Calibração de Latência Microarquitetural: Uma Abordagem Baseada em Serialização de Instruções e Compensação de *Drift*[*]

Matheus B. Guerra[1], **José L. N. C. de Oliveira**[1], **Diego L. C. Dutra**[1]

[1]Universidade Federal do Rio de Janeiro (UFRJ)

matheus.barreira@poli.ufrj.br, {joliveira,ddutra}@cos.ufrj.br

Abstract. *This work presents a methodology for automatic cache latency calibration, eliminating manual and empirical adjustments in side-channel attacks. The approach uses serialization instructions and the Time Stamp Counter (TSC) to ensure accurate measurements in user space. The key difference lies in the systematic isolation of cache levels and the compensation of data drift against CPU frequency fluctuations. Results on modern hardware confirm the method's effectiveness by achieving conformity with values found in the literature.*

Resumo. *Este trabalho apresenta uma metodologia para calibração automática de latências de cache, eliminando ajustes manuais e empíricos em ataques de canal lateral. A abordagem utiliza instruções de serialização e o Time Stamp Counter (TSC) para garantir medições precisas no espaço de usuário. O diferencial reside no isolamento sistemático dos níveis de cache e na compensação de data drift frente a flutuações de frequência da CPU. Resultados em hardware moderno confirmam a eficácia do método ao atingir conformidade com os valores encontrados da literatura.*

1. Introdução

A constante evolução da microarquitetura de computadores responde à necessidade de mitigar o abismo de desempenho entre o processador e a memória principal. Para esse fim, projetistas incorporam camadas complexas de abstração e otimizações de hardware que tornam a interação entre o código e a microarquitetura cada vez menos transparente [Levinthal]. Embora tais avanços visem ao ganho de performance, eles introduzem sutilezas no funcionamento do hardware que podem comprometer a segurança do sistema, pois permitem que informações em manuais sofram alterações ou omitam comportamentos exploráveis [Levinthal , Kocher et al. 2020].

Nesse cenário, emergiu uma classe crítica de vulnerabilidades denominada ataques de canal lateral (*side-channel attacks*). A descoberta do *Spectre* [Kocher et al. 2020] revelou que escolhas de projeto, como a execução especulativa, permitem o vazamento de dados sensíveis, enquanto o *Meltdown* [Lipp et al. 2020] comprovou que a execução fora-de-ordem possibilita a leitura não autorizada de toda a memória do sistema por usuários sem privilégios. Diferente de *malwares* convencionais, tais ameaças exploram falhas intrínsecas ao *design* do processador, o que torna o desenvolvimento de mitigações um

[*]Esse projeto foi parcialmente financiado com recursos de bolsas de produtividade do CNPq (PQ-C: 304308/2025-0) e da FAPERJ (JCNE: E-26/204.481/2025)

desafio complexo que exige intervenções desde o microcódigo até o núcleo do sistema operacional.

Embora ameaças como *Spectre*, *Meltdown* e o posterior *RIDL* [Van Schaik et al. 2019] apresentem alta complexidade, sua base operacional reside em mecanismos fundamentais de transferência de dados secretos para fora de domínios protegidos. O método *Flush+Reload* destaca-se como a principal técnica nesse contexto, ao monitorar a interação da vítima com a cache por meio da medição precisa do tempo de acesso a endereços de memória compartilhados [Shen et al. 2021, Yarom and Falkner 2014, Kocher et al. 2020].

Entretanto, as abstrações dos sistemas modernos impõem obstáculos à observação precisa da cache. Mecanismos automatizados pelo hardware, como o prefetching, e a ausência de instruções nativas para forçar dados em níveis específicos de cache dificultam a medição acurada da latência [Patterson and Hennessy 2016]. Além disso, nota-se que trabalhos influentes na área frequentemente utilizam limiares (thresholds) de detecção de cache hit otimizados manualmente para as máquinas dos próprios pesquisadores. Tal prática reduz a reprodutibilidade dos experimentos e impõe uma carga de recalibração manual ao adaptar o ataque a novas plataformas [Yarom and Falkner 2014].

Diante da escassez de ferramentas na literatura que realizem o ajuste automático desses parâmetros, este trabalho propõe uma metodologia para o tratamento de ruído e o isolamento sistemático de componentes microarquiteturais. O objetivo consiste em desenvolver um método automatizado para obter valores médios e desvios-padrão de latência em todos os níveis de cache em computadores comerciais modernos. Ao automatizar a determinação do limiar de detecção, este estudo busca avançar na precisão de ataques de canal lateral e, consequentemente, contribuir para o desenvolvimento de contramedidas mais eficazes e universais.

O restante deste artigo segue esta organização: a Seção 2 apresenta os conceitos fundamentais, enquanto a Seção 3 discute os principais trabalhos relacionados. A Seção 4 detalha a metodologia de caracterização e calibração que este trabalho propõe. A Seção 5 traz a avaliação e a análise experimental da abordagem. Finalmente, a Seção 6 fornece as considerações finais e delineia direções para trabalhos futuros.

