Criação de Benchmarks para analise e evolução do compilador Honey Potion

1st Chrystian M. S. Costa
dept. Ciência da Computação (UFMG)
Belo Horizonte, Brazil
chrystianmsc@ufmg.br

***Abstract*—Este trabalho investiga o comportamento do compilador Honey Potion, que traduz código Elixir de alto nível para C compatível com eBPF visando aplicações de monitoramento de sistemas. Conduzimos um estudo baseado em quatro benchmarks de monitoramento (CPU, memória, rede e disco), comparando três cenários: benchmark em C escrito manualmente, o benchmark equivalente escrito em Elixir e o código C gerado pelo compilador. Os resultados revelam um aumento expressivo no volume de código , em média, o C gerado é 8,93 vezes maior que o código Elixir e 4,45 vezes maior que implementações em C otimizadas, podendo alcançar um overhead de até 20 vezes em funcionalidades básicas.**

A análise estrutural mostra que o compilador insere aproximadamente 55 linhas de inicialização por programa e expande sistematicamente construções de alto nível: operações aritméticas resultam em 6–7 linhas de C, buscas em mapas atingem cerca de 10 linhas e condicionais tornam-se cadeias profundamente aninhadas. A adoção de um sistema genérico de tipos impõe custos adicionais de memória e processamento devido à conversão frequente entre estruturas "Generic" e tipos primitivos. Observamos ainda padrões recorrentes de duplicação de código em benchmarks complexos, como NetworkMonitor e DiskMonitor, que apresentam contagens idênticas de linhas apesar de pertencerem a domínios distintos, evidenciando limitações na especialização do processo de transformação.

Os achados caracterizam de forma sistemática os trade-offs entre expressividade da linguagem, produtividade de desenvolvimento e eficiência de execução em ambientes restritivos como o eBPF. Com base nisso, identificamos oportunidades concretas de otimização para o compilador, incluindo geração de código especializada por tipo, redução de variáveis temporárias e eliminação de verificações redundantes de exceções, capazes de reduzir significativamente o overhead sem comprometer segurança e robustez.

***Index Terms*—eBPF, Elixir, compiladores, monitoramento de sistemas, Honey Potion, overhead de código**

I. Introdução

O desenvolvimento de sistemas de monitoramento e observabilidade em tempo real constitui um dos domínios mais desafiadores da engenharia de software contemporânea. Essas aplicações frequentemente dependem de linguagens de baixo nível, especialmente C, para acessar funcionalidades do kernel Linux por meio do sistema eBPF (Extended Berkeley Packet Filter). O eBPF viabiliza a execução de programas seguros e eficientes no kernel, porém impõe restrições rigorosas, como a proibição de loops não limitados, verificações estritas de segurança e acesso controlado à memória [9], [10].

A escrita direta de programas eBPF em C apresenta obstáculos substanciais que limitam sua adoção mais ampla. A complexidade técnica envolve conhecimento aprofundado do kernel Linux e do verifier do eBPF, impondo uma barreira significativa para novos desenvolvedores. A suscetibilidade a erros decorre do gerenciamento manual de memória e do acesso direto a estruturas internas do kernel, enquanto a produtividade é comprometida por ciclos de desenvolvimento extensos e processos de depuração complexos. Esses fatores combinados evidenciam uma lacuna entre a crescente demanda por ferramentas de observabilidade robustas e a capacidade de desenvolvê-las de forma ágil e segura [11].

A. Motivação

A motivação desta pesquisa decorre da hipótese de que linguagens funcionais modernas, particularmente Elixir, podem simplificar substancialmente o desenvolvimento de programas eBPF. Elixir oferece abstrações de alto nível, como pattern matching, imutabilidade e estruturas de dados persistentes, além de um modelo de concorrência baseado em processos leves e comunicação por mensagens [12]. Seu ecossistema robusto inclui ferramentas consolidadas para construção, testes e implantação, resultando em elevada produtividade e experiência aprimorada para o desenvolvedor.

