

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



**ESTUDO EXPLORATÓRIO DO DESEMPENHO DE
ALOCADORES DE MEMÓRIA NO ESPAÇO DO USUÁRIO**

DIEGO ELIAS DAMASCENO COSTA

Uberlândia, Minas Gerais

2014

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



**ESTUDO EXPLORATÓRIO DO DESEMPENHO DE ALOCADORES DE MEMÓRIA
NO ESPAÇO DO USUÁRIO**

DIEGO ELIAS DAMASCENO COSTA

Dissertação de Mestrado apresentada à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software

Orientador: Prof. Dr. Rivalino Matias Júnior

Uberlândia, Minas Gerais

2014

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada “Estudo Exploratório do Desempenho de Alocadores de Memória no Espaço do Usuário” por Diego Elias Damasceno Costa como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação.

Uberlândia, Novembro de 2014.

Orientador:

Prof. Dr. Rivalino Matias Júnior
Universidade Federal de Uberlândia

Banca Examinadora:

Prof. Dr. Autran Macêdo
Universidade Federal de Uberlândia

Prof. Dr. Lúcio Borges de Araújo
Universidade Federal de Uberlândia

Prof. Dr. Rômulo Silva de Oliveira
Universidade Federal de Santa Catarina

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Novembro de 2014.

Autor: **Diego Elias Damasceno Costa**
Título: **Estudo Exploratório do Desempenho de Alocadores de Memória no Espaço do Usuário**
Faculdade: **Faculdade de Computação**
Grau: **Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

Autor

O AUTOR RESERVA PARA SI QUALQUER OUTRO DIREITO DE PUBLICAÇÃO DESTE DOCUMENTO, NÃO PODENDO O MESMO SER IMPRESSO E REPRODUZIDO, SEJA NA TOTALIDADE OU EM PARTES, SEM A PERMISSÃO ESCRITA DO AUTOR.

© Todos os direitos reservados a Diego Elias Damasceno Costa

DEDICATÓRIA

*Aos meus pais Elias e Silvana e aos meus irmãos Luan, Ane e Bruno.
A minha amada e companheira Priscila.*

AGRADECIMENTOS

A Deus,
que está sempre a iluminar os meus passos.

Ao meu orientador, Prof. Rivalino Matias Júnior,
pelas oportunidades e desafios que me foram proporcionados. A sua excelente orientação e firmeza foram fundamentais para o meu crescimento e conclusão deste trabalho. A você, Rivalino, minha eterna gratidão.

Aos meus pais Elias e Silvana,
pela educação, amor e apoio incondicional dado durante toda a minha vida. Todas as minhas conquistas são para vocês.

Aos meus irmãos Luan, Ane e Bruno,
pelos momentos indescritíveis de alegria, pelo apoio e compreensão durante essa desafiadora etapa. Vocês são a minha inspiração.

A minha amada Priscila,
pelo companheirismo, amor e apoio dado durante todo este nosso projeto juntos. Você é a minha fonte diária de amor e alegria.

Aos amigos de laboratório,
pela convivência e risadas, pela incessante ajuda e por todos os momentos de cooperação que me fizeram superar cada etapa deste trabalho.

RESUMO

O desempenho de operações de alocação de memória tem significativa influência no desempenho global da maioria das aplicações computacionais. Nesse sentido, a seleção de um alocador de memória é um importante requisito no projeto de sistemas computacionais mais sofisticados. A forte correlação entre o perfil de uso dinâmico da memória com o desempenho do alocador exige que a seleção do alocador ocorra por meio de um estudo experimental. Neste trabalho, foi realizado um estudo exploratório de um conjunto de seis alocadores de memória amplamente utilizados atualmente: Ptmalloc2 (alocador padrão da *glibc*), Ptmalloc3, Hoard, Miser, TCMalloc e Jemalloc. As cargas de trabalho usadas na avaliação dos alocadores foram planejadas com base em um estudo de caracterização de uso de memória de sete aplicações, duas aplicações do tipo Servidor e cinco aplicações *Desktop*. Cada alocador foi avaliado com relação ao seu tempo de resposta e consumo de memória, em um conjunto de 648 cenários diferentes de execução. Os resultados mostraram que, alocadores que obtiveram a menor média no tempo de execução foram também os melhores em aproveitar o paralelismo dos cenários com mais de uma *thread* e um processador. Os alocadores Jemalloc e TCMalloc foram em média cinco vezes mais rápidos do que o alocador padrão da *glibc*. Em certas condições o alocador Hoard apresentou uma economia de memória substancial se comparado aos demais alocadores avaliados, chegando a economizar cerca de 75% de memória. Tais resultados enfatizam a importância da escolha do alocador no projeto das aplicações, face às diferenças significativas observadas experimentalmente neste trabalho.

Palavras-chave: Alocadores de Memória, Caracterização de Alocações Dinâmicas de Memória, Avaliação de Desempenho

ABSTRACT

The performance of memory allocation operations significantly impacts the global performance in most computing applications. Thus, the choice of a memory allocator is an important aspect when projecting more sophisticated computing systems. The strong correlation between the dynamic memory use profile and the allocator performance requires the allocator to be chosen through an experimental study. This research carried out an exploratory study on a set of six widely used memory allocators: Ptmalloc2 (standard *glibc* allocator), Ptmalloc3, Hoard, Miser, TCMalloc, and Jemalloc. The workloads used to evaluate the allocators were based on a characterization study on memory use of seven application, two of which for servers and five for desktops. Each allocator was evaluated regarding its response time and memory usage in a set with 648 different execution scenarios. The results showed that the allocators with the shortest mean execution time also made the best use of parallelism in scenarios with more than one thread and one processor. The allocators Jemalloc and TCMalloc were, on average, five times faster than the standard *glibc* allocator. Under certain conditions, the allocator Hoard provided substantial memory savings of up to 75% when compared with the other allocators assessed. These results emphasize the importance of allocator choice when projecting applications in face of the significant differences observed experimentally in this study.

Keywords: Memory Allocators, Dynamic Memory Allocation Characterization, Performance Evaluation

SUMÁRIO

1	INTRODUÇÃO	1
1.1	CONTEXTUALIZAÇÃO	1
1.2	RELEVÂNCIA DO TRABALHO	2
1.3	OBJETIVOS	3
	<i>Geral:.....</i>	<i>3</i>
	<i>Específicos:</i>	<i>3</i>
1.4	MÉTODOS ADOTADOS	4
1.5	CONTRIBUIÇÕES DA PESQUISA.....	5
1.6	ESTRUTURA DO DOCUMENTO	5
2	FUNDAMENTAÇÃO TEÓRICA	7
2.1	INTRODUÇÃO.....	7
2.2	CARACTERÍSTICAS DE UM ALOCADOR DE MEMÓRIA	8
2.3	DESAFIOS DO GERENCIAMENTO DE ALOCAÇÕES DINÂMICAS DE MEMÓRIA	10
2.4	TRABALHOS RELACIONADOS	12
2.5	CONSIDERAÇÕES FINAIS	15
3	ALOCADORES INVESTIGADOS	17
3.1	INTRODUÇÃO.....	17
3.2	PTMALLOC2	17
	<i>3.2.1 Arena.....</i>	<i>17</i>
	<i>3.2.2 Chunk.....</i>	<i>18</i>
	<i>3.2.3 Bin.....</i>	<i>19</i>
3.3	PTMALLOC3	20
	<i>3.3.1 Chunk.....</i>	<i>20</i>
	<i>3.3.2 Bins.....</i>	<i>21</i>
3.4	HOARD	21
	<i>3.4.1 Superbloco.....</i>	<i>22</i>
	<i>3.4.2 Bins.....</i>	<i>23</i>
	<i>3.4.3 Implementação da heap.....</i>	<i>23</i>
3.5	MISER.....	24
	<i>3.5.1 Estruturas de Dados.....</i>	<i>24</i>
3.6	JEMALLOC	25
	<i>3.6.1 Estruturas de Dados.....</i>	<i>26</i>

3.6.2 Implementação da heap.....	27
3.7 THREAD-CACHE MALLOC (TCMalloc)	27
3.7.1 Estruturas de Dados.....	28
3.7.2 Implementação da heap.....	28
4 CARACTERIZAÇÃO DA ALOCAÇÃO DINÂMICA DE MEMÓRIA EM APLICAÇÕES	30
4.1 INTRODUÇÃO	30
4.2 INSTRUMENTAÇÃO.....	30
4.3 APLICAÇÕES AVALIADAS	32
4.4 PLANEJAMENTO EXPERIMENTAL.....	33
4.4.1 MySQL	33
4.4.2 Cherokee.....	34
4.4.3 CodeBlocks	34
4.4.4 VLCPlayer	34
4.4.5 Octave.....	34
4.4.6 Inkscape	35
4.4.7 Lynx.....	35
4.4.8 Bancada de Testes	35
4.5 ASPECTOS AVALIADOS	35
4.5.1 Total de Memória Alocada.....	36
4.5.2 Tamanhos Alocados	36
4.5.3 Rotinas de Alocação	36
4.5.4 Memória Liberada	36
4.5.5 Tempo de Retenção	37
4.5.6 Padrões de Uso das Alocações de Longa Duração	37
4.6 ANÁLISE DOS RESULTADOS POR APLICAÇÕES	38
4.6.1 MySQL	38
4.6.2 Cherokee.....	40
4.6.3 CodeBlocks	41
4.6.4 VLCPlayer (áudio).....	43
4.6.5 VLCPlayer (vídeo)	44
4.6.6 Octave.....	46
4.6.7 Inkscape	47
4.6.8 Lynx.....	49
4.7 ANÁLISE COMPARATIVA DOS RESULTADOS	50
4.7.1 Total de Memória Alocada.....	50

4.7.2 Tamanhos Alocados.....	51
4.7.3 Rotinas de Alocação	52
4.7.4 Memória Liberada.....	52
4.7.5 Tempo de Retenção.....	53
4.8 CONSIDERAÇÕES FINAIS	54
5 ANÁLISE EXPERIMENTAL COMPARATIVA DE ALOCADORES DE MEMÓRIA EM NÍVEL DO USUÁRIO.....	55
5.1 INTRODUÇÃO.....	55
5.2 INSTRUMENTAÇÃO USADA.....	55
5.3 PLANEJAMENTO EXPERIMENTAL	58
5.4 ANÁLISE DOS RESULTADOS POR ALOCADOR.....	61
5.4.1 Tempo de Execução	61
5.4.2 Consumo de Memória	74
5.5 ANÁLISE COMPARATIVA DOS ALOCADORES	81
5.5.1 Tempo de Execução	82
5.5.2 Consumo de Memória	85
5.6 CONSIDERAÇÕES FINAIS	88
6 CONCLUSÕES	91
6.1 PRINCIPAIS RESULTADOS	91
6.2 LIMITAÇÕES DA PESQUISA.....	91
6.3 CONTRIBUIÇÕES PARA A LITERATURA	92
6.4 DIFICULDADES ENCONTRADAS.....	92
6.5 TRABALHOS FUTUROS.....	93
REFERÊNCIAS BIBLIOGRÁFICAS.....	94
APÊNDICE A: TOPOLOGIA DO PROCESSADOR UTILIZADO NO EXPERIMENTO DE COMPARAÇÃO DOS ALOCADORES.....	99
APÊNDICE B: TABELAS DO TESTE DE TUKEY	101
APÊNDICE C: ANÁLISE DE VARIÂNCIA DOS FATORES NO TEMPO DE EXECUÇÃO DOS ALOCADORES.....	113
APÊNDICE D: ANÁLISE DE VARIÂNCIA DOS FATORES NO CONSUMO DE MEMÓRIA DOS ALOCADORES.....	115