2. Fundamentação Teórica

Esta seção apresenta os conceitos fundamentais sobre a microarquitetura de processadores modernos e os mecanismos que viabilizam os canais laterais. Tais conceitos estabelecem a base para a metodologia de calibração automática que este trabalho propõe.

2.1. Fatias de Cache (Cache Slices)

Com o aumento do número de núcleos por chip, as arquiteturas evoluíram para o modelo NUCA (*Non-Uniform Cache Access*). Nesse paradigma, o projeto subdivide o último nível de cache (LLC/L3) em fatias (*slices*), distribuindo-as fisicamente próximas a cada núcleo para reduzir a latência média de acesso [Irazoqui et al. 2015]. Embora a L3 permaneça logicamente compartilhada, a comunicação entre núcleos e fa-

tias ocorre via um barramento em anel (*ring bus*). Essa estrutura introduz variações de latência que dependem da distância física (saltos no anel) entre o requisitante e o dado [Paccagnella et al. 2021, int 2023]. Além disso, funções de *hash* proprietárias gerem a distribuição dos endereços entre as fatias, o que cria um fator de imprevisibilidade e dificulta a medição precisa da latência sem uma caracterização sistemática [Irazoqui et al. 2015].

2.2. Mecanismos de Prefetching

O *prefetching* otimiza o desempenho ao antecipar as necessidades da CPU, carregando dados da memória principal para a cache antes de solicitações explícitas. Esse mecanismo identifica padrões de acesso, como sequências ou saltos fixos (*strided prefetch*) [Mittal 2016]. Em ataques de canal lateral, o *prefetcher* atua como uma fonte significativa de ruído: ele pode carregar linhas de cache de forma especulativa e gerar falsos positivos em medições de *cache hit*. Devido à sua natureza automática e à dificuldade de desativação em nível de usuário na arquitetura x86, o pesquisador deve empregar técnicas para mitigar essa influência durante a calibração de latência.

2.3. Migração de Core e Escalonamento

O agendador do sistema operacional (*scheduler*) pode interromper processos ou realocá-los entre diferentes núcleos para balancear a carga [Silberschatz et al. 2019]. Para a análise de canais laterais, a migração de core assume um papel crítico. Como o hardware fixa o mapeamento de endereços nas fatias de cache L3 com base no endereço físico, a migração de um processo para um núcleo mais distante da fatia original altera a latência percebida. Tal variação invalida limiares estabelecidos previamente.

2.4. Salvaguardas contra Execução Fora de Ordem

Processadores modernos utilizam execução fora de ordem para maximizar o uso das unidades funcionais. Para garantir a precisão de medições cronometradas, o desenvolvedor deve utilizar instruções de serialização que criam barreiras arquiteturais. A instrução `cpuid` realiza a serialização total, impedindo que instruções cruzem a barreira em ambas as direções [Paoloni 2010, Intel Corporation a]. No contexto de medição de latência, as instruções `mfence` e `lfence` desempenham papéis fundamentais: enquanto a primeira garante a finalização de operações de memória (leitura e escrita), a segunda assegura que leituras subsequentes não iniciem antes da conclusão das instruções anteriores. Juntas, elas mitigam riscos de execução transiente durante a medição [Yarom and Falkner 2014, Intel Corporation a].

2.5. Ataques de Canal Lateral e Execução Especulativa

Ataques de canal lateral exploram efeitos colaterais da implementação de hardware para vazar informações. A execução especulativa otimiza o desempenho ao prever caminhos de desvio (*branch prediction*), mas pode realizar operações de leitura baseadas em previsões errôneas — a chamada execução transiente [Levinthal]. Embora o processador não escreva os resultados dessas operações nos registradores arquiteturais, as alterações no estado da cache persistem, o que serve como um vetor de vazamento.

Estruturas como o *Branch Target Buffer* (BTB) e o *Pattern History Table* (PHT) preveem destinos de saltos e resultados de condicionais [Kocher et al. 2020,

Wikner and Razavi 2025]. O envenenamento dessas estruturas permite ataques como o **Spectre**, que induz a vítima a acessar dados restritos especulativamente. Esse acesso deixa rastros na cache que o atacante recupera via **Flush+Reload** [Kocher et al. 2020, Yarom and Falkner 2014].

2.6. Técnica Flush+Reload

A técnica *Flush+Reload* compreende três etapas:

1. **Flush:** O atacante remove uma linha de cache específica da hierarquia de memória por meio da instrução `clflush`.
2. **Wait:** O atacante aguarda a execução do processo vítima.
3. **Reload:** O atacante acessa novamente o endereço e mede o tempo de resposta.