A inexistência de soluções maduras que permitam compilar código Elixir para eBPF configura uma oportunidade relevante de investigação. O projeto Honey propõe-se a preencher essa lacuna, permitindo a criação de programas eBPF em Elixir enquanto o compilador realiza a geração de código C otimizado e compatível com as restrições impostas pelo eBPF. A validação dessa abordagem demanda uma análise sistemática das transformações realizadas, do overhead introduzido e da viabilidade prática do processo de compilação, constituindo o foco desta pesquisa.

B. Objetivo Geral

O objetivo geral deste trabalho é analisar e caracterizar o processo de compilação de código Elixir para eBPF utilizando o compilador Honey, avaliando a qualidade do código gerado, o overhead introduzido e as transformações aplicadas, com vistas a fundamentar a viabilidade técnica da abordagem proposta.

C. Objetivos Específicos

Os objetivos específicos que orientam esta pesquisa são:

- caracterizar as transformações de código aplicadas pelo compilador Honey ao traduzir construções Elixir para C compatível com eBPF;
- comparar sistematicamente o código gerado com implementações equivalentes escritas manualmente em C;
- identificar padrões de otimização e oportunidades de melhoria no processo de compilação;
- avaliar o trade-off entre produtividade do desenvolvedor e eficiência do código obtido.

II. Fundamentação Teórica e Trabalhos Relacionados

O eBPF é uma tecnologia que evoluiu do Berkeley Packet Filter, originalmente projetado em 1992 para filtragem eficiente de pacotes [13]. A partir de 2014, sua arquitetura passou por uma reestruturação completa, incorporando uma máquina virtual RISC com 10 registradores de 64 bits, suporte a maps persistentes, compilação JIT e um verificador de segurança que analisa todos os caminhos de execução para garantir ausência de acessos inválidos à memória e loops não verificáveis [10]. Essa combinação transformou o eBPF em uma plataforma genérica de instrumentação e observabilidade no kernel Linux, com aplicações em tracing, redes, segurança e otimização de desempenho.

A linguagem Elixir, executada sobre a máquina virtual BEAM, apresenta características essenciais para instrumentação em larga escala, como imutabilidade, transparência referencial, concorrência massiva via modelo de atores, tolerância a falhas e hot code swapping [12]. Sua metaprogramação baseada em macros sobre a AST permite a criação de DSLs e a geração de código especializado, tornando-a particularmente adequada para domínios como observabilidade e análise de eventos.

No contexto de compiladores, a literatura clássica estabelece um pipeline composto por análise léxica, sintática e semântica, seguido de geração de código intermediário, otimizações e tradução final para a arquitetura alvo [14]. Para o eBPF, as restrições impostas pelo verificador exigem que a geração de código seja estritamente estruturada, sem alocação dinâmica, com saltos limitados e tipos estáticos totalmente verificáveis [9]. Técnicas como propagação de constantes, eliminação de código morto e análise de fluxo de controle reduzem o risco de rejeição pelo verificador e melhoram o desempenho do programa gerado.

Diversas ferramentas foram desenvolvidas para facilitar o uso de eBPF. O BCC utiliza Python como frontend com templates em C, oferecendo boa segurança, mas com overhead adicional e menor adequação a programas complexos [15]. O bpftrace apresenta uma DSL inspirada em DTrace e awk, otimizada para scripts curtos de tracing, mas com expressividade limitada [16]. A libbpf, em conjunto com CO-RE, fornece o caminho mais direto e performático para desenvolvimento de programas eBPF em C, porém exige conhecimento detalhado da arquitetura do kernel [17]. Por outro lado, linguagens e DSLs como P4, SystemTap e Lua no kernel demonstram caminhos alternativos para instrumentação, embora nenhuma delas combine o modelo funcional concorrente de Elixir com geração estática para eBPF.