LISTA DE TABELAS

TABELA 2.1 EXEMPLOS DE POLÍTICAS DE ALOCAÇÃO.....	10
TABELA 4.1. DADOS COLETADOS POR OPERAÇÃO DE ALOCAÇÃO.	31
TABELA 4.2. DADOS COLETADOS POR OPERAÇÃO DE DESALOCAÇÃO.	32
TABELA 4.3. ESPECIFICAÇÕES DO ARQUIVO DE ÁUDIO.....	34
TABELA 4.4. ESPECIFICAÇÕES DO ARQUIVO DE VÍDEO	34
TABELA 4.5. ALOCAÇÕES, TAMANHOS ALOCADOS E ROTINAS DE ALOCAÇÃO DO MYSQL.....	38
TABELA 4.6. RESULTADOS OBTIDOS NA CARACTERIZAÇÃO DO MYSQL.....	39
TABELA 4.7. ALOCAÇÕES, TAMANHOS ALOCADOS E ROTINAS DE ALOCAÇÃO DO CHEROKEE.	40
TABELA 4.8. DESALOCAÇÕES E TEMPO DE RETENÇÃO DO CHEROKEE.....	41
TABELA 4.9. ALOCAÇÕES, TAMANHOS ALOCADOS E ROTINAS DE ALOCAÇÃO DO CODEBLOCKS.....	41
TABELA 4.10. DESALOCAÇÕES E TEMPO DE RETENÇÃO DO CODEBLOCKS.	42
TABELA 4.11. ALOCAÇÕES, TAMANHOS ALOCADOS E ROTINAS DE ALOCAÇÃO DO VLCPLAYER(ÁUDIO).43	
TABELA 4.12. DESALOCAÇÕES E TEMPO DE RETENÇÃO DO VLCPLAYER (ÁUDIO).	44
TABELA 4.13. ALOCAÇÕES, TAMANHOS ALOCADOS E ROTINAS DE ALOCAÇÃO DO VLCPLAYER (VÍDEO).	45
TABELA 4.14. DESALOCAÇÕES E TEMPO DE RETENÇÃO DO VLCPLAYER (VÍDEO).....	45
TABELA 4.15. ALOCAÇÕES, TAMANHOS ALOCADOS E ROTINAS DE ALOCAÇÃO DO OCTAVE.....	46
TABELA 4.16. DESALOCAÇÕES E TEMPO DE RETENÇÃO DO OCTAVE.....	47
TABELA 4.17. ALOCAÇÕES, TAMANHOS ALOCADOS E ROTINAS DE ALOCAÇÃO DO INKSCAPE.....	48
TABELA 4.18. DESALOCAÇÕES E TEMPO DE RETENÇÃO DO INKSCAPE.....	48
TABELA 4.19 ALOCAÇÕES, TAMANHOS ALOCADOS E ROTINAS DE ALOCAÇÃO DO LYNX.....	49
TABELA 4.20. DESALOCAÇÕES E TEMPO DE RETENÇÃO DO LYNX.....	50
TABELA 4.21. MEMÓRIA ALOCADA POR APLICAÇÃO.	51
TABELA 4.22. PORCENTAGEM DO USO DAS ROTINAS DE ALOCAÇÃO.	52
TABELA 4.23. DESALOCAÇÃO POR APLICAÇÃO.....	53
TABELA 5.1. APLICAÇÕES AGRUPADAS DE ACORDO COM O SEU NÚMERO DE ALOCAÇÕES.	59
TABELA 5.2. APLICAÇÕES CARACTERIZADAS COM CONCENTRAÇÃO DE PELO MENOS 70% DE SUAS ALOCAÇÕES EM UM DOS NÍVEIS DEFINIDOS PARA O TA.....	59
TABELA 5.3. FATORES E NÍVEIS DO PROJETO EXPERIMENTAL.	60
TABELA 5.4. RANKING DOS ALOCADORES PELO TEMPO MÉDIO DE EXECUÇÃO.....	85
TABELA 5.5. RANKING DOS ALOCADORES PELO CONSUMO MÉDIO DE MEMÓRIA NOS TESTES FEITOS COM O PADRÃO $AL=PICO$	88
TABELA 5.6. RANKING DOS ALOCADORES PELO CONSUMO MÉDIO DE MEMÓRIA NOS TESTES FEITOS COM O PADRÃO $AL=USO CRESCENTE$	88
TABELA 6.1. PUBLICAÇÕES CIENTÍFICAS.....	92

LISTA DE FIGURAS

FIGURA 2.1. ESTRUTURAS GENÉRICAS DE UM ALOCADOR DE MEMÓRIA.	8
FIGURA 3.1. ESTRUTURAS DE DADOS DO ALOCADOR PTMALLOC2.....	18
FIGURA 3.2. ESTRUTURAS DE DADOS DO ALOCADOR PTMALLOC3.....	20
FIGURA 3.3. ESTRUTURAS DE DADOS DO ALOCADOR HOARD.....	22
FIGURA 3.4. ESTRUTURAS DE DADOS DO ALOCADOR MISER.	24
FIGURA 3.5. ESTRUTURAS DE DADOS DO ALOCADOR JEMALLOC.....	26
FIGURA 3.6. ESTRUTURAS DE DADOS DO ALOCADOR TCMALLOC.	28
FIGURA 4.1. DIAGRAMA DE FUNCIONAMENTO DO DEBUGMALLOC.....	31
FIGURA 4.2. TOPOLOGIA DO PROCESSADOR USADO NOS EXPERIMENTOS.	35
FIGURA 4.3. DISTRIBUIÇÃO DOS TAMANHOS ALOCADOS NO MySQL.	39
FIGURA 4.4. ALOCAÇÕES DE LONGA DURAÇÃO AGRUPADAS POR BLOCOS DE 1000 OPERAÇÕES (ALOCAÇÕES/DESALOCAÇÕES) DA APLICAÇÃO MySQL.....	39
FIGURA 4.5. DISTRIBUIÇÃO DOS TAMANHOS ALOCADOS PELO CHEROKEE.	40
FIGURA 4.6. ALOCAÇÕES DE LONGA DURAÇÃO AGRUPADAS POR BLOCOS DE 1000 OPERAÇÕES (ALOCAÇÕES/DESALOCAÇÕES) DA APLICAÇÃO CHEROKEE.	41
FIGURA 4.7. DISTRIBUIÇÃO DOS TAMANHOS ALOCADOS PELO CODEBLOCKS.	42
FIGURA 4.8. ALOCAÇÕES DE LONGA DURAÇÃO AGRUPADAS POR BLOCOS DE 1000 OPERAÇÕES (ALOCAÇÕES/DESALOCAÇÕES) DA APLICAÇÃO CODEBLOCKS.	42
FIGURA 4.9. DISTRIBUIÇÃO DOS TAMANHOS ALOCADOS PELO VLCPLAYER (ÁUDIO).	43
FIGURA 4.10. ALOCAÇÕES DE LONGA DURAÇÃO AGRUPADAS POR BLOCOS DE 1000 OPERAÇÕES (ALOCAÇÕES/DESALOCAÇÕES) DA APLICAÇÃO VLCPLAYER (ÁUDIO).	44
FIGURA 4.11. DISTRIBUIÇÃO DOS TAMANHOS ALOCADOS PELO VLCPLAYER (VÍDEO).	45
FIGURA 4.12. ALOCAÇÕES DE LONGA DURAÇÃO AGRUPADAS POR BLOCOS DE 1000 OPERAÇÕES (ALOCAÇÕES/DESALOCAÇÕES) DA APLICAÇÃO VLCPLAYER (VÍDEO).	46
FIGURA 4.13. DISTRIBUIÇÃO DOS TAMANHOS ALOCADOS PELO OCTAVE.	46
FIGURA 4.14. ALOCAÇÕES DE LONGA DURAÇÃO AGRUPADAS POR BLOCOS DE 1000 OPERAÇÕES (ALOCAÇÕES/DESALOCAÇÕES) DA APLICAÇÃO OCTAVE.	47
FIGURA 4.15. DISTRIBUIÇÃO DOS TAMANHOS ALOCADOS PELO INKSCAPE.	48
FIGURA 4.16. ALOCAÇÕES DE LONGA DURAÇÃO AGRUPADAS POR BLOCOS DE 1000 OPERAÇÕES (ALOCAÇÕES/DESALOCAÇÕES) DA APLICAÇÃO INKSCAPE.	49
FIGURA 4.17. DISTRIBUIÇÃO DOS TAMANHOS ALOCADOS PELO LYNX.	49
FIGURA 4.18. ALOCAÇÕES DE LONGA DURAÇÃO AGRUPADAS POR BLOCOS DE 1000 OPERAÇÕES (ALOCAÇÕES/DESALOCAÇÕES) DA APLICAÇÃO LYNX.	50
FIGURA 4.19. PORCENTAGEM DOS DEZ TAMANHOS MAIS ALOCADOS POR APLICAÇÃO.....	52
FIGURA 4.20. PROPORÇÃO DAS ALOCAÇÕES DE CURTA, MÉDIA E LONGA DURAÇÃO.....	53
FIGURA 5.1. TOPOLOGIA DE UM DOS QUATRO <i>sockets</i> DO PROCESSADOR UTILIZADO NOS TESTES.	56
FIGURA 5.2. PSEUDOCÓDIGO DA ETAPA DE EXECUÇÃO DO EXPERIMENTO DO PROGRAMA DE TESTE.....	58