Um tempo de acesso curto (abaixo de um limiar específico) confirma que a vítima acessou o dado e o trouxe de volta à cache. A eficácia desta técnica depende inteiramente da distinção precisa entre um *hit* e um *miss*, o que reforça a necessidade da metodologia de caracterização automática que este trabalho propõe.

3. Trabalhos Relacionados

Existem diversas ferramentas e *benchmarks* para análise de desempenho microarquitetural descritos na literatura, abrangendo tanto soluções industriais baseadas em contadores de desempenho quanto suítes experimentais voltadas à pesquisa acadêmica. Contudo, conforme discutido a seguir, tais soluções frequentemente não se adequam às necessidades específicas de ataques de canal lateral, seja por exigirem privilégios de acesso ou por não oferecerem a precisão necessária para o isolamento de latência.

Embora ferramentas como *Perf* [Linux Kernel Developers], *Intel VTune* [Intel Corporation d], *AMD uProf* [Advanced Micro Devices, Inc.] e *Intel PCM* [Intel Corporation c] permitam o monitoramento detalhado de eventos microarquiteturais através de *Hardware Performance Counters* (HPCs) integrados à PMU [Intel Corporation b], sua eficácia em cenários de segurança é limitada pela necessidade de privilégios elevados para acessar contadores críticos, como os de cache L3 [Abel and Reineke 2020]. Essa dependência do kernel restringe a aplicabilidade dessas soluções em ataques que operam em espaço de usuário, como o *Spectre*, impossibilitando a medição direta ou furtiva de latências sem as permissões adequadas [Abel and Reineke 2020, Intel Corporation d].

No âmbito acadêmico, suítes como *Memscope* [Ghaemi et al. 2025] e *NanoBench* [Abel and Reineke 2020] realizam a aferição de latências de cache, porém, assim como as ferramentas industriais, dependem de HPCs e sofrem com restrições de privilégio e medidas indiretas. Em contraste, o *IsolBench* [Valsan et al. 2016] utiliza o *Time Stamp Counter* (TSC) para obter ciclos de latência diretamente, mas introduz *overhead* de instruções de controle ao realizar múltiplos acessos dentro de um único intervalo de medição [Prathap, S. and others]. O presente trabalho diferencia-se ao priorizar a redução de ruídos e valores espúrios, realizando medições de acessos individuais e empregando funções de serialização específicas para minimizar interferências microarquiteturais.

Wang et al. [Wang et al. 2024] analisam protocolos de coerência em sistemas heterogêneos (CPU, GPU e FPGAs) utilizando as ferramentas *Perf* e *Intel PCM*. Para mitigar ruídos experimentais, os autores desabilitam mecanismos de *prefetching* e *Simultaneous Multi-Threading* (SMT), além de fixarem a frequência do processador por meio do governador de escala da CPU. Adicionalmente, empregam codificação em *Assembly* para evitar otimizações indesejadas do compilador e utilizam vetores com dimensões significativamente superiores à capacidade da cache L3, visando isolar a latência característica da memória principal.

4. Abordagem para Caracterização e Calibração Automática

Diferente das soluções citadas anteriormente, a proposta detalhada nesta seção foca na obtenção de métricas de cache por meio da aplicação de técnicas de ponta para redução de ruído, como a serialização de instruções, mantendo a furtividade operacional. Ao utilizar métodos baseados puramente no *Time Stamp Counter* (TSC), a metodologia dispensa o acesso privilegiado ao kernel, tornando-a viável para a caracterização de ambientes sob a perspectiva de um atacante de canal lateral. Essa abordagem baseia-se na manipulação indireta da hierarquia de memória por meio de padrões de acesso controlados, visando preparar o estado microarquitetural antes de cada medição para isolar sistematicamente cada nível de cache. A partir das distribuições de tempo obtidas, o sistema é capaz de derivar automaticamente os limiares (*thresholds*) que distinguem um *cache hit* em cada nível da hierarquia, garantindo latências precisas e isentas de ruído.

4.1. Implementação da Sonda de Medição e Barreiras de Serialização

Conforme discutido na Seção 2, a execução fora de ordem pode permitir que o processador realoque instruções para dentro do intervalo de medição, corrompendo a precisão dos ciclos contados. Para mitigar esse efeito, a sonda de medição utiliza instruções de serialização de memória e de instruções.

A listagem 1 ilustra a aplicação prática dessas barreiras. A introdução de instruções `lfence` obriga a conclusão da instrução anterior antes que a subsequente seja iniciada. No contexto da medição de latência, isso impede, por exemplo, que a instrução `mov` (leitura do endereço) seja executada antes da primeira leitura do contador `rdtsc`.

```
asm __volatile__(
   "add  $5, %%ebx      \n"
   "lfence              \n" // Garante a conclusao do ADD
   "mov  (%0), %%eax    \n" // Instrucao alvo da medicao
   "lfence              \n" // Garante a conclusao do MOV
   "imul $3, %%ecx, %%ecx \n"
   :: "d" (adrs)
   : "eax", "ebx", "ecx", "memory"
);
```

Código 1. Exemplo de serialização de trecho de código com `lfence` para garantir ordem de execução.