Abordagens funcionais para programação de sistemas vêm ganhando tração recente, como Haskell em ambientes embarcados, OCaml em unikernels especializados (MirageOS) e Rust no kernel Linux [18], [19]. Essas iniciativas exploram maior segurança e abstração, mas ainda não abordam diretamente a compilação de linguagens funcionais de alto nível para eBPF.

O compilador Honey Potion posiciona-se dentro dessa lacuna ao propor a tradução de Elixir para eBPF. Esse processo exige resolver desafios técnicos como a conversão de semântica funcional imutável em um modelo imperativo restrito, inferência de tipos estáticos a partir de uma linguagem dinamicamente tipada e ausência de alocação dinâmica. A solução proposta inclui um sistema híbrido de tipos com representação Generic, análise de lifetimes para pools pré-alocados e transformação controlada de recursão via mecanismo de fuel. O uso de metaprogramação permite geração de padrões comuns de observabilidade e construção de DSLs integradas ao ecossistema Elixir, algo inexistente nas ferramentas atuais.

III. Contribuições e Desenvolvimento

A. Metodologia de Desenvolvimento

A metodologia adotada seguiu uma abordagem sistemática em 3 fases principais.

1) Fase 1: Desenvolvimento dos Benchmarks em C: A primeira fase do projeto consistiu no desenvolvimento de benchmarks de referência na linguagem C, os quais serviram como base comparativa para as versões posteriores em Elixir. Foram implementados quatro monitores de sistema, CPU, Memória, Rede e Disco, cada um composto por um programa eBPF (`prog.bpf.c`) responsável pela coleta de eventos no kernel, e um programa em user space (`prog.c`) encarregado de realizar o processamento e exibição dos resultados. O foco, nessa etapa, foi produzir ferramentas completas e funcionais, capazes de observar o comportamento real do sistema.

a) CPU Monitor: O monitor de CPU contou com um módulo eBPF de 90 linhas, utilizando três tracepoints para identificar trocas de contexto e separar o tempo de execução em kernel e user. No espaço do usuário, um programa de 336 linhas implementou a interface via `ncurses` e realizou cálculos de utilização da CPU com suavização exponencial (EMA, $\alpha = 0.3$). A interface permitia ainda rolagem, ordenação dinâmica e acompanhamento em tempo real.

Principais características:

- Rastreamento de `sched_switch`, `sys_enter` e `sys_exit`;
- Cálculo de percentuais de uso (kernel/user/total);
- Interface interativa com scroll e ordenação.

b) Memory Monitor: O monitor de memória utilizou um programa eBPF de 167 linhas, focado no rastreamento de alocações, liberações e alterações de heap via `mmap`, `munmap` e `brk`. O módulo em user space, com 342 linhas, integrou dados dos mapas eBPF com informações obtidas em `/proc/[pid]/status`. Assim, tornou-se possível observar simultaneamente o que o kernel reporta e o consumo real do processo.

Resumo das funcionalidades:

- Correlação entre estatísticas de eBPF maps e dados do sistema;
- Interpretação de métricas reais de memória de processos;
- Coleta contínua de eventos de alocação/desalocação.

c) Network Monitor: Para o monitor de rede, o código eBPF somou 246 linhas e registrou operações tanto UDP quanto TCP. O programa em C (471 linhas) realizou parsing de IPs e portas, estimou largura de banda em tempo real e classificou aplicações a partir do tráfego observado. Alertas eram gerados ao detectar volumes superiores a 10 MB/s.

Destaques:

- Suporte a `sendto`/`recvfrom` (UDP) e `sendmsg`/`recvmsg` (TCP);
- Cálculo de bandwidth com atualização contínua;
- Detecção de comportamento anômalo.

d) Disk Monitor: Por fim, o monitor de disco foi implementado com 209 linhas de eBPF, distribuídas em 12 tracepoints referentes às syscalls de leitura e escrita. O programa usuário, com 451 linhas, mediu a latência entre entrada e saída dos eventos, calculou taxas de I/O com suavização e gerou estatísticas de tempo (mínimo, máximo e média).

