. Chapman and Hall;CRC, 2017. ISBN 0849326915; 9780849326912; 9781351830638; 1351830635. Disponível em: libgen.li/file.php?md5=c2fef4dc36c84134aba1e5698e913f73.

LYNE, C. **Remapping Python Opcodes**. 2023. Disponível em: https://medium.com/tenable-techblog/remapping-python-opcodes-67d79586bfd5.

MAIER, O. et al. Pyqmri: an accelerated python based quantitative mri toolbox. **Journal of Open Source Software**, v. 5, n. 56, p. 2727, 2020.

MATUSIAK, M. **python bytecode: object model**. 2023. Disponível em: https://matusiak.eu/numerodix/blog/2013/8/12/python-bytecode-object-model/.

MITTAL, S.; VETTER, J. S. A survey of methods for analyzing and improving gpu energy efficiency. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 47, n. 2, aug 2014. ISSN 0360-0300. Disponível em: https://doi.org/10.1145/2636342.

MULLER, G. et al. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. In: **COOTS**. [S.l.: s.n.], 1997. p. 1–20.

NEWHALL, T. K. **Performance Measurement of Interpreted, Just-in-time Compiled, and Dynamically Compiled Executions**. 1. ed. Madison, WI: University of Wisconsin–Madison, 1999. 2 p. ISBN 9937236.

O'CONNOR, J. M.; TREMBLAY, M. picojava-i: The java virtual machine in hardware. **iEEE Micro**, IEEE, v. 17, n. 2, p. 45–53, 1997.

PEREIRA, D.; AYCOCK, J. Instruction set architecture of mamba, a new virtual machine for python. University of Calgary, 2002.

POWER, R.; RUBINSTEYN, A. How fast can we make interpreted python? **arXiv preprint arXiv:1306.6047**, 2013.

ROSSUM, G. V.; DRAKE, F. L. **Python 3 Reference Manual**. Scotts Valley, CA: CreateSpace, 2009. ISBN 1441412697.

SAABITH, A.; FAREEZ, M.; VINOTHRAJ, T. Python current trend applications-an overview. **International Journal of Advance Engineering and Research Development**, v. 6, n. 10, 2019.

SCHOEBERL, M. Jop: A java optimized processor. In: SPRINGER. **On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops: OTM Confederated International Workshops, HCI-SWWA, IPW, JTRES, WORM, WMS, and WRSM 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings**. [S.l.], 2003. p. 346–359.

SCHOEBERL, M. A java processor architecture for embedded real-time systems. **Journal of Systems Architecture**, Elsevier, v. 54, n. 1-2, p. 265–286, 2008.

SCHOENHOLZ, S. S.; CUBUK, E. D. Jax md: End-to-end differentiable, hardware accelerated, molecular dynamics in pure python. **Journal of Chemical Physics**, AIP Publishing, v. 150, n. 24, p. 244110, 2019.

SRINATH, K. Python–the fastest growing programming language. **International Research Journal of Engineering and Technology**, v. 4, n. 12, p. 354–357, 2017.

STALLINGS, W. **Arquitetura e Organização de Computadores 8a Edição**. 8th. ed. [S.l.]: São Paulo: Prentice Hall do Brasil, 2010. (Computer Science Series). ISBN 978-8535237487.

SURBIRYALA, J.; RONG, C. Cloud computing: History and overview. **IEEE Access**, v. 7, p. 1–7, 2019.

TSAI, C.-J.; WU, T.-H.; SU, H.-C. Jaip-mp: A four-core java application processor. In: IEEE. **2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)**. [S.l.], 2015. p. 189–194.

TULCHAK, L.; MARCHUK, A. **History of python**. Tese (Doutorado) — BHTY, 2016.

WILSON, P. R. Uniprocessor garbage collection techniques. In: SPRINGER. **International Workshop on Memory Management**. [S.l.], 1992. p. 1–42.

YIYU, T. et al. A java processor with hardware-support object-oriented instructions. **Microprocessors and Microsystems**, Elsevier, v. 30, n. 8, p. 469–479, 2006.