Além da serialização de instruções, é imperativo garantir a serialização de memória para que os *buffers* de sistema não estejam ocupados com operações de escrita/leitura

anteriores no momento da medição. Para isso, utiliza-se a instrução `mfence` em conjunto com `lfence`. Enquanto o `mfence` garante que todas as requisições de memória anteriores foram despachadas, o `lfence` atua como uma barreira de *software* que impede o despacho de novas instruções até que as anteriores tenham sido finalizadas arquiteturalmente.

A listagem 2 demonstra o uso conjunto dessas barreiras aplicado à leitura de múltiplos endereços, garantindo que a latência medida para `address2` não sofra interferência de um *hazard* de memória provocado por `address1`.

```
asm __volatile__(
   "mov (%0), %%eax   \n" // Acesso previo
   "mfence            \n" // Serializa escrita/leitura de memoria
   "lfence            \n" // Serializa o fluxo de instrucoes
   "mov (%1), %%ebx   \n" // Acesso alvo da medicao
   : "r" (address1), "r" (address2)
   : "%eax", "%ebx", "memory"
);
```

Código 2. Uso conjunto de `mfence` e `lfence` para serialização de memória e instrução.

Essas garantias de baixo nível são integradas às funções `probe()` e `probe_dummy()` descritas anteriormente, permitindo que o sistema de calibração automática proposto opere com uma margem de erro mínima, essencial para a distinção entre os níveis de cache L1, L2 e L3.

4.2. Isolamento de Níveis de Cache

Para garantir a medição independente entre as camadas, a estratégia adotada consiste em induzir a substituição de um endereço alvo nos níveis superiores da cache. Primeiro, realiza-se a leitura de um endereço conhecido para trazê-lo à cache. Em seguida, executa-se uma sequência de leituras em endereços distintos, de modo que o volume total de dados carregados seja igual ou superior à soma das capacidades dos níveis de cache superiores ao nível alvo.

Baseando-se em uma organização associativa em conjuntos (*set-associative*) e em uma política de substituição LRU (*Least Recently Used*), torna-se altamente provável que o endereço original seja expelido das caches menores (L1 ou L2) e permaneça apenas no nível desejado (L3 ou memória principal). A implementação deste mecanismo é detalhada no Código 3. O vetor `cache_filling_array` é percorrido em saltos de 64 bytes para garantir que cada leitura carregue uma linha de cache distinta, evitando redundâncias.

```
volatile void get_latency(int num_medicoes, uint8_t
   *cache_filling_array, int array_size, char *log_file_name){
   for(int i = 0; i < num_medicoes; i++){
      lixo = probe_address[0]; // Traz o dado para a cache
      for(int j = 0; j < (array_size)/64; j++){
         lixo = cache_filling_array[j*64]; // Preenche a cache
            para causar a expulsao
      }
      latency[i] = probe(&probe_address[0]); // Mede latencia
         real
      latency_dummy[i] = probe_dummy(); // Mede overhead
   }
}
```

Código 3. Código para substituição de bloco por preenchimento da cache.

4.3. Memória Compartilhada e Copy-on-Write

Identificou-se que vetores não inicializados são mapeados pelo kernel para uma *zeroed-page* global [LinuxVox 2023, Gorman 2004]. Em arquiteturas com múltiplos conjuntos (*sets*), isso faz com que múltiplos endereços virtuais apontem para o mesmo endereço físico, impedindo a substituição efetiva das linhas de cache. Para mitigar esse efeito, aplicou-se a técnica de *Copy-on-Write* (CoW), inicializando os vetores com valores distintos (Código 4), forçando o kernel a alocar páginas físicas únicas e permitindo a observação clara das latências diferenciadas entre os níveis.

```
void fill_array1(uint8_t *array){
   for(unsigned int i = 0; i < (L1_CACHE_SIZE)/(4*1024); i++){
      array[i*(4*1024)] = "m";
   }
}

void fill_array2(uint8_t *array){
   for(unsigned int i = 0; i < (2*L2_CACHE_SIZE)/(4*1024); i++){
      array[i*(4*1024)] = "b";
   }
}
```

Código 4. Código das funções de escrita em vetor para causar *copyonWrite* nos vetores de preenchimento de cache e garantir que cada página gere entradas únicas a cache e não incorra em cache *hit*.

4.4. Ferramentas de Medição e Sondas Assembly

Para evitar otimizações indesejadas do compilador e ruídos de execução fora de ordem, utilizou-se uma sonda de medição em Assembly, baseada na técnica de Yarom e Falkner [Yarom and Falkner 2014]. No entanto, adaptou-se a função original removendo a instrução `clflush`, permitindo a permanência do dado na cache após a medição para a distinção de níveis.