FIGURA 5.3. TEMPO DE EXECUÇÃO DO PTMALLOC2 POR FATORES DE CARGA DE TRABALHO (NA x TA).	62
FIGURA 5.4. TEMPO DE EXECUÇÃO DO PTMALLOC2 POR NÚMERO DE PROCESSADORES E NÚMERO DE THREADS (NP x NT).	63
FIGURA 5.5. TEMPO DE EXECUÇÃO DO PTMALLOC2 POR NÚMERO DE PROCESSADORES, THREADS E NÚMERO DE ALOCAÇÕES (NP x NT x NA).	63
FIGURA 5.6. TEMPO DE EXECUÇÃO DO PTMALLOC2 POR NÚMERO DE PROCESSADORES, THREADS E TAMANHO DAS ALOCAÇÕES (NP x NT x TA).	63
FIGURA 5.7. TEMPO DE EXECUÇÃO DO PTMALLOC3 POR FATORES DE CARGA DE TRABALHO (NA x TA).	64
FIGURA 5.8. TEMPO DE EXECUÇÃO DO PTMALLOC3 POR NÚMERO DE PROCESSADORES E NÚMERO DE THREADS (NP x NT).	65
FIGURA 5.9. TEMPO DE EXECUÇÃO DO PTMALLOC3 POR NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E NÚMERO DE ALOCAÇÕES (NP x NT x NA).	65
FIGURA 5.10. TEMPO DE EXECUÇÃO DO PTMALLOC3 POR NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E TAMANHO DAS ALOCAÇÕES (NP x NT x TA).	65
FIGURA 5.11. TEMPO DE EXECUÇÃO DO HOARD POR FATORES DE CARGA DE TRABALHO (NA x TA).	66
FIGURA 5.12. TEMPO DE EXECUÇÃO DO HOARD POR NÚMERO DE PROCESSADORES E NÚMERO DE THREADS (NP x NT).	67
FIGURA 5.13. TEMPO DE EXECUÇÃO DO HOARD POR NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E NÚMERO DE ALOCAÇÕES (NP x NT x NA).	67
FIGURA 5.14. TEMPO DE EXECUÇÃO DO HOARD POR NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E TAMANHO DAS ALOCAÇÕES (NP x NT x TA).	67
FIGURA 5.15. TEMPO DE EXECUÇÃO DO MISER POR FATORES DE CARGA DE TRABALHO (NA x TA).	68
FIGURA 5.16. TEMPO DE EXECUÇÃO DO MISER POR NÚMERO DE PROCESSADORES E NÚMERO DE THREADS (NP x NT).	69
FIGURA 5.17. TEMPO DE EXECUÇÃO DO MISER POR NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E NÚMERO DE ALOCAÇÕES (NP x NT x NA).	69
FIGURA 5.18. TEMPO DE EXECUÇÃO DO MISER POR NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E TAMANHO DAS ALOCAÇÕES (NP x NT x TA).	69
FIGURA 5.19. TEMPO DE EXECUÇÃO DO JEMALLOC POR FATORES DE CARGA DE TRABALHO (NA x TA).	70
FIGURA 5.20. TEMPO DE EXECUÇÃO DO JEMALLOC POR NÚMERO DE PROCESSADORES E NÚMERO DE THREADS (NP x NT).	71
FIGURA 5.21. TEMPO DE EXECUÇÃO DO JEMALLOC POR NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E NÚMERO DE ALOCAÇÕES (NP x NT x NA).	71
FIGURA 5.22. TEMPO DE EXECUÇÃO DO JEMALLOC POR NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E NÚMERO DE ALOCAÇÕES (NP x NT x NA).	71
FIGURA 5.23. TEMPO DE EXECUÇÃO DO TCMALLOC POR FATORES DE CARGA DE TRABALHO (NA x TA).	72
FIGURA 5.24. TEMPO DE EXECUÇÃO DO TCMALLOC POR NÚMERO DE PROCESSADORES E NÚMERO DE THREADS (NP x NT).	73
FIGURA 5.25. TEMPO DE EXECUÇÃO DO TCMALLOC POR NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E NÚMERO DE ALOCAÇÕES (NP x NT x NA).	73

FIGURA 5.26. TEMPO DE EXECUÇÃO DO TCMALLOC POR <i>NÚMERO DE PROCESSADORES, NÚMERO DE THREADS E TAMANHO DAS ALOCAÇÕES (NP x NT x TA)</i> .	73
FIGURA 5.27. CONSUMO DE MEMÓRIA DO PTMALLOC2 POR <i>NÚMERO DE ALOCAÇÕES E TAMANHOS DA ALOCAÇÕES (NA x TA)</i> .	74
FIGURA 5.28. CONSUMO DE MEMÓRIA DO PTMALLOC2 POR <i>NÚMERO DE THREADS (NT)</i> .	75
FIGURA 5.29. CONSUMO DE MEMÓRIA DO PTMALLOC2 POR <i>PADRÃO DE ALOCAÇÕES DE LONGA DURAÇÃO (AL)</i> .	75
FIGURA 5.30. CONSUMO DE MEMÓRIA DO PTMALLOC3 POR <i>NÚMERO DE ALOCAÇÕES E TAMANHOS DA ALOCAÇÕES (NA x TA)</i> .	75
FIGURA 5.31. CONSUMO DE MEMÓRIA DO PTMALLOC3 POR <i>NÚMERO DE THREADS (NT)</i> .	76
FIGURA 5.32. CONSUMO DE MEMÓRIA DO PTMALLOC3 POR <i>PADRÃO DAS ALOCAÇÕES DE LONGA DURAÇÃO (AL)</i> .	76
FIGURA 5.33. CONSUMO DE MEMÓRIA DO HOARD POR <i>NÚMERO DE ALOCAÇÕES E TAMANHOS DA ALOCAÇÕES (NA x TA)</i> .	77
FIGURA 5.34. CONSUMO DE MEMÓRIA DO HOARD POR <i>NÚMERO DE PROCESSADORES (NP)</i> .	77
FIGURA 5.35. CONSUMO DE MEMÓRIA DO HOARD POR <i>NÚMERO DE THREADS (NT)</i> .	77
FIGURA 5.36. CONSUMO DE MEMÓRIA DO HOARD POR <i>PADRÃO DAS ALOCAÇÕES DE LONGA DURAÇÃO (AL)</i> .	78
FIGURA 5.37. CONSUMO DE MEMÓRIA DO MISER POR <i>NÚMERO DE ALOCAÇÕES E TAMANHOS DA ALOCAÇÕES (NA x TA)</i> .	78
FIGURA 5.38. CONSUMO DE MEMÓRIA DO MISER POR <i>NÚMERO DE THREADS (NT)</i> .	79
FIGURA 5.39. CONSUMO DE MEMÓRIA DO MISER POR <i>PADRÃO DAS ALOCAÇÕES DE LONGA DURAÇÃO (AL)</i> .	79
FIGURA 5.40. CONSUMO DE MEMÓRIA DO JEMALLOC POR <i>NÚMERO DE ALOCAÇÕES E TAMANHOS DAS ALOCAÇÕES (NA x TA)</i> .	79
FIGURA 5.41. CONSUMO DE MEMÓRIA DO JEMALLOC POR <i>NÚMERO DE THREADS (NT)</i> .	80
FIGURA 5.42 CONSUMO DE MEMÓRIA DO JEMALLOC POR <i>PADRÃO DAS ALOCAÇÕES DE LONGA DURAÇÃO (AL)</i> .	80
FIGURA 5.43. CONSUMO DE MEMÓRIA DO TCMALLOC POR <i>NÚMERO DE ALOCAÇÕES E TAMANHOS DAS ALOCAÇÕES (NA x TA)</i> .	81
FIGURA 5.44. CONSUMO DE MEMÓRIA DO TCMALLOC POR <i>NÚMERO DE THREADS (NT)</i> .	81
FIGURA 5.45. CONSUMO DE MEMÓRIA DO TCMALLOC POR <i>PADRÃO DAS ALOCAÇÕES DE LONGA DURAÇÃO (AL)</i> .	81
FIGURA 5.46. TEMPO DE EXECUÇÃO MÉDIO POR ALOCADOR DE MEMÓRIA.	82
FIGURA 5.47. TEMPO DE EXECUÇÃO MÉDIO POR ALOCADOR AGRUPADO PELO <i>NÚMERO DE PROCESSADORES (NP)</i> .	83
FIGURA 5.48. TEMPO DE EXECUÇÃO MÉDIO POR ALOCADOR AGRUPADO PELO <i>NÚMERO DE PROCESSADORES E NÚMERO DE THREADS (NP x NT)</i> .	84
FIGURA 5.49. TEMPO DE EXECUÇÃO MÉDIO POR ALOCADOR AGRUPADO PELO <i>NÚMERO DE ALOCAÇÕES (NA)</i> .	84
FIGURA 5.50. TEMPO DE EXECUÇÃO MÉDIO POR ALOCADOR AGRUPADO PELO <i>TAMANHO DAS ALOCAÇÕES (TA)</i> .	85
FIGURA 5.51. CONSUMO MÉDIO DE MEMÓRIA POR ALOCADOR.	86

FIGURA 5.52. CONSUMO MÉDIO DE MEMÓRIA DOS ALOCADORES POR PADRÃO DE ALOCAÇÕES DE LONGA DURAÇÃO.....	86
FIGURA 5.53. CONSUMO DE MEMÓRIA DOS ALOCADORES POR COMBINAÇÃO DOS FATORES NA E AL. ...	87
FIGURA 5.54. CONSUMO DE MEMÓRIA DOS ALOCADORES POR COMBINAÇÃO DOS FATORES TA E AL.....	87

LISTA DE ABREVIATURAS E SIGLAS

AL	Padrão de Distribuição das Alocações de Longa Duração
CPU	<i>Central Processing Unit</i>
DOE	<i>Design of Experiments</i>
GB	Gigabytes
GC	<i>Garbage Collector</i>
HD	<i>High-definition</i>
IDE	<i>Integrated Development Environment</i>
KB	Kilobytes
KMA	<i>Kernel Memory Allocator</i>
LIFO	<i>Last In, First Out</i>
MB	Megabytes
NA	Número de Alocações
NP	Número de Processadores
NT	Número de <i>Threads</i>
RSS	<i>Resident Set Size</i>
SGBD	Sistema de Gerenciamento de Banco de Dados
SO	Sistema Operacional
TA	Tamanho das Alocações
TF	Tamanho Máximo da Alocação
TI	Tamanho Mínimo da Alocação
TLAB	<i>Thread Local Allocation Buffer</i>
TMA	Tamanhos Mais Alocados
TPS	Transações por Segundo
UMA	<i>User Memory Allocator</i>

1 INTRODUÇÃO

1.1 Contextualização

Em engenharia de sistemas computacionais, o gerenciamento de memória principal tem um impacto importante no desempenho e escalabilidade do software. Em geral, aplicações mais sofisticadas necessitam frequentemente alocar dinamicamente porções de memória de tamanhos variados durante suas execuções. Quando se trata de aplicações mais sofisticadas (ex. SGBD), a grande frequência destas operações causa um impacto significativo no tempo de execução e uso de memória da aplicação (Ferreira et al., 2011a).

O conjunto de rotinas responsáveis pela alocação dinâmica de memória é denominado alocador de memória (Vahalia, 1996). Alocadores de memória existem em dois níveis, no *kernel-level* e no *user-level*. No primeiro, essas operações são implementadas por um KMA (*kernel memory allocator*) disponibilizado como parte do núcleo do sistema operacional. No segundo, as operações de alocação de memória são implementadas por um UMA (*user memory allocator*), em geral localizado dentro da biblioteca padrão do sistema operacional (Ferreira et al., 2011c).

Em nível de *kernel*, a alocação dinâmica de memória tem como principal objetivo fornecer porções de memória aos subsistemas do *kernel* para executar suas funções e armazenar suas estruturas. Por exemplo, quando uma aplicação requisita a abertura de um arquivo qualquer, o subsistema de gerenciamento de arquivos necessita alocar memória para diversas estruturas utilizadas no armazenamento de metadados sobre o arquivo sendo aberto. Esta memória deve ser alocada dinamicamente, no ato da abertura do arquivo, a qual é solicitada pelo subsistema de gerenciamento de arquivos ao KMA em uso.

Já no nível de usuário, o alocador de memória tem como principal função gerenciar a área da *heap* (Vahalia, 1996). A área da *heap* corresponde a uma porção de memória, dentro do espaço de endereçamento de um processo, para onde as operações de gerenciamento dinâmico de memória são direcionadas. Esse gerenciamento é normalmente realizado por meio de funções como *malloc()*, *realloc()* e *free()*, as quais são implementadas como parte do alocador de memória. Nos casos onde a requisição ultrapassa o tamanho disponível no espaço de memória da *heap* do processo, o alocador pode solicitar mais memória para o sistema operacional a fim de atender a requisição. Essa porção de memória adicional é incorporada ao conjunto de páginas de memória da *heap* do processo que é gerenciado pelo alocador (Matias et al., 2011).