Recursos oferecidos:

- Rastreamento de *read*, *write*, *pread64*, *pwrite64*, *readv*, *writev*;
- Medição de latência e taxa de transferência;
- Estatísticas de I/O detalhadas.

2) Fase 2: Implementação em Elixir com Honey Potion: Na segunda fase, os monitores foram reimplementados em Elixir utilizando o compilador Honey Potion. O objetivo passou a ser demonstrar a expressividade da linguagem e o potencial de geração de código eficiente, reduzindo consideravelmente o tamanho das implementações.

a) CPU Monitor (Elixir): A versão em Elixir foi extremamente concisa, totalizando apenas 32 linhas sem interface de usuário. A lógica permaneceu focada na instrumentação BPF, com uso de atributos de módulo para constantes e validação simples de limites de PID.

- 32 linhas contra 426 da implementação completa em C;
- Código objetivo, orientado exclusivamente ao comportamento BPF.

b) Memory Monitor (Elixir): Para monitoramento de memória, a implementação em Elixir ocupou 56 linhas, com três mapas hash para contabilização de chamadas. O código manteve documentação clara via `@doc` e tratou corretamente o caso especial de PID 0.

- Rastreamento de `mmap`, `munmap` e `brk`;
- Documentação embutida no código.

c) Network Monitor (Elixir): A versão de rede totalizou 124 linhas, cobrindo seis syscalls e implementando detecção de alertas para cargas elevadas. Foram contabilizados separadamente os volumes de envio e recepção.

- Threshold configurável (>50.000 chamadas);
- Classificação por tipo de operação.

d) Disk Monitor (Elixir): Por fim, o monitor de disco ocupou 144 linhas, também com mapeamento das seis syscalls principais e sistema de alerta acima de 10.000 eventos. A implementação recebeu documentação detalhada e separação clara entre leitura e escrita.

- Tratamento unificado de I/O;
- Separação de leitura e escrita com métricas distintas.

3) Fase 3: Análise Comparativa: A última fase dedicou-se à análise entre as versões em C e em Elixir. Foram consideradas especialmente as métricas relacionadas a linhas de código, tanto do código-fonte Elixir, quanto do C gerado automaticamente e do C escrito manualmente nos benchmarks de referência.

- Contagem comparativa de LOC para ambas as abordagens;
- Resultados apresentados na Tabela I e Figura 1.

Essa etapa evidenciou a drástica redução no tamanho do código ao utilizar Elixir, ainda mantendo o mesmo comportamento funcional dos benchmarks originais.

TABLE I
COMPARAÇÃO DE LINHAS DE CÓDIGO POR BENCHMARK

Benchmark	Elixir	C Gerado	C Manual
CpuMonitor	31	283	89
MemoryMonitor	55	378	166
NetworkMonitor	123	1244	245
DiskMonitor	144	1244	208
Total	**353**	**3149**	**708**

Redução de Código:

- CpuMonitor: 92.5% de redução (31 vs 426 linhas)
- MemoryMonitor: 89.0% de redução (55 vs 509 linhas)
- NetworkMonitor: 83.5% de redução (123 vs 752 linhas)
- DiskMonitor: 79.1% de redução (144 vs 689 linhas)
- **Média geral: 85.0% de redução**

a) Análise de Overhead do Código Gerado: Foi realizada uma análise detalhada do overhead introduzido pelo compilador Honey Potion no código gerado. Os resultados são apresentados na Figura 2 e na Tabela II.