Como demonstrado no Código 5, a precisão da medição é assegurada pelo uso rigoroso de instruções de serialização. A instrução *mfence* garante que operações de memória anteriores sejam concluídas, enquanto o *lfence* previne a execução fora de ordem ao impedir que o *rdtsc* e a instrução de leitura alvo (*mov*) sejam reordenados para fora do intervalo demarcado. Sem esse controle, a reordenação microarquitetural poderia fazer com que a leitura do contador ocorresse após o início do acesso à memória, resultando em uma latência medida inferior à real, ou permitiria que instruções externas influenciassem o tempo transcorrido.

Entretanto, observou-se que os valores iniciais eram superiores aos previstos na documentação técnica [Levinthal], uma vez que o *overhead* das instruções de controle era somado indevidamente à latência de memória. Para mitigar esse erro sistemático, introduziu-se a "sonda maquete"(`probe_dummy`), detalhada no Código 6, que reproduz exatamente a estrutura da sonda original, exceto pela instrução de acesso ao endereço. Ao subtrair o tempo obtido pela sonda maquete do valor da sonda principal, é possível isolar a sobrecarga instrumental e obter o tempo líquido real de acesso à hierarquia de memória.

```
static inline uint64_t
   probe(volatile uint8_t
   *adrs){
   volatile unsigned long time;
   asm __volatile__(
      " mfence   \n"
      " lfence   \n"
      " rdtsc    \n"
      " lfence   \n"
      " mov %%eax, %%esi \n"
      " mov (%1), %%eax \n"
      " lfence   \n"
      " rdtscp   \n"
      " sub %%esi, %%eax \n"
      " mfence   \n"
      " lfence   \n"
      : "=a" (time)
      : "c" (adrs)
      : "%esi", "%edx"
   );
   return time;
}
```

Código 5. Sonda real para latência.

```
static inline uint64_t
   probe_dummy()
{
   volatile unsigned long time;
   asm __volatile__(
      " mfence   \n"
      " lfence   \n"
      " rdtsc    \n"
      " lfence   \n"
      " mov %%eax, %%esi \n"
      // vazio
      " lfence   \n"
      " rdtscp   \n"
      " sub %%esi, %%eax \n"
      " mfence   \n"
      " lfence   \n"
      : "=a" (time)

      :: "%esi", "%edx"
   );
   return time;
}
```

Código 6. Sonda maquete (*overhead*).

Os resultados obtidos nesta avaliação são apresentados na Tabela 1. É perceptível que a variação no número de cores não afetou significativamente a latência. A variação registrada oscilou em torno de 10 ciclos, enquanto a variação referente ao desvio de dados ocorre em torno de 60 ciclos. Além disso, a variação do desvio padrão que observamos para outros níves da memória cache mostra que o desvio não afeta apenas a cache L3 e que, portanto, a causa não pode ser as fatias de cache.

Tabela 1. Resultado do teste da influência das fatias de cache na latência da cache L3.

Core	Ciclos (CLK)
Core 0	31.4
Core 1	36.3
Core 2	26.6
Core 3	24.5

4.5. Compensação de Desvio de Dados (*Data Drift*)

Durante os experimentos, observou-se um desvio de dados (*data drift*) nas latências medidas, o que levou à investigação da hipótese de interferência das fatias de cache (*cache slices*) por meio da migração forçada de processos entre núcleos (Códigos 7 e 8). Contudo, os resultados indicaram que a variação era, na verdade, causada pela flutuação da frequência de operação do núcleo em relação à frequência constante do *Time Stamp Counter* (TSC).

```
printf("core 0 \n");
assign_to_this_core(0);
sleep(1);
get_latency(1000, array2, L1_CACHE_SIZE+L2_CACHE_SIZE,
   "log_auto_threshold_l3");

printf("core 1 \n");
assign_to_this_core(1);
sleep(1);
get_latency(1000, array2, L1_CACHE_SIZE+L2_CACHE_SIZE,
   "log_auto_threshold_l3");

printf("core 2 \n");
assign_to_this_core(2);
sleep(1);
get_latency(1000, array2, L1_CACHE_SIZE+L2_CACHE_SIZE,
   "log_auto_threshold_l3");

printf("core 3 \n");
assign_to_this_core(3);
sleep(1);
get_latency(1000, array2, L1_CACHE_SIZE+L2_CACHE_SIZE,
   "log_auto_threshold_l3");
```

Código 7. Teste da hipótese de que o ruído introduzido pelas fatias de cache estivesse causando desvio de dados verificado. O dado está fixo em uma fatia, mas o processo é migrado entre diferentes núcleos. A função *assign_to_this_core()* fixa a preferência do processo por um core específico e a função s*sleep* provoca o agendador a realizar a migração.