Em ambos os níveis citados anteriormente, há uma implementação de um alocador de memória que é automaticamente acionada pelo programa do usuário durante sua execução, sem que haja necessidade de envolvimento do programador neste processo. Como isto se dá de forma transparente, ou seja, a escolha do alocador de memória, principalmente em nível de usuário, normalmente este processo não é percebido durante o desenvolvimento de um *software*. Em *user-level*, o alocador é parte integrante do processo de aplicação, normalmente seu código é parte de uma biblioteca ligada ao espaço de endereçamento da aplicação, como a *libc* (ou *glibc*) (Ferreira et al., 2011b). Desta forma, sendo parte de uma biblioteca, é possível alterar uma aplicação para que esta utilize um determinado UMA que melhor gerencie suas requisições de memória, em alternativa ao uso do UMA padrão do sistema operacional. O mesmo se aplica ao KMA adotado em um dado sistema operacional.

É comum que aplicações mais sofisticadas não usem o UMA padrão do sistema, pois sendo este um alocador de propósito geral seu projeto não é otimizado para atender necessidades específicas destas aplicações. O uso de múltiplas *threads* em sistemas multiprocessados é um exemplo destas necessidades que tem um impacto significativo no desempenho do UMA (Berger et al., 2000). Por este motivo, aplicações sofisticadas, como o servidor *web* Apache (Apache Software Foundation, 2014b), o gerenciador de banco de dados PostgreSQL (PostgreSQL Development Group, 2014) e o navegador *web* Firefox (Mozilla Foundation, 2014), se utilizam de implementações próprias de UMAs (Costa et al., 2013). Atualmente, existem diferentes algoritmos de UMA disponíveis, com diferentes estratégias de gerenciamento de memória como alternativa ao alocador padrão.

Este trabalho se concentra nos alocadores do tipo UMA, ou seja, alocadores de memória em *user-level*. Dessa forma, o termo alocador, utilizado daqui em diante neste documento, irá se referir apenas ao UMA.

1.2 Relevância do Trabalho

O desempenho de operações de alocação de memória tem significativa influência no desempenho global da maioria das aplicações computacionais. Nesse sentido, a seleção de um alocador de memória é um importante requisito no projeto de sistemas computacionais mais sofisticados. A forte correlação entre o perfil de uso dinâmico da memória com o desempenho do alocador exige que a seleção do alocador ocorra por meio de um estudo experimental. Em (Matias et al., 2011) o software MySQL (Oracle Corporation, 2014) foi submetido a um estudo experimental e comparativo com cinco alocadores, tendo apresentado diferenças significativas ao se alterar o alocador padrão para um dos cinco alocadores avaliados. Resultados similares publicados em (Matias et al., 2011), obtidos com um *middleware* de missão crítica, corroboram esses resultados.

Compreender o uso de memória de uma aplicação, bem como o funcionamento dos diferentes alocadores disponíveis atualmente, a fim de se selecionar o mais adequado para determinada aplicação, exige um extenso trabalho experimental. Esta pesquisa visa contribuir para o campo de estudo em alocação dinâmica de memória, em duas frentes: i) na caracterização do uso de

memória de diferentes grupos de aplicações e *ii*) na avaliação de diferentes alocadores largamente utilizados atualmente.

Através do estudo de caracterização do uso de memória por diferentes tipos de aplicações, foi possível entender melhor o comportamento da alocação dinâmica de memória em diferentes classes de aplicações, facilitando a identificação de padrões gerais e específicos de uso de memória nestes diferentes tipos de aplicações. Tal conhecimento foi usado neste trabalho para planejar os testes de avaliação experimental dos diferentes alocadores investigados, contudo estes resultados podem ser usados também para diferentes trabalhos no campo da simulação e avaliação quantitativa de algoritmos de alocação de memória.

Já na avaliação dos alocadores, foi possível estabelecer quais alocadores são mais adequados para determinados cenários de uso de memória ou perfil de execução, tais como o número de *threads* e processadores. Através dessas informações foi possível identificar os principais fatores que influenciaram diretamente no desempenho dos alocadores em termos de tempo de resposta e consumo de memória. Como resultado, tem-se uma coleção de evidências experimentais importantes para a tomada de decisão sobre qual alocador adotar para um dado perfil de uso de memória da aplicação.

1.3 Objetivos

Geral:

- Avaliar teórica e experimentalmente o tempo de resposta e consumo de memória de diferentes alocadores de memória, em nível de usuário, considerando perfis de carga de trabalho típicos de diferentes categorias de aplicações.

Específicos:

- Estudar o funcionamento de seis alocadores de memória, usados largamente, mapeando suas principais estruturas de dados e estratégias de alocação adotadas.
- Caracterizar a utilização de memória de um grupo representativo de diferentes tipos de aplicações, visando compreender como essas aplicações utilizam a memória principal.
- Avaliar quantitativamente e comparativamente o desempenho (tempo de resposta e espaço usado) dos seis alocadores investigados.
- Identificar e quantificar os principais fatores que influenciam o desempenho dos alocadores investigados. Tais fatores podem ser oriundos do ambiente de execução, como o número de processadores ou provenientes da própria aplicação, tais como o número de *threads*, a quantidade de alocações e o tamanho das alocações.
- Identificar quais estratégias adotadas pelos alocadores melhor se beneficiam dos fatores do ambiente de execução e da própria aplicação.

1.4 Métodos Adotados

Este trabalho se caracteriza por ser uma pesquisa de natureza teórica e experimental (Wazlawick, 2009). Para atingir os objetivos propostos na Seção 1.3, o estudo foi dividido em três etapas.

Na primeira etapa foi realizado um estudo teórico de diferentes alocadores de memória usados atualmente, visando compreender o funcionamento detalhado destes alocadores. Foram escolhidos para o estudo os seguintes alocadores: Hoard (v3.8) (Berger et. al, 2000), Ptmalloc (v2) (Gloger, 2006), Ptmalloc (v3) (Gloger, 2006), TCMalloc (v1.5) (Ghemawat e Menage, 2014), Jemalloc (v2.0.1) (Evans, 2006) e Miser (Tannenbaum, 2010). Estes alocadores foram selecionados por já terem sido usados em trabalhos da literatura científica correlata. A análise dos alocadores contou tanto com o estudo da documentação disponível, para cada alocador, quanto da análise dos seus códigos-fonte.

Na segunda etapa, foi realizada a caracterização das alocações dinâmicas de memória de um grupo de aplicações. O objetivo desta etapa foi compreender, através de uma análise experimental, como cada tipo de aplicação realiza as suas alocações dinâmicas de memória, bem como identificar possíveis padrões de uso de memória. Os resultados desta etapa foram usados, principalmente, para o planejamento dos experimentos de avaliação dos alocadores, tais como o estabelecimento de cargas de trabalho comumente observadas em aplicações reais.

Na terceira e última etapa desta pesquisa, foi realizada a avaliação experimental de cada alocador investigado. Para tanto, foi utilizada a técnica de planejamento estatístico de experimentos, DOE (*Design of Experiments*) (Montgomery, 2000), a fim de se planejar e executar experimentos controlados, bem como analisar estatisticamente seus resultados. Nessa etapa, cada alocador avaliado foi avaliado com base em um projeto fatorial completo, com 30 replicações visando o melhor tratamento do erro experimental.

Os fatores avaliados na última etapa foram: número de alocações, tamanho das alocações, número de processadores, número de *threads* e padrão de comportamento das alocações de longa duração. Esses fatores foram variados de acordo com os níveis obtidos na caracterização de memória, de modo a emular diferentes cenários de aplicações reais. Para cada teste foi coletado o tempo total gasto na execução e o espaço total gasto pela aplicação. Tais resultados foram analisados utilizando a técnica de análise de variância (ANOVA) (Montgomery, 2000), a fim de identificar os principais fatores e interações de fatores que afetam o tempo de execução e o consumo de memória dos alocadores. Para as interações significativas, realizou-se a comparação múltipla das médias por meio do teste de Tukey, ao nível de significância de 5% ($\alpha = 0,05$). Através desta análise, foi possível estabelecer um *ranking* dos alocadores com relação ao tempo e espaço usados, bem como observar qual deles se comporta melhor em determinados cenários.

1.5 Contribuições da Pesquisa

Este estudo contribui com a área de alocação dinâmica de memória por meio de duas frentes: contribuição por meio dos resultados obtidos na caracterização das alocações dinâmicas de memória e a contribuição através da avaliação experimental de seis alocadores de memória amplamente utilizados atualmente.

Com a caracterização das alocações dinâmicas de diferentes tipos de aplicação, objetivou-se contribuir na identificação de um conjunto de padrões de alocação de memória. Estes padrões no uso de memória podem ser utilizados para a geração de cargas de trabalho sintéticas que representem de maneira fidedigna o comportamento das aplicações reais, com aplicações em diferentes campos de pesquisa na área, tais como projeto de novos alocadores, avaliação de desempenho de sistemas, entre outros. Além disso, os padrões no uso de memória podem ser explorados para a definição de estratégias no projeto de novos alocadores.

Através da avaliação experimental e comparativa dos alocadores de memória objetivou-se contribuir com os seguintes achados:

- Levantamento dos fatores da aplicação e do ambiente de execução que mais afetam o comportamento do alocador com relação ao seu desempenho e consumo de memória.
- Compreensão da eficiência de cada um dos algoritmos de alocação de memória em determinados cenários reais de execução.
- Estabelecimento de um *ranking* dos alocadores avaliados, com respeito ao tempo de execução e ao consumo de memória, face a diferentes cenários de execução. Este *ranking* pode servir para orientar a escolha do alocador apropriado para uma dada aplicação ou classe de aplicações.

1.6 Estrutura do Documento

Esta dissertação está organizada da seguinte forma.

O Capítulo 2 apresenta a revisão da literatura correlata, as principais características de um alocador de memória, os principais desafios no gerenciamento das alocações dinâmicas e a fundamentação teórica que serviu de base para este estudo.

No Capítulo 3 será feita a descrição detalhada dos alocadores investigados neste trabalho, apresentando as principais estruturas de dados, políticas de alocação e as soluções adotadas pelos alocadores para lidar com os problemas oriundos da alocação dinâmica de memória.

O Capítulo 4 apresenta a caracterização das alocações dinâmicas de memória em um grupo de diferentes tipos de aplicações. Neste capítulo, são detalhadas quais aplicações foram utilizadas, bem como o método adotado para cada teste de caracterização, já que cada tipo de aplicação exigiu abordagem de caracterização diferente. O capítulo é finalizado com a análise dos resultados dos experimentos de caracterização, descrevendo os principais padrões encontrados.

No Capítulo 5 é apresentada a avaliação experimental comparativa dos alocadores investigados nessa pesquisa. Nesse capítulo, é descrito como foi realizado o planejamento dos experimentos, a análise dos seus resultados e as conclusões obtidas.

Por fim, o Capítulo 6 apresenta as considerações finais sobre a pesquisa, destacando seus principais resultados, suas limitações, as principais dificuldades encontradas e os futuros trabalhos propostos para sua continuidade.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Introdução

O alocador de memória é o código responsável pelo gerenciamento das alocações de memória dinâmica de uma aplicação. A função primária do alocador de memória é o gerenciamento da *heap* (Vahalia, 1996), a porção de memória localizada na região de dados do processo que é usada para atender requisições de alocações dinâmicas de memória.