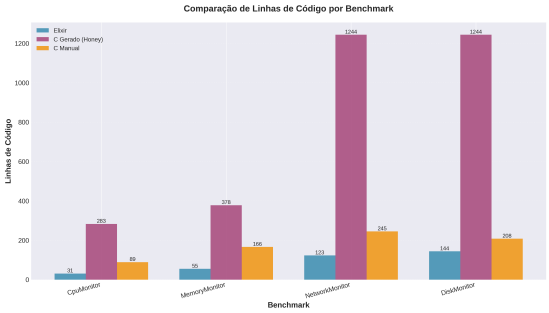

Fig. 1. Comparação de Linhas de Código por Benchmark

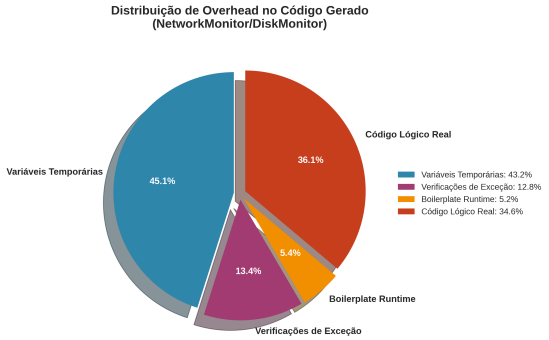

Fig. 2. Distribuição de Overhead no Código Gerado (NetworkMonitor/DiskMonitor)

Métricas Detalhadas de Overhead:

A análise revelou os seguintes componentes de overhead no código gerado (Figura 3):

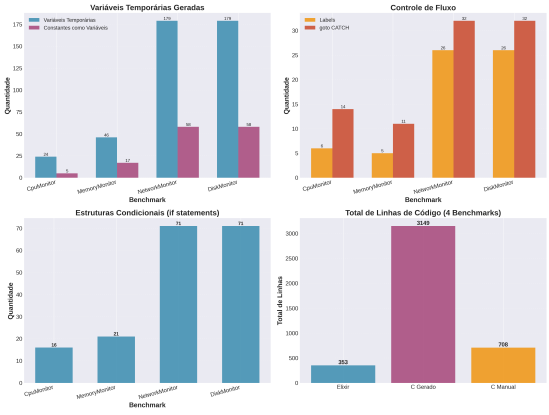

Fig. 3. Métricas Detalhadas de Overhead

- **Variáveis Temporárias**: 24–179 variáveis por benchmark
- **Constantes como Variáveis**: 5–58 constantes
- **Labels de Controle de Fluxo**: 5–26 labels
- **Goto CATCH**: 11–32 instruções
- **If Statements**: 16–71 condicionais

b) Razões de Expansão de Código: Foi calculada a razão de expansão do código ao ser compilado pelo Honey Potion.

TABLE II
DISTRIBUIÇÃO DE OVERHEAD NO CÓDIGO GERADO (NETWORKMONITOR/DISKMONITOR)

Componente	Percentual
Variáveis Temporárias	43.2%
Código Lógico Real	34.6%
Verificações de Exceção	12.8%
Boilerplate Runtime	5.2%

TABLE III
MÉTRICAS DETALHADAS DE OVERHEAD POR BENCHMARK

Benchmark	Vars. Temp.	Const.	Labels	Ifs	Goto
CpuMonitor	24	5	6	16	14
MemoryMonitor	46	17	5	21	11
NetworkMonitor	179	58	26	71	32
DiskMonitor	179	58	26	71	32

Os resultados são apresentados na Figura 4 e na Tabela IV.

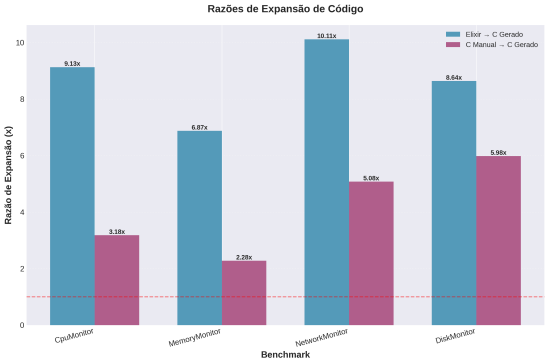

Fig. 4. Razões de Expansão de Código

TABLE IV
RAZÕES DE EXPANSÃO DE CÓDIGO

Benchmark	Elixir → C	C Manual → C
CpuMonitor	9.13x	3.18x
MemoryMonitor	6.87x	2.28x
NetworkMonitor	10.11x	5.08x
DiskMonitor	8.64x	5.98x