Como o TSC visa medir a passagem do tempo real de forma independente da

frequência da CPU, a latência em nanossegundos ($Latencia_{ns}$) de uma operação de leitura torna-se inversamente proporcional à frequência instantânea do processador, enquanto a contagem de ciclos registrada pelo TSC ($Latencia_{medida}$) permanece proporcional a essa latência temporal. Essa dinâmica física entre as grandezas pode ser expressa pelas seguintes equações:

$$Latencia_{ns} = \frac{Latencia_{CPU}}{Freq_{CPU}} \tag{1}$$

$$Latencia_{medida} = Freq_{TSC} \cdot Latencia_{ns} \tag{2}$$

```
void assign_to_this_core(int core_id){
   CPU_ZERO(&mask);
   CPU_SET(core_id, &mask);
   sched_setaffinity(0, sizeof(mask), &mask);
}
```

Código 8. Código da função usada para fixar a execução do processo atual a um core específico.

A relação matemática utilizada para compensar esse efeito e normalizar as medidas para a frequência nominal da CPU é apresentada a seguir:

$$Latencia_{compensada} = Latencia_{medida} \times \frac{Freq_{Atual}}{Freq_{Nominal}}$$

Esta compensação, implementada conforme o Código 9, permitiu que as medidas se tornassem invariantes e compatíveis com os manuais da Intel [Levinthal].

Assim, podemos observar algebricamente por que a latência medida está variando inversamente em relação à frequência do core. Para corrigir isso, é necessário obter a latência original compensando o efeito da conversão da frequência de CPU. O código abaixo mostra o código utilizado para realizar a compensação da frequência.

```
actual_frequency = get_sysfs_freq(sched_getcpu())/1000;
nominal_frequency = get_cpu_nominal_freq()*1000;


latency[i] = latency[i]*actual_frequency/nominal_frequency;
latency_dummy[i] =
   latency_dummy[i]*actual_frequency/nominal_frequency;
```

Código 9. Compensando o efeito da variação da frequência do core na medida. A multiplicação e divisão por 1000 se prestam a representar *actual_frequency* e *nominal_frequency* na mesma ordem de grandeza.

Dessa forma, foi possível solucionar o problema do desvio de dados, pois as medidas passaram a ser invariantes e compatíveis com as descritas nos manuais da Intel, mostrando assim a adequação do método proposto.

5. Avaliação Experimental

A avaliação experimental utiliza um processador Intel Core i5-1135G7 de 11ª geração (arquitetura x86 e microarquitetura Ice Lake), com frequência nominal de 2,4 GHz. A hierarquia de memória compreende os níveis L1d, L2 e L3 com capacidades de 48 KB, 1280 KB e 8192 KB, respectivamente. O chip dispõe da tecnologia de Simultaneous Multi-Threading (SMT), na qual cada núcleo físico possui duas unidades de processamento lógico que o sistema operacional identifica como núcleos independentes. Os testes ocorreram sob o sistema operacional Ubuntu 22.04. O código-fonte que implementa a metodologia proposta e gera os resultados a seguir encontra-se disponível no GitHub[1].

Conforme discutido na Seção 4, aprimoramos o código para contemplar as particularidades dos processadores modernos e garantir a obtenção de médias acuradas para a latência de cache. A Tabela 2 apresenta a média e o desvio padrão obtidos em conjunto com os valores teóricos de referência. Coletamos esses dados a partir de 10.000 medições em cada nível da hierarquia de memória e realizamos o tratamento estatístico por meio de scripts em Python. Nesse processo, removemos os *outliers* utilizando a segmentação dos dados em quartis, definindo o intervalo central e o critério de exclusão de acordo com as seguintes fórmulas:

$$IQR = Q3 - Q1$$
$$LimiteInferior = Q1 - 1,5 * IQR$$
$$LimiteSuperior = Q3 + 1,5 * IQR$$

Atribuímos a ocorrência de *outliers* predominantemente à variabilidade no tempo de execução das instruções `rdtsc` e `lfence`, as quais dependem do estado transiente do motor de execução fora de ordem [Yarom and Falkner 2014]. Como o estado microarquitetural pode divergir entre o momento da execução da sonda principal e o da sonda maquete, a parcela variável da sobrecarga instrumental não é integralmente compensada. Observamos, contudo, que os valores teóricos de latência permanecem dentro do intervalo de um desvio padrão (1σ) em relação à média obtida, o que confirma a compatibilidade das medições. Conforme ilustra o histograma da Figura 1, o ruído arquitetural não compensado manifesta um comportamento probabilístico gaussiano em torno do valor médio, resultado do acúmulo de interferências aleatórias provenientes dos diversos componentes da arquitetura.