Quando uma aplicação faz a primeira alocação dinâmica de memória, o alocador associado cria e inicializa as estruturas de dados que usa para gerenciar a memória. Em geral, neste momento o alocador faz uma chamada de sistema para o Sistema Operacional (SO) requisitando espaço de memória para a *heap* da aplicação. As chamadas de sistema normalmente possuem um tempo de resposta alto, por conta disso, o alocador requisita uma área suficiente para que futuras requisições da aplicação sejam atendidas, sem a necessidade de novas chamadas para o SO.

Uma das funções do alocador é o de mapear quais áreas da *heap* da aplicação estão em uso e quais estão livres. Para isso, o alocador precisa manter na *heap* uma estrutura de dados responsável por mapear os blocos de memória. Essa estrutura de dados é chamada de diretório. Os blocos de memória ganham uma entrada no diretório à medida que são liberados pela aplicação, e seu local nesta entrada é, normalmente, definido de acordo com o seu tamanho, facilitando assim a sua busca. O diretório é usualmente implementado no cabeçalho da *heap*.

O bloco de memória, por sua vez, é a área que será destinada de fato para o uso da aplicação. Ao receber uma requisição para alocação, o alocador irá procurar no diretório de sua *heap* por uma entrada que contenha um bloco de memória igual ou maior ao requisitado. Este bloco sairá do registro do diretório enquanto estiver em uso pelo processo, voltando apenas quando este o liberar por meio de uma rotina de liberação de memória (ex. *free*). Por questões de espaço, o diretório não consegue referenciar todos os blocos de memória de uma *heap*. Cada bloco de memória livre também possui um ponteiro para o próximo bloco livre de sua mesma categoria (que pode ser por tamanho, por localidade na memória, etc.). Na Figura 2.1 são apresentadas as estruturas genéricas de um alocador de memória. A identidade visual apresentada nessa figura será usada no Capítulo 3, para explicar as estruturas de dados de cada um dos alocadores investigados neste trabalho.

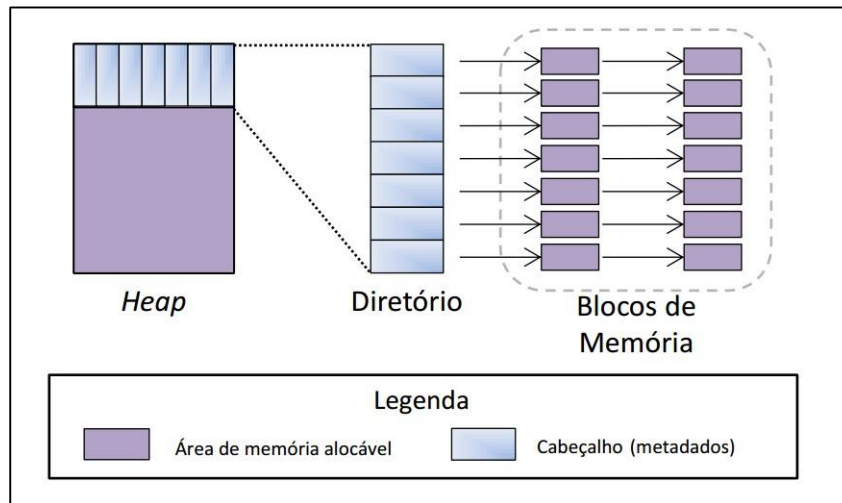


Figura 2.1. Estruturas genéricas de um alocador de memória.

2.2 Características de um Alocador de Memória

O objetivo do projeto de um alocador de memória é o de gerenciar a memória dinâmica da aplicação, respondendo imediatamente às suas requisições de alocação/desalocação utilizando o mínimo de memória (*overhead*) possível. Um alocador ideal é aquele que alocaria a quantidade de memória exata solicitada e responderia a todas as requisições em menor tempo possível (Wilson et al., 1995).

A aplicação aloca e desaloca blocos de memória de acordo com a sua necessidade, portanto, o alocador não possui qualquer controle do fluxo de alocação requisitado pela aplicação. Basicamente, a única escolha que cabe ao alocador é a de decidir qual porção de memória livre será usada para atender ao pedido da aplicação. Essa decisão não pode ser mudada posteriormente, o alocador não pode compactar a memória alocada, movendo blocos alocados de forma a os tornar contíguos, a fim de liberar espaço para alocações maiores.

O alocador de memória é, portanto, um algoritmo *online*, que precisa responder imediatamente às requisições feitas pela aplicação na sua estrita sequência, e suas decisões são irrevogáveis (Wilson et al., 1995).

O elemento mais importante dos algoritmos dos alocadores de memória é a sua política de alocação. A política de alocação é a estratégia adotada pelo alocador para dividir a *heap* nos blocos de memória e escolher qual bloco livre será utilizado para atender à requisição da aplicação. Um alocador pode utilizar apenas uma ou mesclar várias políticas, dependendo das condições de alocação. Uma política de alocação comum e utilizada em certo nível pelos alocadores investigados neste estudo é o *best-fit*. Essa estratégia busca atender à requisição da aplicação atribuindo o bloco de menor tamanho que seja suficiente para atender a requisição. É uma estratégia que visa minimizar o espaço de memória gasto pela aplicação, mas que pode gerar um tempo de resposta alto, em decorrência da busca pelo melhor bloco.

As políticas de alocação são classificadas em duas categorias: as *Sequential Fits* (Wilson et al., 1995), onde a divisão dos blocos é feita de acordo com a

necessidade da aplicação; e as *Segregated Fits* (Collins, 1961), onde a divisão dos blocos de memória na *heap* é feita previamente segundo um padrão específico, de modo a facilitar os algoritmos de busca. A Tabela 2.1 apresenta algumas políticas de alocação comumente utilizadas. Existem inúmeras variações das políticas de alocação, especialmente quando mais de um bloco pode ser selecionado para atender à requisição. Nesse caso o alocador pode retornar o primeiro bloco que encontrar e privilegiar o desempenho, ou retornar o bloco mais próximo da última alocação para privilegiar o uso da memória *cache*. Todas essas decisões ficam a cargo do alocador de memória e é comum que os alocadores modernos mesquem diferentes estratégias a fim de se explorar o melhor aspecto de cada uma delas, relevante para o tipo de alocação que está sendo requisitado.

Dois mecanismos comuns encontrados nas estratégias dos alocadores são o *Splitting* e *Coalescing* (Wilson et al., 1995). Cada requisição de memória só pode ser atendida por um único bloco, uma vez que o bloco já esteja alocado ele não pode ser usado para atender nenhuma outra requisição. Dessa forma, se faz necessário dividir os blocos de memória para suprir requisições menores, a fim de se evitar o desperdício de memória. Este processo de divisão dos blocos é chamado de *Splitting*, e pode ser implementado de diferentes maneiras. A implementação do *Splitting* faz parte da estratégia do alocador, alguns alocadores só dividem blocos em tamanhos que resultem em uma potência de dois bytes (Tannenbaum, 2010), enquanto outros utilizam blocos com tamanhos múltiplos de oito bytes (Gloger, 2006). Além disso, quanto menor o bloco, uma maior porcentagem dele será utilizada para guardar os metadados de gerência do alocador, resultando em um desperdício maior. Portanto, a decisão de como fazer o *Splitting* influencia diretamente no uso de memória do próprio alocador, bem como nas fragmentações interna e externa da memória gerenciada (ver Seção 2.3).

Por sua vez, quando um bloco é liberado, é possível que este possua adjacente ao seu endereço de memória um ou mais blocos livres, liberados anteriormente. Nestes casos, o alocador pode escolher juntar dois ou mais blocos adjacentes livres, a fim de gerar um bloco maior, apto a atender requisições maiores. O mecanismo de coalisção dos blocos livres é chamada de *Coalescing*. Assim como no *Splitting*, parte da estratégia do alocador envolve decidir como ou em que momento a *Coalescing* será feita. O alocador pode, por exemplo, escolher realizar a *Coalescing* de dois ou mais blocos apenas quando uma nova requisição de memória não for atendida. Dessa forma, ele evita que a rotina de liberação da memória tome muito tempo mesclando inúmeros blocos, enquanto ainda há memória disponível.

Todas essas características estão presentes em todos os alocadores investigados. É natural, no entanto, que cada alocador busque desenvolver os seus próprios mecanismos e estratégias, visando se adequar melhor a um tipo de uso de memória de um conjunto de aplicações.

Tabela 2.1 Exemplos de políticas de alocação.

Sequential Fits (Wilson et al., 1995)			
Política de Alocação	Diretiva básica	Vantagens	Desvantagens
Best-fit	Procura o menor bloco possível para satisfazer a requisição da aplicação.	Menor consumo de memória.	Maior tempo de resposta em decorrência da busca.
First-fit	Procura o primeiro bloco possível para satisfazer a requisição da aplicação.	Busca simplificada, tempo de resposta reduzido.	Gera desperdício de memória.
Next-fit	Procura o bloco mais próximo do último bloco a ser alocado, para satisfazer a requisição.	Tende a gerar uma boa localidade de referência, uma vez que a memória alocada já pode estar na <i>cache</i> .	Pode gerar desperdício de memória.
Segregated Fits (Collins, 1961)			
Política de Alocação	Diretiva básica	Vantagens	Desvantagens
Segregated Free List	Segrega uma porção da <i>heap</i> com blocos do mesmo tamanho (<i>buffer</i>).	Simplifica o gerenciamento e a busca da memória. Como o número de blocos é conhecido os metadados podem ser centralizados e a busca pode ser feita com um tempo constante.	Gera desperdício de memória quando as requisições não forem do tamanho gerenciado (fragmentação interna).
Binary Buddies	A <i>heap</i> é dividida em duas áreas (<i>buddies</i>) de forma recursiva, de acordo com a requisição da aplicação.	Metadados centralizados, simplifica a busca. Simplifica a junção de blocos livres (<i>Coalescence</i>).	Gera desperdício de memória quando as requisições não forem múltiplos de dois (fragmentação interna).

2.3 Desafios do Gerenciamento de Alocações Dinâmicas de Memória

Em decorrência da maneira como a aplicação aloca e desaloca dinamicamente a memória, juntamente com a política de alocação do alocador, o alocador pode deixar espaços livres de memória após um tempo de operação, os quais não podem ser utilizados para atender outras requisições. Este comportamento gera um desperdício de memória e é chamado de fragmentação. Tradicionalmente, a fragmentação pode ser classificada como fragmentação externa ou interna (Randell, 1969).

A fragmentação externa ocorre quando há memória suficiente para atender requisições, mas que não pode ser usada para suprir a aplicação. Normalmente, isso ocorre porque os blocos livres possuem um tamanho muito pequeno, e não estão posicionados continuamente na memória, onde a técnica de *Coalescing* poderia criar blocos maiores.

A fragmentação interna, por sua vez, aparece quando o bloco destinado a atender a requisição da memória é maior do que o que será usado de fato pela aplicação. Por conseguinte, parte da memória do bloco é desperdiçada e não pode ser usada por outra requisição. Esta fragmentação é chamada de interna, por ocorrer dentro do bloco de memória.

Em ambos os casos de fragmentação, o alocador irá requisitar mais memória para o Sistema Operacional, mesmo possuindo memória livre que seria suficiente para atender a requisição. Embora algumas estratégias visem reduzir o nível de fragmentação imposta por um determinado padrão de alocação, de forma geral, é difícil garantir que ela não ocorra. Já foi provado que para cada algoritmo de alocação possível, sempre haverá uma ordem em que as alocações e desalocações de uma aplicação resultem em uma severa fragmentação da memória (Robson, 1971) (Garey et al., 1972) (Robson, 1974) (Robson, 1977). Portanto, cada alocador adota um conjunto de estratégias e mecanismos específicos que visam reduzir o nível de fragmentação da memória, dado um padrão de alocação.

