

Realocação Física de Objetos em um *Garbage Collector* Tricolor sobre um Acelerador PIM: Avaliação em Ciclos Reais com o Sim2PIM

Christopher Eduardo Zai¹, João Fabrício Filho¹

¹Bacharelado em Ciência da Computação
Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brasil

christopher.310704@alunos.utfpr.edu.br, joaof@utfpr.edu.br

Abstract. *We extend a concurrent tri-color Garbage Collector (GC) integrated with Sim2PIM so that compaction actually relocates objects, and measure in real cycles whether offloading relocation to a Reconfigurable Vector Unit (RVU) pays off. Over 300 runs on the full simulation pipeline show that dispatching compaction to the RVU costs $2.60\text{--}3.05\times$ more cycles than the CPU backend, with no crossover across a $4096\times$ sweep of batch size, a cost we trace to software simulation of PIM instructions. Two results favour the approach: relocating objects physically costs no more than bookkeeping and needs $35\times$ fewer recompactions, and asynchronous dispatch does hide latency.*

Resumo. *Estendemos um Garbage Collector (GC) concorrente tricolor integrado ao Sim2PIM para que a compactação realocze objetos de fato e medimos, em ciclos reais, se descarregar a realocação para uma unidade vetorial reconfigurável (RVU) compensa. Em mais de 300 execuções sobre o pipeline completo, despachar a compactação para a RVU custou de 2,60 a 3,05 vezes mais ciclos que o backend de CPU, sem ponto de cruzamento em uma varredura de $4096\times$ no tamanho do lote, custo que rastreamos até a simulação em software das instruções PIM. Dois resultados favorecem a abordagem: realocar objetos fisicamente não custa mais que a contabilidade pura e exige 35 vezes menos recompactações, e o despacho assíncrono esconde latência.*

1. Introdução

Um *Garbage Collector* (GC) concorrente marca objetos vivos com *threads* entrelaçadas à aplicação, e a abstração tricolor [Dijkstra et al. 1978] torna essa marcação segura; o intervalo em que a aplicação é suspensa é a pausa *Stop-The-World* (STW). Coletores de baixa latência, como o do *runtime* Go [Hudson 2018], minimizam essas pausas, mas não compactam o *heap*, tolerando fragmentação que degrada a localidade das marcações seguintes [Blackburn and McKinley 2008], custo amplificado pelo *Memory Wall* [Wulf and McKee 1995]. *Processing-in-Memory* (PIM) aproxima a computação da DRAM [Mutlu et al. 2019], tornando a realocação de objetos candidata natural ao descarregamento.

Um GC geracional embarcado em DPUs UPMEM reduz acessos à DRAM em até 85,9% na coleta menor, mas deixa a coleta maior, com compactação, como trabalho futuro [Morimoto et al. 2025]; descarregar só a marcação de um coletor Java para núcleos

near-memory rende até $2\times$ de *speedup* [Thomas et al. 2022]. Falta evidência quantitativa sobre a fase que realoca objetos.

Nossa etapa anterior integrou um GC tricolor ao Sim2PIM [Forlin et al. 2022] e manteve a STW abaixo de 0,061 ms ao compactar fora do caminho da pausa, mas nenhum objeto mudava de endereço e o binário não emitia instruções PIM. Agora o GC realoca objetos fisicamente, o binário atravessa o *pipeline* real do simulador e comparamos ciclos de *hardware*, respondendo a três questões: **QP1**, o custo de despachar a compactação para a RVU frente à CPU e se alguma escala de lote o amortiza; **QP2**, se mover objetos custa mais que a compactação contábil e se o adensamento dura; **QP3**, se o despacho assíncrono reduz o bloqueio do *mutator*.

2. Implementação e Metodologia

O GC em C++ usa *threads* concorrentes drenando uma *worklist*. Implementamos três *backends* com a mesma política de seleção de *spans*, o que dá a linha de base justa cobrada nas revisões. **RVU_PIM** despacha o lote para a unidade vetorial [Santos et al. 2017], que emite pares `PIM_LOAD/PIM_STORE` de 1024 bits sobre a memória simulada, e devolve o controle ao *mutator* imediatamente. **CPU_ONLY** faz a mesma contabilidade em C++ puro, bloqueante. **CPU_ONLY_MOVING** é o único que realoca de fato: `gc_clone()` copia o objeto para um endereço novo e `update_all_pointers()` corrige raízes e campos internos dos objetos vivos.

O binário é o real: passa por LLVM 4.0 com `-mattr=+pim`, é ligado às unidades de execução da RVU e despacha 1248 instruções PIM ao simulador. Os ciclos vêm do contador *Actual Cycles* via RDPMC, e cada execução isola uma combinação de *backend* e cenário; temporizadores de *software* cercam a fase relevante e faltas de cache são amostradas via `perf_event_open`. A carga sintética constrói correntes encadeadas e árvores binárias de 64 nós com 70% de mortalidade por ciclo; o experimento de escala varia a capacidade do *span* de 64 a 262 144 objetos e o de latência usa 20 rodadas sob três volumes de trabalho do *mutator*. Cada combinação é repetida de 10 a 15 vezes em dois hospedeiros, um virtualizado de 2 núcleos e um AMD FX-4300 de 4, descartando *outliers* acima de $2\times$ a mediana; as contagens têm desvio zero e a suíte roda sob ASan/UBSan.

3. Resultados

3.1. QP1: custo em ciclos e ponto de cruzamento

Em 90 execuções isoladas com 384 objetos, RVU_PIM gastou 1,49 milhão de ciclos contra 476 e 468 mil dos *backends* de CPU, em faixas disjuntas. A hipótese natural era o custo fixo de despacho, diluído em lotes maiores, e a Tabela 1 a refuta: em 210 execuções sobre sete escalas, a razão RVU_PIM/CPU_ONLY fica entre 2,60 e 3,05, sem tendência de queda.