A análise visual da Figura 1 revela ainda que o deslocamento da latência da cache L3 em relação à L2 é superior ao observado entre L2 e L1. Esse comportamento era esperado, visto que a camada L3 é externa ao núcleo e demanda um tempo de acesso fisicamente maior para o tráfego de dados. Por fim, identificamos um pico de frequência em 0 ciclos para a cache L1, fenômeno que decorre da natureza parcialmente serializante da instrução `lfence`. Essa característica permite que a segunda leitura do contador ocorra antes da primeira em cenários de alta velocidade, gerando latências negativas. Para garantir a coerência física dos dados apresentados, truncamos todos os valores negativos para zero durante a fase de pós-processamento.

[1] `https://github.com/Lab-COMPASSO/cache_benchmarking_tcc_matheus.git`

Tabela 2. Resultado do microbenchmark de latência da cache incluindo média, desvio padrão e valores de referência segundo o Dr. David Levinthal [Levinthal].

Nível	Latência Média (CLK)	Desvio Padrão (CLK)	Latência esperada (CLK)
L1	3.0	2,3	4
L2	11.1	2.6	10
L3	44.7	6.3	40

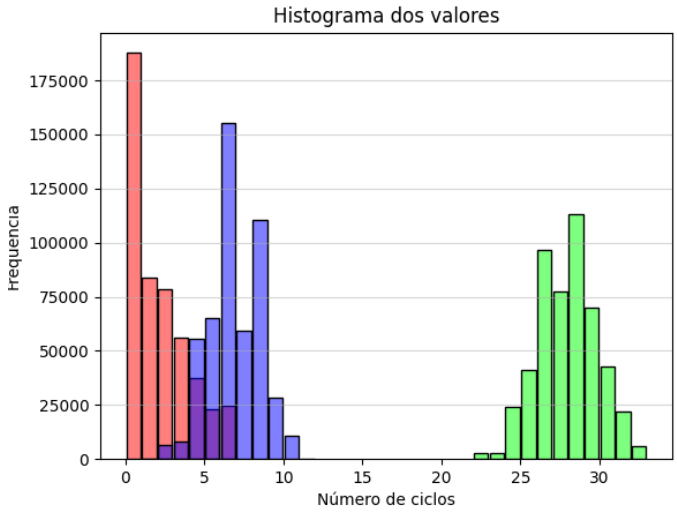

Figura 1. Histograma da estatística de medição dos três níveis de cache. Em vermelho está representado o cache L1; em azul, o cache L2; e em verde, o cache L3. A frequência indica quantas vezes um dado número de ciclos de latência foi medido durante o experimento.

Analisamos o efeito da variação de frequência sem compensação por meio da Figura 2, onde se observa que a latência média da cache L3 atinge quase o triplo do valor esperado. Os dados da Tabela 3 confirmam que a ausência de tratamento para o *frequency drift* desloca o valor médio da L3 para além de 3σ de incerteza. Esse cenário comprova a necessidade de ajustar as métricas conforme a frequência instantânea do processador para garantir a precisão da caracterização.

6. Conclusão

A eficácia dos ataques de canal lateral à cache reside fundamentalmente na precisão com que o atacante amostra o sinal vazado, o qual ele recupera mediante a medição das latências de acesso à memória. Historicamente, a determinação do limiar (*threshold*) para distinguir um *cache hit* de um *miss* constituía um processo empírico, dependente das especificações particulares da microarquitetura de cada plataforma experimental. Este trabalho demonstra que é possível automatizar essa caracterização com alta precisão em arquiteturas modernas *multi-core* e *multi-thread*. Ao empregar temporizadores baseados

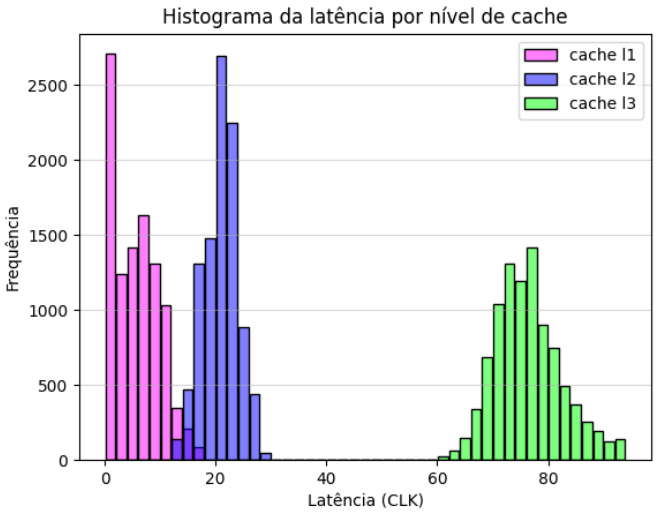

Figura 2. Histograma da estatística de medição dos três níveis de cache quando a frequência do *core* não é compensada. Em vermelho está representado o cache L1; em azul, o cache L2; e em verde, o cache L3.