Além da fragmentação de memória, os alocadores precisam lidar com problemas que norteiam o gerenciamento de memória em ambientes multiprocessados, como a contenção de memória, o *memory blowup* e o *false sharing*. A contenção de memória (M. Masmano, 2006) é caracterizada pela disputa entre duas ou mais *threads* pela mesma área da *heap*. Quando há a possibilidade de compartilhamento de uma mesma área da *heap* por mais de uma *thread*, o alocador precisa evitar que seus metadados se tornem inconsistentes, utilizando mecanismos de exclusão mútua para o controle de concorrência. Se uma *thread* tentar acessar a *heap* compartilhada que já esteja em uso por outra *thread*, aquela que tentou acessar ficará impedida de continuar a execução de suas tarefas até que a *heap* seja liberada. É fácil concluir que a contenção de memória pode prejudicar o tempo de resposta de uma aplicação que utilize múltiplas *threads* em um ambiente multiprocessado.

O *Memory Blowup* (Berger et al., 2000) é um tipo específico de fragmentação que ocorre em aplicações *multithreaded*. É comum que os alocadores atribuam a cada *thread* (ou a um conjunto de *threads*) uma área de memória exclusiva, a fim de se evitar a concorrência com outras *threads* nas alocações e desalocações. Por não ser uma área compartilhada, o alocador não pode utilizar essa área para atender requisições das demais *threads*. Esse comportamento faz com que o alocador requisiite memória ao sistema operacional continuamente, mesmo quando possui memória suficiente em sua *heap*.

O *False Sharing* (Krishnan, 1999), ou falso compartilhamento, ocorre quando duas ou mais *threads* estão usando blocos de memória mapeados na mesma linha de *cache* do processador. Este mapeamento geralmente acontece quando tais *threads* acessam memória de endereços muito próximos. Enquanto as *threads* que compartilham a linha de *cache* estão apenas lendo o conteúdo da memória, o falso compartilhamento não gera nenhum tipo de prejuízo. Os problemas decorrentes do *false sharing* aparecem quando há necessidade de escrita em algum endereço mapeado na linha de *cache*. Neste caso, cada vez que uma *thread* tenta acessar um conteúdo da linha de *cache* compartilhada, se outra *thread* diferente desta realizou alguma escrita, o conteúdo será recarregado da memória principal para a memória *cache*, provocando uma perda de desempenho.

2.4 Trabalhos Relacionados

Nesta Seção serão apresentados trabalhos relacionados no estudo da gerência de alocações dinâmicas de memória.

Em (Grunwald et al., 1993), os autores apresentam a importância da memória *cache* e de algoritmos que privilegiem o seu uso no desempenho de um alocador de memória. Quando um processador necessita de algum dado, este verifica, inicialmente, se o dado está presente na memória *cache*; caso positivo (*cache hit*), o dado é transferido para o processador em alta velocidade; caso contrário (*cache miss*), o dado é recuperado da memória principal, sendo que este mecanismo pode exigir vários ciclos de processamento. Uma alta taxa de *cache miss* pode acarretar em uma queda de desempenho de até 25% do tempo total de uma aplicação, portanto, é um fator de extrema importância para aplicações de alto-desempenho.

Com a responsabilidade de gerenciar a memória da aplicação, o alocador tem um papel determinante na taxa de *cache miss*. O trabalho avalia cinco alocadores, em quatro diferentes aplicações, analisando-os com relação ao tempo de execução da aplicação e a sua referência de localidade. O trabalho conclui que a política de alocação de um algoritmo tem uma influência significativa na taxa de *cache miss* de uma aplicação. Adicionalmente, o estudo apresentou que as políticas de alocação que privilegiam o uso otimizado da memória, como *first-fit* e *best-fit* não possuem uma boa localidade de referência. Em contrapartida, mecanismos como *Coalescing* (ver Seção 2.2), embora aumentem o tempo de resposta do alocador, ajudam a melhorar a taxa de *cache hit* de uma aplicação. Os alocadores que tiveram o melhor uso da CPU foram aqueles com o uso mais eficiente da *cache*. Esses algoritmos possuíam um uso mais eficiente da *cache* devido ao seu mecanismo de alinhamento dos blocos de memória. Tais alocadores alocavam um número pequeno de tamanhos diferentes, arredondando as requisições das aplicações para um dos tamanhos gerenciados. Tal estratégia gera uma alta fragmentação interna, mas faz com que os objetos possam ser mais facilmente reutilizados por outras requisições.

(Wilson et al., 1995) apresenta uma revisão sistemática da literatura com relação ao estudo dos alocadores de memória. É um dos trabalhos mais referenciados na área, servindo de base para inúmeros outros artigos. O trabalho apresenta as características básicas de um alocador (ver Seção 2.2), principais estratégias, mecanismos e desafios inerentes à gerência de memória até metade da década de 90. O estudo cataloga e estabelece uma taxonomia para as estratégias utilizadas pelos alocadores. Tais estratégias permeiam implementações até dos mais recentes alocadores, tais como as políticas de alocação e mecanismos como o *Coalescing* e o *Splitting*. Outra contribuição relevante dessa revisão sistemática é o estudo acerca da fragmentação de memória, considerado o maior problema dos alocadores na época (meados de 1995). Os autores estabeleceram uma base teórica e um conjunto de diretivas para que a fragmentação possa ser estudada de forma sistemática e assertiva. Por último, o trabalho faz uma revisão da literatura com trabalhos relacionados ao estudo dos alocadores de memória desde a década de 60. O trabalho em análise, no entanto, foca quase que exclusivamente no aspecto de uso de memória e pouco no tempo de resposta de um alocador. Além

disso, os autores desconsideraram em seu estudo aspectos importantes de um alocador como o uso da *cache* e aspectos de multiprogramação, porque eram pouco explorados até meados de 1995.

Em (Lea, 1996), Doug Lea apresenta o *DlMalloc*, alocador que serviu de base para o alocador *Ptmalloc2* (Gloger, 2006), este último o atual alocador padrão para todas as aplicações que usam a *glibc*. Neste trabalho, o autor apresenta as principais estruturas de dados e mecanismos utilizados até hoje no desenvolvimento dos alocadores. As técnicas mais relevantes apresentadas pelo autor são o uso dos *bins* e das *boundary tags*. Os *bins* são implementações de diretórios voltados para o mapeamento de objetos através do seu tamanho, e é usado na boa parte dos alocadores a serem investigados neste estudo. A *boundary tag*, por sua vez, é uma técnica que consiste em colocar os metadados de gerência de cada bloco de memória no cabeçalho e no rodapé do próprio bloco, ao invés de um cabeçalho central. Esta técnica tem como objetivo facilitar a operação de *Coalescing*, uma vez que examinar se um bloco adjacente está livre é feito apenas checando esses metadados no próprio bloco. Além disso, Lea apresenta o conceito de *Wilderness Preservation*, que basicamente consiste em alocar a maior parte das requisições em porções de memória que já tenham sido usadas alguma vez, evitando alocar em áreas novas, ao que ele chama de *wilderness*. Essa estratégia minimiza a fragmentação, fazendo com que o alocador só use a nova área quando não é mais possível alocar em outra porção de memória da *heap*.

Com o incremento cada vez maior da multiprogramação, houve a necessidade do desenvolvimento de alocadores que se comportassem melhor em ambientes com múltiplas threads. Até então, o suporte dado para múltiplas threads pelos alocadores sofria muito com a contenção da memória. Em (Berger et al., 2000), os autores apresentam o *Hoard*, um alocador com a proposta de ser mais rápido e mais escalável que seus antecessores, especialmente no que tange aos sistemas multiprocessados. O artigo apresenta problemas inerentes à multiprogramação (ver Seção 2.3) e quais as abordagens utilizadas pelo alocador para resolver ou minimizar estes desafios (ver Seção 3.4). O autor realiza uma avaliação experimental do *Hoard*, comparando-o aos alocadores *Ptmalloc2*, *Mtmalloc* e *Solaris* (Larson e Krishnan, 1998), com um conjunto de onze aplicações, sendo destas quatro aplicações *single-threaded* e sete aplicações *multithreaded*. A comparação foi feita em termos de tempo de execução, escalabilidade, minimização do *false-sharing* e fragmentação. Os resultados mostraram que, embora o *Hoard* não tenha sido o alocador com o menor tempo de execução (cerca de 8% mais lento do que a média geral), esse teve o melhor fator de escalabilidade entre os alocadores testados, em todas as aplicações *multithreaded*. Além disso, o *Hoard* conseguiu evitar completamente o *false-sharing* nos testes feitos com até 14 processadores. Com relação à fragmentação, os autores embora afirmem que a estratégia do *Hoard* tenha tido um bom resultado, não foi possível medir com precisão o índice de fragmentação dos outros alocadores, deixando esta questão comparativa em aberto.

Em (Chang et al., 2000) os autores apresentam o *CHL*, um alocador que utiliza informações de execução da aplicação para obter melhores resultados em termos de tempo de resposta. Os autores se baseiam na premissa de que a maior parte da memória utilizada por uma aplicação depende, primariamente, do estilo de programação dos desenvolvedores e de estruturas básicas que são pouco

dependentes da entrada do programa. Se apoiando nisso, os autores propõem que a aplicação rode uma vez utilizando o alocador padrão, com um programa que realize a caracterização da sua memória, armazenando informações como a quantidade de memória alocada, os tamanhos de memória mais requisitados, número de alocações simultâneas, etc. Com a caracterização prévia do uso de memória da aplicação, o alocador CHL adapta-se automaticamente para a aplicação. O CHL utiliza de estratégias como, alocar inicialmente a quantidade de memória que a aplicação utilizou previamente, reservar um número maior de blocos para alocações mais frequentes e cria um número de *heaps* com base no número de alocações simultâneas. Para avaliar o CHL, os autores realizaram um estudo experimental comparando-o com outros cinco alocadores na execução de seis aplicações. A análise foi feita apenas no tempo de execução das aplicações, e foi medida nas rotinas de alocação/desalocação através dos ciclos de máquina. Os resultados obtidos mostraram que o CHL obteve um desempenho cerca de quatro vezes mais rápido do que os outros alocadores. Embora o resultado tenha sido bom para as aplicações testadas, o trabalho presume que o uso de memória de uma aplicação seja pouco dependente de sua entrada, e que a primeira execução contemple um uso comum da aplicação, ambas as suposições podem não ser verdadeiras para diversas aplicações e cenários atuais, caracterizando uma solução não aplicável para todos os casos. Adicionalmente, ao focar apenas no aspecto de tempo de execução, os autores deixaram de lado inúmeros outros aspectos que foram considerados no desenvolvimento dos outros alocadores testados.

Em (Häggander et al., 2001), os autores propõem um método de pré-processamento chamado de Amplify, que analisa o código-fonte da aplicação para incrementar o seu gerenciamento de memória. O princípio deste trabalho está nas chamadas *pools* de objetos. Uma *pool* é uma estrutura de dados que armazena várias instâncias do mesmo tipo de objeto que podem ser reutilizados sem a necessidade da alocação/desalocação dinâmica de memória. Com isso a alocação e liberação de objetos na *pool* é feita muito mais rapidamente do que na alocação dinâmica de memória. Contudo, implementar essas *pools* no código-fonte de uma aplicação é uma tarefa dispendiosa, complexa e propensa a erros. O Amplify busca resolver esse problema realizando todo o processamento do código e gerando em tempo de compilação, as *pools* e suas estruturas de gerenciamento.