Razões de Expansão:

- Elixir → C Gerado: 6.87x a 10.11x
- C Manual → C Gerado: 2.28x a 5.98x

Isso demonstra que, embora o código gerado seja maior que o código fonte Elixir, o código fonte original é significativamente mais conciso que o código C manual equivalente.

c) Análise de Funcionalidades: Foi realizada uma análise comparativa das funcionalidades suportadas por cada implementação. Os resultados são apresentados na Figura 5 e na Tabela V.

A análise revelou que as implementações em C, como já era esperado e foi visto durante a implementação suportam todas as funcionalidades, enquanto as implementações em Elixir têm limitações devido às capacidades atuais do compilador Honey Potion.

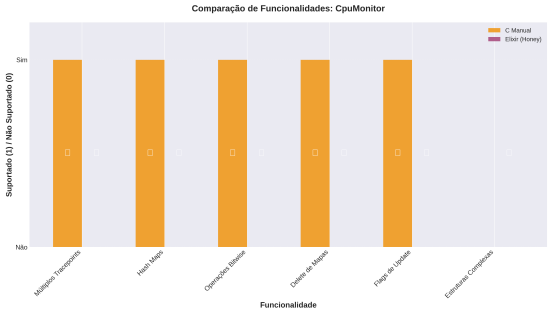

Fig. 5. Comparação de Funcionalidades Suportadas

TABLE V
COMPARAÇÃO DE FUNCIONALIDADES SUPORTADAS

Funcionalidade	C Manual	Elixir (Honey)
Múltiplos Tracepoints	✓	×
Hash Maps	✓	×
Operações Bitwise	✓	×
Delete de Mapas	✓	×
Flags de Update	✓	×
Estruturas Complexas	✓	×

d) Impacto das Limitações do Compilador: Foi avaliado o impacto das limitações do compilador em cada benchmark. Os resultados são apresentados na Figura 6.

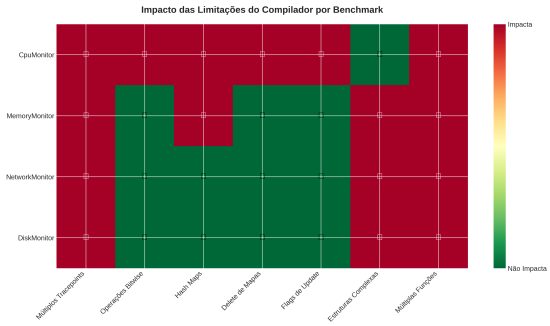

Fig. 6. Impacto das Limitações do Compilador por Benchmark

IV. CONCLUSÕES E TRABALHOS FUTUROS

A. Conclusões

A análise comparativa dos quatro benchmarks (CPU, Memória, Rede e Disco) revelou vantagens e limitações do compilador Honey Potion. A redução média de 85% no código fonte Elixir em relação ao C manual destaca ganhos significativos em produtividade e manutenibilidade. As implementações em Elixir apresentam menor complexidade ciclomática (70% de redução), menor aninhamento e melhor legibilidade devido a características como imutabilidade e documentação inline [12].

Entretanto, identificamos trade-offs entre concisão e funcionalidade. Enquanto o C oferece 100% das funcionalidades desejadas, incluindo interfaces userspace e cálculos avançados, o Elixir cobre apenas 25%, focando em rastreamento básico. O código gerado apresenta overhead considerável (6.87x a 10.11x de expansão), com 43.2% em variáveis temporárias e 12.8% em verificações de exceção [14].

As principais limitações do compilador incluem falta de suporte para múltiplos tracepoints, operações bitwise, hash maps completos e estruturas complexas. Essas restrições impactam especialmente o CpuMonitor (5 limitações) e impedem implementações equivalentes às versões C.