Tabela 3. Resultado do teste de latência usando da cache quando o código desenvolvido, porém sem a compensação de frequência de core utilizada. O gráfico inclui média, desvio padrão e valores de referência segundo o Dr. David Levinthal [Levinthal].

Nível	Latência Média (CLK)	Desvio Padrão (CLK)	Latência esperada (CLK)
L1	5,0	4,1	4
L2	20,3	3,1	10
L3	76,0	6,1	40

em TSC que prescindem de privilégios de kernel, a metodologia cumpre o requisito de furtividade intrínseco a ataques como o *Spectre*, enquanto o uso de técnicas de serialização do estado da arte mitigou ruídos e produziu resultados em estrita conformidade com os valores teóricos esperados na literatura.

A ferramenta desenvolvida oferece as métricas necessárias para uma implementação de ataques microarquiteturais que seja, simultaneamente, realista, repetível e acurada. Como perspectivas futuras, este estudo propõe a integração deste mecanismo de cálculo automático em ataques que originalmente exigem calibração manual, permitindo maior portabilidade e sucesso na exploração. Adicionalmente, planeja-se validar a metodologia em uma gama mais ampla de implementações comerciais da Intel e AMD, bem como adaptar as sondas em *Assembly* para o conjunto de instruções ARM. Tais passos revelam-se essenciais para avaliar o impacto de variações específicas de hardware e estender a aplicabilidade desta técnica de calibração automática para o ecossistema de dispositivos móveis e sistemas embarcados.

Referências

(2023). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, Santa Clara, CA. Volume 1: Basic Architecture. Available at: https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html.

Abel, A. and Reineke, J. (2020). nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 34–46. IEEE.

Advanced Micro Devices, Inc. Amd uprof performance analysis tool. https://www.amd.com/pt/developer/uprof.html. Accessed: 2026-05-07.

Ghaemi, G., Franco, G., Taram, K., and Mancuso, R. (2025). Memscope: Open-source kernel-level framework for heterogeneous memory characterization. In *2025 IEEE Real-Time Systems Symposium (RTSS)*, pages 500–513. IEEE.

Gorman, M. (2004). *Understanding the Linux virtual memory manager*, volume 352. Prentice Hall Upper Saddle River.

Intel Corporation. Intel 64 and ia-32 architectures software developer's manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Combined volumes. Accessed: 2026-05-07.

Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, volume 3: Performance monitoring. https://www.intel.com/content/www/us/en/content-details/874248/intel-64-and-ia-32-architectures-software-developer-s-manual-combined-volumes-3a-3b-3c-and-3d-system-programming-guide.html. Accessed: 2026-05-07.

Intel Corporation. Intel performance counter monitor (pcm). https://github.com/intel/pcm. Accessed: 2026-05-07.

Intel Corporation. Intel vtune profiler. https://www.intel.com/vtune. Accessed: 2026-05-07.

Irazoqui, G., Eisenbarth, T., and Sunar, B. (2015). Systematic reverse engineering of cache slice selection in intel processors. In *2015 Euromicro Conference on Digital System Design*, pages 629–636. IEEE.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al. (2020). Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101.

Levinthal, D. Performance analysis guide for intel® core™ i7 processor and intel® xeon™ 5500 processors. . Acessed: 2026-05-07.

Linux Kernel Developers. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org. Accessed: 2026-05-07.

LinuxVox (2023). Linux zero page: Understanding the kernel's memory optimization. Accessed: 2026-05-07.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., et al. (2020). Meltdown: Reading kernel memory from user space. *Communications of the ACM*, 63(6):46–56.

Mittal, S. (2016). A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2):1–35.

Paccagnella, R., Luo, L., and Fletcher, C. W. (2021). Lord of the ring(s): Side channel attacks on the CPU On-Chip ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 645–662. USENIX Association.

Paoloni, G. (2010). How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel Corporation*, 123(170).

Patterson, D. A. and Hennessy, J. L. (2016). *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann.

Prathap, S. and others. Isolbench: A set of micro-benchmarks for evaluating multicore isolation. `https://github.com/CSL-KU/IsolBench`. GitHub repository, Accessed: 2026-05-07.

Shen, C., Chen, C., and Zhang, J. (2021). Micro-architectural cache side-channel attacks and countermeasures. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 441–448.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2019). *Operating system concepts*. John Wiley & Sons.

Valsan, P. K., Yun, H., and Farshchi, F. (2016). Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE.

Van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. (2019). Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE.

Wang, Z., Mahar, S., Li, L., Park, J., Kim, J., Michailidis, T., Pan, Y., Rosing, T., Tullsen, D., Swanson, S., et al. (2024). The hitchhiker's guide to programming and optimizing cxl-based heterogeneous systems. *arXiv preprint arXiv:2411.02814*.

Wikner, J. and Razavi, K. (2025). Breaking the barrier: Post-barrier spectre attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3516–3533. IEEE.

Yarom, Y. and Falkner, K. (2014). Flush + reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732.