Para avaliar a eficiência do Amplify, este foi testado em comparação aos alocadores de memória Ptmalloc2 e Hoard. O experimento foi conduzido em duas fases, primeiro com aplicações simples com cargas sintéticas e na segunda etapa com uma aplicação real. Na etapa com as aplicações simples o Amplify se mostrou cerca de seis vezes mais rápido em comparação aos alocadores testados e com um melhor fator de escalabilidade. Com relação à aplicação real, essa abordagem apresentou um conjunto de problemas. Primeiramente, inúmeras bibliotecas da aplicação não tinham o código-fonte disponível, restringindo a aplicabilidade do Amplify. Os autores também alegaram que o código da aplicação em si, não possuía uma boa escalabilidade, o que fez com que o ganho da escalabilidade do Amplify fosse reduzido. Com essas limitações o Amplify não se mostrou uma solução viável de forma geral, como são os alocadores atuais.

Em (Matias et al., 2011) os alocadores Jemalloc, Hoard, TCMalloc, Nedmalloc e o Ptmalloc2 foram avaliados experimentalmente com a execução do gerenciador de banco de dados MySQL (Oracle Corporation, 2014). Para avaliar os alocadores

os autores criaram um experimento contendo a execução de um conjunto de operações de *select*, *insert*, *update* e *delete* durante dez minutos. As execuções do MySQL, com cada alocador, foram analisadas com relação ao tempo de execução, quantificado de acordo com o número total de transações, leituras, escritas e tempo médio da consultas feitas durante o período de teste. Os resultados mostraram que a utilização de diferentes alocadores influenciaram diretamente no desempenho do MySQL. Dentre os resultados, se destaca que na execução com o Jemalloc o MySQL obteve um aumento significativo da capacidade de transações por segundo (TPS), na ordem de 15 TPS em relação ao alocador padrão Ptmalloc2.

Em (Ferreira et al., 2011a) foi realizado um estudo teórico aprofundado dos mesmos alocadores de memória a serem apresentados no Capítulo 4. O trabalho apresentou de forma geral, as principais estruturas de dados dos alocadores e aspectos gerais de seus algoritmos. Cada alocador teve a sua complexidade assintótica avaliada com relação ao tempo de resposta de uma alocação e o espaço usado pelo alocador com os seus metadados (*overhead*). O trabalho conclui que ambas as versões do Ptmalloc e o Jemalloc possuem tempo constante de alocação, e no pior caso a complexidade da alocação é linear, $O(R)$, onde R é o número de alocações. Hoard, Miser e TCMalloc possuem tempo de alocação linear para todos os cenários. Portanto, do ponto de vista teórico, os alocadores Ptmalloc2, Ptmalloc3 e Jemalloc são melhores que os outros três. Com relação ao *overhead*, os alocadores foram ranqueados da seguinte maneira: Ptmalloc3 (menor *overhead*), seguido por Ptmalloc2, Jemalloc, TCMalloc, Miser e Hoard. Avaliando os alocadores de forma teórica, foi possível concluir que os alocadores Ptmalloc3, Ptmalloc2 e Jemalloc foram os melhores alocadores tanto em tempo de execução quanto em relação ao *overhead*.

Em (Ferreira et al., 2011b), os alocadores Hoard, Ptmalloc2, Ptmalloc3, TCMalloc, Jemalloc, TLSF e Miser foram submetidos a um teste com três aplicações distintas de *middleware*. A avaliação foi feita executando o *middleware* com cada alocador, observando o seu tempo de execução e uso de memória. Além dos alocadores, o número de processadores também foi variado, a fim de se avaliar o desempenho da aplicação e dos alocadores com relação à escalabilidade. Os resultados mostraram que o TCMalloc obteve o melhor resultado em todos os critérios, seguido pelo Ptmalloc3. Além da avaliação dos alocadores, o trabalho fez uma caracterização da memória dos três *middlewares*, com relação à sua distribuição dos tamanhos requisitados. Os resultados mostraram que a maior parte das alocações foi feita com tamanhos até 64 bytes, caracterizando os *middlewares* como um tipo de aplicação com predominância de alocações pequenas.

2.5 Considerações Finais

Os alocadores de memória são objetos de estudo desde a década de 60, quando as primeiras políticas de alocação começaram a ser publicadas no meio acadêmico, iniciando com (Collins, 1961) apresentando as políticas *best-fit*, *worst-fit*, *first-fit* e *random-fit*. De lá para cá, inúmeros outros aspectos vêm sendo estudados com relação ao estudo do gerenciamento de memória e novos algoritmos vêm sendo propostos para solucionar de forma otimizada os desafios oriundos dos ambientes *multithreaded* e multiprocessado.

Dessa forma, há atualmente inúmeros alocadores com diferentes propostas e técnicas de gerenciamento de memória. Como o desempenho de operações de alocação de memória tem significativa influência no desempenho global da maioria das aplicações computacionais, a seleção de um alocador de memória é um importante requisito no projeto de sistemas mais sofisticados, sendo muitas vezes negligenciada pelos projetistas de software. A forte correlação entre o perfil de uso dinâmico da memória com o desempenho do alocador exige que a seleção do alocador ocorra por meio de um estudo experimental, este estudo pode ser custoso e complexo de ser avaliado.

Os trabalhos de avaliação experimental dos alocadores de memória encontrados na literatura possuem uma de duas limitações. Nos trabalhos onde a aplicação utilizada é uma aplicação de carga real, a avaliação dos alocadores está sujeita ao seu perfil de alocação específico, que por muitas vezes não é apresentado pelos autores, ou pode não pode ser generalizado para outros cenários. Nos trabalhos onde a aplicação utilizada é uma aplicação com carga sintética, a avaliação dos alocadores está sujeito ao perfil da carga estabelecida pelos autores, que pode não ser relevante para todos os cenários.

Nenhum dos trabalhos buscou realizar uma avaliação ampla e sistemática envolvendo caracterização de aplicações reais para a criação da aplicação teste. Adicionalmente, nenhum trabalho realizou um experimento com a variabilidade dos principais fatores que afetam o desempenho do alocador. Objetivando suprir essa necessidade, foi estabelecido um estudo exploratório de avaliação dos alocadores, criando um experimento que através da caracterização de aplicações reais, gere cargas sintéticas realistas para a avaliação dos alocadores. Este trabalho visa compreender quais as vantagens e desvantagens de cada abordagem estabelecida por cada alocador, em um grupo significativo de cenários de execução.

3. ALOCADORES INVESTIGADOS

3.1 Introdução

Nesta dissertação de mestrado foram investigados seis alocadores de memória. Os alocadores foram escolhidos por serem amplamente utilizados e conhecidos. Nesta seção serão apresentadas as estratégias de gerenciamento de memória adotadas por cada alocador e suas principais estruturas de dados. Embora todos os alocadores possuam as estruturas de dados básicas: *heap*, diretório e bloco de memória, cada alocador as renomeiam de acordo com sua estratégia de alocação. Nesta dissertação, serão relacionadas as estruturas básicas com a nomenclatura adotada pelos alocadores, para facilitar a correspondência na análise da documentação oficial de cada alocador. Para este capítulo, foi realizada uma pesquisa na documentação oficial e no código-fonte de cada alocador, bem como os trabalhos (Ferreira et al., 2011b) e (Ferreira, 2012) que apresentam a estruturas de dados básicas dos alocadores.

3.2 Ptmalloc2

O Ptmalloc2 (Gloger, 2006) é o alocador que gerencia as operações de alocação dinâmica de memória da GNU C Library (*glibc*). Portanto, a memória alocada e desalocada dinamicamente, nos programas que executam em distribuições Linux que utilizam a *glibc* como biblioteca C padrão, é gerenciada pelo Ptmalloc2. O Ptmalloc2 é um alocador de propósito geral, seu desenvolvedor, Wolfram Gloger, o programou baseado no código de Doug Lea (Lea, 1996), acrescentando adaptações para lidar com múltiplas *threads*, mas mantendo a maioria das estruturas de dados do seu alocador de memória antecessor, Dlmalloc. A Figura 3.1 apresenta as estruturas de dados do Ptmalloc2.

3.2.1 Arena

No Ptmalloc2, a *heap* é implementada pelas *arenas*. O conceito de *arenas* foi introduzido neste alocador para lidar com a contenção de memória das aplicações *multithreaded*. No carregamento do código de inicialização do alocador, o Ptmalloc2 requisita para o sistema operacional a primeira área de memória da aplicação chamada de *main arena*. Quando uma *thread* tenta alocar a memória e as *arenas* criadas estão bloqueadas pelas demais *threads* da aplicação, uma nova *arena* é criada, evitando assim que a *thread* fique esperando. Essa estratégia, portanto, minimiza a contenção das *threads* no acesso as *arenas*.

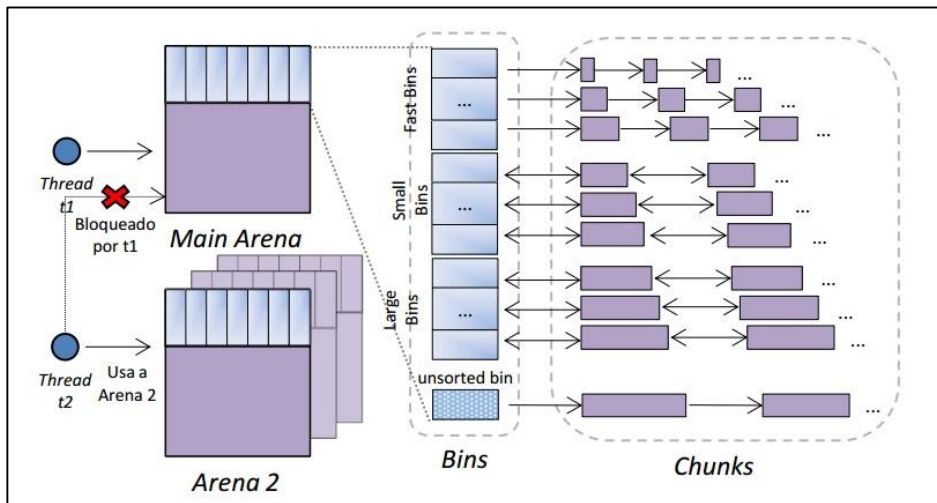


Figura 3.1. Estruturas de dados do alocador Ptmalloc2.

Apesar de não haver contenção na alocação, este problema pode ocorrer na liberação da memória alocada. Se uma *thread* adquire o *lock* de uma determinada *arena* e, logo em seguida, outra *thread* tenta desalocar um bloco de memória que pertence a *arena* bloqueada, a segunda *thread* terá que esperar até a que operação de alocação seja concluída e a *arena* liberada.

As *arenas* também introduzem um novo problema no gerenciamento de memória do Ptmalloc2, o *memory-blowup*. Não existe a possibilidade de mover a memória de uma *arena* para a outra, portanto, é possível que o alocador requisiite mais memória para o sistema operacional se a *arena* com espaço disponível para atender uma requisição estiver bloqueada por outra *thread*.