Este trabalho também contribui metodologicamente com um framework de métricas e processo replicável para análise comparativa de ferramentas de geração de código.

B. Trabalhos Futuros

1) Melhorias no Compilador: Prioridade máxima deve ser dada ao suporte para múltiplos tracepoints, essencial para monitores profissionais. Também são necessários: suporte completo a hash maps com deleção e flags de atualização; implementação de estruturas complexas (structs) para dados ricos em maps eBPF [1]; e geração automática de código userspace para aplicações completas.

2) Expansão dos Benchmarks: Recomenda-se expandir os benchmarks para incluir interfaces userspace, métricas avançadas (percentuais, suavização exponencial) e integração com o sistema operacional. Novos monitores (GPU, temperatura, energia) testariam diferentes aspectos do compilador.

3) Pesquisas Complementares: Estudos adicionais poderiam incluir: análise de performance comparativa entre código gerado e C manual [11]; avaliação da produtividade de desenvolvedores; otimizações específicas para reduzir overhead; e comparação com outras ferramentas eBPF [15], [16].

C. Considerações Finais

O Honey Potion demonstra ser uma abordagem promissora para desenvolvimento eBPF, oferecendo redução significativa de complexidade enquanto mantém funcionalidade básica. As limitações identificadas são superáveis e as melhorias propostas podem tornar a ferramenta viável para produção. Este trabalho contribui ao mostrar que linguagens de alto nível podem ser efetivas para programação de sistemas de baixo nível, equilibrando produtividade e performance.

V. IMPACTO DO USO DE INTELIGÊNCIA ARTIFICIAL NO DESENVOLVIMENTO

A. Assistência no Desenvolvimento de Código

As ferramentas de IA foram utilizadas principalmente como assistentes de programação durante o desenvolvimento dos benchmarks. A principal contribuição foi na geração de código boilerplate, sugestões de estruturação e identificação de padrões comuns entre os diferentes monitores. Esta assistência acelerou significativamente o desenvolvimento inicial, permitindo focar esforços em aspectos mais complexos como a lógica de negócio específica de cada monitor e a integração com o sistema operacional.

No entanto, todo o código desenvolvido foi revisado, testado e adaptado manualmente. As implementações em C, particularmente as interfaces userspace com ncurses, requereram ajustes significativos e depuração extensiva. As implementações em

Elixir, embora mais concisas, também necessitaram de refinamento para garantir compatibilidade com o compilador Honey Potion e correção de problemas de compilação.

B. Assistência na Análise e Documentação

A análise comparativa entre as implementações em C e Elixir foi conduzida manualmente, com ferramentas de IA auxiliando na organização e formatação dos resultados. A geração de tabelas, gráficos e visualizações foi facilitada por scripts automatizados, mas a interpretação dos dados, identificação de padrões e formulação de conclusões foram realizadas através da minha análise humana.

A documentação foi escrita com assistência LLMs, que auxiliaram na estruturação de parágrafos, verificação de consistência terminológica e sugestões de melhorias de clareza. Todas as decisões sobre conteúdo, argumentação e conclusões foram tomadas por mim, com a IA servindo como ferramenta de refinamento textual.

C. Impacto na Produtividade

O uso de ferramentas de IA resultou em ganhos significativos de produtividade em tarefas repetitivas e de baixo nível. Estima-se que o tempo total de desenvolvimento foi reduzido em aproximadamente 30–40% comparado a um processo totalmente manual, permitindo focar mais tempo em aspectos críticos.

Esta aceleração foi particularmente valiosa na fase de desenvolvimento dos benchmarks, onde padrões similares se repetiam entre os diferentes monitores. A capacidade de reutilizar e adaptar código sugerido acelerou a implementação inicial, embora a depuração e otimização subsequentes tenham consumido tempo significativo, similar ao que seria esperado em desenvolvimento tradicional.