A causa é mecânica: a cada 16 objetos o trabalhador da RVU emite um par `PIM_LOAD/PIM_STORE` interpretado instrução a instrução pelas unidades do simulador, na mesma *thread* e sobre a mesma CPU do hospedeiro. É trabalho proporcional aos dados que CPU_ONLY nunca paga, e simular *hardware* em *software* não sai mais barato que executar o C nativamente. O custo por objeto converge em ambos, para 298 ciclos na CPU e 910 na RVU, mas nessa mesma razão.

Tabela 1. Ciclos reais por escala e *backend*, sem *outliers* (210 execuções)

Objetos/lote	RVU_PIM	CPU_ONLY	MOVING	RVU/CPU
64	354 803	116 642	120 235	3,04
256	519 409	195 564	165 302	2,66
1 024	1 130 400	391 912	369 370	2,88
4 096	3 867 751	1 332 255	1 066 893	2,90
16 384	12 708 398	4 893 429	4 921 697	2,60
65 536	55 036 372	19 365 215	19 055 397	2,84
262 144	238 621 369	78 158 200	80 387 737	3,05

3.2. QP2: custo e durabilidade da realocação física

CPU_ONLY e CPU_ONLY_MOVING são indistinguíveis, com razão entre 0,80 e 1,03 nas sete escalas: clonar objetos e corrigir ponteiros não aparece no custo total porque o gargalo, em ambos, é a seleção de *spans*. O adensamento também dura mais: em 200 ciclos sustentados, mover objetos exigiu 125 compactações de *span* contra 4380 e 4360 dos demais, cerca de 35 vezes menos.

Instrumentamos faltas de cache por fase, como pedido nas revisões, mas o sinal não é resolvível sem fixação de núcleos e isolamento de NUMA: entre réplicas do mesmo *heap*, com e sem realocação, a variação média na marcação foi de +35% com desvio de 73 p.p., ou seja, limitação de medição e não evidência contra a técnica.

3.3. QP3: latência do *mutator* e sobreposição

RVU_PIM é o único *backend* assíncrono, e ciclos totais não são a métrica que um coletor de baixa latência otimiza. Na Tabela 2 o padrão previsto por QP3 aparece limpo: o bloqueio de RVU_PIM cai de forma monotônica conforme o *mutator* recebe mais trabalho paralelo, e a espera medida na fase cai de 7,15 ms para 1,51 ms enquanto a sobreposição sobe de 8,20 ms para 76,18 ms. O mecanismo funciona, mas não compensa: mesmo no melhor cenário RVU_PIM bloqueia por 11,78 ms contra 0,144 ms de CPU_ONLY, coerente com o custo por instrução simulada de QP1.

Tabela 2. Bloqueio total do *mutator* em ms (desvio entre parênteses), 20 rodadas, AMD FX-4300, $N = 15$

<i>Backend</i>	Sem trabalho	Pouco trabalho	Muito trabalho
CPU_ONLY	0,144 (0,006)	0,144 (0,009)	0,143 (0,004)
CPU_ONLY_MOVING	0,499 (0,025)	0,508 (0,038)	0,495 (0,014)
RVU_PIM	18,77 (4,61)	16,83 (3,78)	11,78 (0,70)

4. Conclusão

Dentro do que o Sim2PIM simula hoje, não há evidência de que descarregar a compactação para a RVU acelere o GC tricolor: o custo é cerca de três vezes o da CPU e não se amortiza em 4096 vezes de escala (QP1). Em contrapartida, realocar objetos fisicamente não custa mais que a contabilidade pura e reduz em 35 vezes as recompactações (QP2), e o despacho assíncrono esconde latência atrás do *mutator* (QP3). As ressalvas são que RVU_PIM ainda não move bytes, que cada lote é fixado a um núcleo físico, o que

exercita só em parte o paralelismo pretendido, e que a carga permanece sintética, o que motiva repetir a avaliação com *benchmarks* tradicionais de coleta; o passo decisivo é o trabalhador copiar os bytes por pares PIM_LOAD/PIM_STORE de 128 bytes em hospedeiro com núcleos livres.

Uso de IA generativa: empregada na revisão redacional do texto; experimentos e análises são de responsabilidade dos autores.

Referências

- Blackburn, S. M. and McKinley, K. S. (2008). Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32.
- Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. (1978). On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975.
- Forlin, B. E., Santos, P. C., Becker, A. E., Alves, M. A. Z., and Carro, L. (2022). Sim2pim: a complete simulation framework for processing-in-memory. *Journal of Systems Architecture*, 128:102528.
- Hudson, R. L. (2018). Getting to go: the journey of go’s garbage collector. Keynote at the ACM SIGPLAN International Symposium on Memory Management (ISMM). Disponível em: <https://go.dev/blog/ismmkeynote>.
- Morimoto, R., Ichinose, K., and Ugawa, T. (2025). Gray-in-young: a generational garbage collection for processing-in-memory. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 122–133.
- Mutlu, O., Ghose, S., Gómez-Luna, J., and Ausavarungnirun, R. (2019). Enabling practical processing in and near memory for data-intensive computing. In *Proceedings of the 56th Annual Design Automation Conference (DAC)*.
- Santos, P. C., Oliveira, G. F., Tomé, D. G., Alves, M. A. Z., Almeida, E. C., and Carro, L. (2017). Operand size reconfiguration for big data processing in memory. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 710–715.
- Thomas, S., Choe, J., Gordon, O., Petrank, E., Moreshet, T., Herlihy, M., and Bahar, R. I. (2022). Towards hardware accelerated garbage collection with near-memory processing. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6.
- Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24.