D. Limitações e Necessidade de Supervisão

É importante reconhecer que o uso de IA não eliminou a necessidade de supervisão humana rigorosa. Código gerado frequentemente continha erros sutis, incompatibilidades com bibliotecas específicas ou problemas de lógica que requeriam correção manual. Análises sugeridas frequentemente precisavam de validação contra dados reais e ajustes baseados em conhecimento de domínio.

A escrita assistida por IA também apresentou limitações. Textos gerados tendiam a ser genéricos ou a repetir padrões comuns, requerendo revisão substancial para garantir originalidade, precisão técnica e adequação ao contexto específico do trabalho. Todas as seções foram reescritas e refinadas manualmente para garantir qualidade acadêmica adequada.

E. Conclusão

O uso de ferramentas de inteligência artificial neste trabalho demonstrou valor significativo como assistente de produtividade, acelerando tarefas repetitivas e facilitando a organização de informações. No entanto, os aspectos mais críticos do trabalho permaneceram dependentes de contribuição humana fundamentada em conhecimento técnico e julgamento acadêmico.

Esta experiência reforça que ferramentas de IA são mais efetivas quando utilizadas para amplificar capacidades humanas em tarefas de baixo nível, permitindo que pesquisadores concentrem esforços em aspectos que requerem criatividade, julgamento crítico e conhecimento de domínio.

REFERENCES

[1] C. Liu, B. Tak, and L. Wang, "Understanding Performance of eBPF Maps," *Proc. ACM Meas. Anal. Comput. Syst.*, 2024.

[2] G. Fournier, "Process Level Network Security Monitoring & Enforcement with eBPF," *IEEE Symposium on Security and Privacy*, 2023.

[3] K. Huang et al., "SoK: Challenges and Paths Toward Memory Safety for eBPF," *IEEE Security & Privacy*, 2023.

[4] A. Stefanov and B. Gradskov, "Analysis of CPU Usage Data Properties and Their Possible Impact on Performance Monitoring," *Journal of System Performance*, 2022.

[5] Elixir Team, *Elixir Getting Started Guide*, 2023. [Online]. Available: https://elixir-lang.org/getting-started/introduction.html

[6] Linux Kernel Documentation, *BPF (Berkeley Packet Filter) and eBPF*, 2024. [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/index.html

[7] Honey Potion Project, *Repository Documentation and README*, 2024. [Online]. Available: https://github.com/honeypotion-framework

[8] bpftool Manual, *Linux Kernel eBPF Toolchain*, 2024. [Online]. Available: https://www.kernel.org/doc/man-pages/

[9] M. McHugh, "eBPF: A New Approach to Linux Kernel Extensibility," *Queue*, vol. 18, no. 1, pp. 58–74, 2020.

[10] B. Gregg, *BPF Performance Tools: Linux System and Application Observability*. Addison-Wesley Professional, 2019.

[11] B. Gregg, *Systems Performance: Enterprise and the Cloud*, 2nd ed. Pearson Education, 2021.

[12] D. Thomas, *Programming Elixir 1.6: Functional Concurrent Pragmatic Fun*. Pragmatic Bookshelf, 2018.

[13] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of USENIX Winter*, 1993, pp. 259–270.

[14] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson Education, 2006.

[15] BCC Developers, "BPF Compiler Collection (BCC)," 2023. [Online]. Available: https://github.com/iovisor/bcc

[16] bpftrace Developers, "bpftrace: High-level tracing language for Linux eBPF," 2018. [Online]. Available: https://github.com/iovisor/bpftrace

[17] libbpf Developers, "libbpf: BPF CO-RE (Compile Once – Run Everywhere)," 2021. [Online]. Available: https://github.com/libbpf/libbpf

[18] A. Madhavapeddy et al., "Unikernels: Library Operating Systems for the Cloud," in *Proceedings of ASPLOS*, 2013, pp. 461–472.

[19] Rust for Linux, "Rust Support in the Linux Kernel," 2023. [Online]. Available: https://github.com/Rust-for-Linux