3.2.2 Chunk

O *Chunk* é a implementação do bloco de memória do Ptmalloc2 e representa a unidade de alocação do alocador. Quando uma aplicação requisita a memória para o Ptmalloc2 este procura em sua *arena* por um *chunk* livre que atenda à requisição da aplicação. Cada *chunk* é composto por um cabeçalho, que contém informações sobre o bloco de memória em questão e a área alocável que pode ser de fato usada pela aplicação.

Cada *chunk* tem um tamanho múltiplo de oito bytes. Por esse motivo, quando o alocador recebe uma requisição de memória, este arredonda o tamanho requisitado para o múltiplo acima. Por causa deste arredondamento, o alocador sofre com a fragmentação interna, que pode ser de até sete bytes por alocação (casos onde o tamanho requisitado é um múltiplo de oito mais um byte).

O Ptmalloc2 mantém um *chunk* especial para delimitar a área de memória em uma *heap* que ainda não foi utilizada (não foi alocada nem está na lista de livres), este é chamado de *top chunk* ou *wilderness chunk*. Quando ocorre uma requisição de memória que não pode ser atendida pelos *chunks* presentes nas listas de *chunks* livres, o Ptmalloc2 usa a memória disponível no *top chunk*. Caso a requisição não possa também ser atendida pela memória disponível no *top chunk*, o tamanho da

arena é incrementada com uma chamada para o SO (ex. *sbrk* no Linux). O *top chunk* é o único *chunk* que pode ter seu tamanho incrementado.

3.2.3 Bin

Os *bins* são as implementações dos diretórios no Ptmalloc2, ou seja, são as estruturas de dados utilizadas para referenciar os *chunks* livres. Um *bin* é uma lista encadeada de *chunks* livres de uma determinada classe de tamanho. Quando ocorre uma desalocação de um *chunk*, o seu tamanho é usado para calcular em qual *bin* ele será encaixado. Esta estrutura de *bins* permite que o Ptmalloc2 utilize em primeiro nível o *best-fit*, como política de alocação, uma vez que o melhor tamanho é priorizado na busca de *chunks* livres. Os *bins* são divididos em duas categorias: *Fast Bins* e *Normal Bins* de forma que o alocador tenha o comportamento adequado para agilizar alocações de acordo com o tamanho requisitado.

Os *Fast Bins* foram criados para permitir acesso rápido aos *chunks* de memória e agrupam as listas de *chunks* de tamanhos menores que 64 bytes. É o único tipo de *bin* que é formado por encadeamento simples, para economizar memória. Além disso, o acesso à lista segue a política de pilha (LIFO), de forma a tornar mais provável que o primeiro *chunk* na lista já esteja em *cache*, ganhando em desempenho cada vez que isto ocorre. O Ptmalloc2 não realiza o *Coalescing* dos *chunks* presentes nestes *bins*. Esta estratégia faz com que os blocos de até 64 bytes não tenham que realizar o processo de junção de blocos livres, ficando imediatamente prontos para o uso, uma vez que este tipo de alocação é considerado muito frequente (Costa e Matias, 2014).

Os *Normal Bins* referenciam *chunks* livres de tamanhos entre 64 bytes e 128 KB. Estes *bins* são listas duplamente encadeadas e são subdivididos em outros dois tipos de *bins*: *Small bins*, entre 64 e 512 bytes, e *Large Bins*, entre 512 bytes e 128 KB.

Nos *Small Bins*, cada *bin* recebe sempre *chunks* de mesmo tamanho. Como cada lista tem o tamanho exato a ser alocado, não há a necessidade de ordenação. Nestes *bins*, toda vez que um *chunk* é liberado, é feita a tentativa de *Coalescing* com os seus adjacentes. Os *Large Bins*, por sua vez, agrupam uma classe de tamanho e, portanto, precisam de ordenação para que a busca seja feita rapidamente. Os *chunks* em cada *bin* são mantidos em ordem crescente de forma a facilitar a política de escolha *best fit*.

O Ptmalloc2 mantém ainda uma lista, chamada de *unsorted bin*, onde os *chunks* recém liberados e *chunks* resultantes do *Splitting* são colocados até que ocorra um *malloc*. Esta é a primeira lista consultada quando ocorre um *malloc* para tamanho maior que 512 bytes. Como o *chunk* no topo desta lista foi o último desalocado, provavelmente ainda está em *cache*. Essa estratégia também evita que seja necessário, por exemplo, reinserir um *chunk* no seu *Large Bin*, evitando inserção ordenada, uma operação de maior custo computacional do que inserir no topo dos *unsorted bins*. É interessante notar que, embora na maior parte das vezes o Ptmalloc2 utiliza o *best-fit* como política de alocação, para alocações maiores do que 512 bytes, ele inicialmente escolhe por privilegiar o desempenho, recuperando um bloco que está em *cache*, ao invés do melhor tamanho.

Tamanhos maiores que 128 KB não são gerenciados por este alocador. Qualquer requisição acima desse tamanho é repassada para o sistema operacional usando as chamadas de sistema *mmap/munmap* (Linux, 2013). Estas duas chamadas de sistema causam uma queda de desempenho, pois envolvem interação direta com o sistema operacional.

3.3 Ptmalloc3

A terceira versão do alocador de Wolfram Gloger (Gloger, 2006), baseada em uma versão mais recente do alocador de Doug Lea (Dlmalloc 2.8.3), trouxe várias modificações em relação ao Ptmalloc2, especialmente nas estruturas de dados que referenciam os blocos livres. Dentre estas modificações destaca-se a forma como são tratadas requisições classificadas como *large*. Para tornar mais eficiente a alocação e a desalocação de memória, os blocos livres de uma mesma classe são agrupados em árvores binárias e não mais em listas ordenadas.

Em relação ao uso das *arenas*, o Ptmalloc3 tem o mesmo comportamento do Ptmalloc2. Na medida em que as *arenas* existentes vão sendo bloqueadas pelas *threads* em execução, o Ptmalloc3 cria uma nova *arena* para evitar que a *thread* atual seja bloqueada. A Figura 3.2 apresenta as principais estruturas de dados do Ptmalloc3.

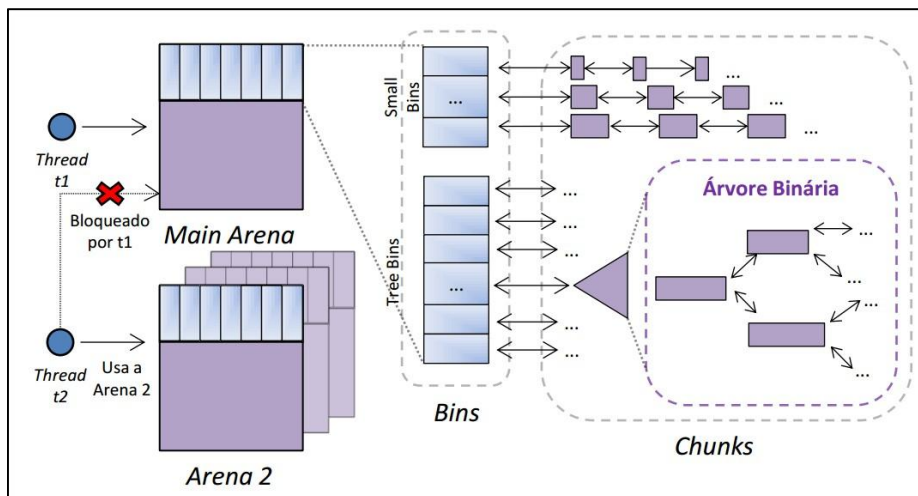


Figura 3.2. Estruturas de dados do alocador Ptmalloc3.

3.3.1 Chunk

A unidade de alocação do Ptmalloc3 é o *chunk*. A estrutura de um *chunk* alocado é exatamente a mesma do Ptmalloc2. Esta possui um cabeçalho com o tamanho do bloco atual, o tamanho do bloco anterior e a área útil que pode ser usada pela aplicação. O *top chunk* também não sofreu alterações, corresponde à memória disponível ainda não alocada no topo da *arena*.

As mudanças ocorrem nos *chunks* livres, que agora possuem dois tipos diferentes de estruturas dependendo do tamanho da região de memória alocável. A

primeira estrutura é chamada de *small chunks* e funciona como um nó de uma lista encadeada, como no Ptmalloc2; os *large chunks* funcionam como um nó de uma árvore binária. Os *large chunks* possuem, além dos atributos de um *small chunk*, ponteiros que apontam para os nós filhos na árvore binária, um ponteiro para o nó pai e um índice que indica a qual árvore este nó pertence.

3.3.2 Bins

Assim como o seu antecessor, o Ptmalloc3 privilegia o *best-fit*, como a sua política de alocação. Para isso ele usa duas estruturas de diretórios para organizar dois tipos distintos de alocações. A implementação do diretório no Ptmalloc3 é feita pelos *bins*, que são divididos em: *small bins* e *tree bins*. *Small bins* são listas duplamente encadeadas de *chunks* que abrigam tamanhos entre 8 e 256 bytes, espaçados de 8 em 8 bytes. Cada *bin* recebe apenas um único tamanho, assim como os *small bins* do Ptmalloc2, garantindo assim acesso imediato ao bloco de memória a ser retornado, usando apenas um cálculo de índice.

Os tamanhos de 256 bytes a 256 KB são mantidos nos *tree bins*, que são representados como uma árvore binária que abriga apenas uma classe de tamanho. Diferente dos tamanhos gerenciados pelo *small-bin*, a classe de tamanho relacionada com cada *bin* nessa estrutura é definida por um intervalo de potência de dois. A classe de tamanho, a cada nível da árvore, é dividida na metade de forma que o nó atual tenha o tamanho médio, a subárvore esquerda contenha apenas nós de tamanho menor que o do atual e a subárvore direita tenha apenas os nós de tamanho maior.

A organização dos *large chunks* em árvores binárias evita que seja necessário percorrer todos os *chunks* da classe de tamanho para encontrar o que melhor atende à solicitação de uma aplicação. Os *chunks* de mesmo tamanho nas árvores são organizados em uma lista encadeada circular, onde o topo da lista é o *chunk* mais antigo. Somente o *chunk* do topo da lista está de fato ligado à árvore, como um nó dela. Assim haverá somente um *chunk* de cada tamanho dentro da árvore.

Diferente do Ptmalloc2, o Ptmalloc3 não utiliza o *unsorted bin*, portanto, para todas as alocações o alocador decidiu por utilizar a política *best-fit*. As requisições acima de 256 KB são repassadas para o sistema operacional via *mmap* e as desalocações via *munmap*. Estas são alocações que devem ser evitadas, pois cada uma irá causar trocas de modo de operação (de *user level* para *kernel level* e *kernel level* de volta para *user level*) e, conseqüentemente, atrasos na aplicação que estiver usando o Ptmalloc3.

3.4 Hoard

O alocador Hoard (Berger et al., 2000) foi projetado para ser rápido e eficiente no gerenciamento de memória em ambientes multiprocessados. A forma como as estruturas e operações deste alocador foram implementadas, especialmente no uso de suas múltiplas *heaps*, trouxe a este alocador a característica pelo qual é conhecido: o tratamento dos efeitos causados pela contenção da *heap*, *false sharing* e a minimização da fragmentação de memória, incluindo o memory blowup. Diferente dos alocadores já apresentados, o Hoard

também introduz um outro nível na organização da memória. Além da *heap*, diretório e bloco de memória, o Hoard cria uma estrutura que agrupa blocos de memória de mesmo tamanho, chamado de Superbloco. A Figura 3.3 apresenta as estruturas de dados do Hoard.

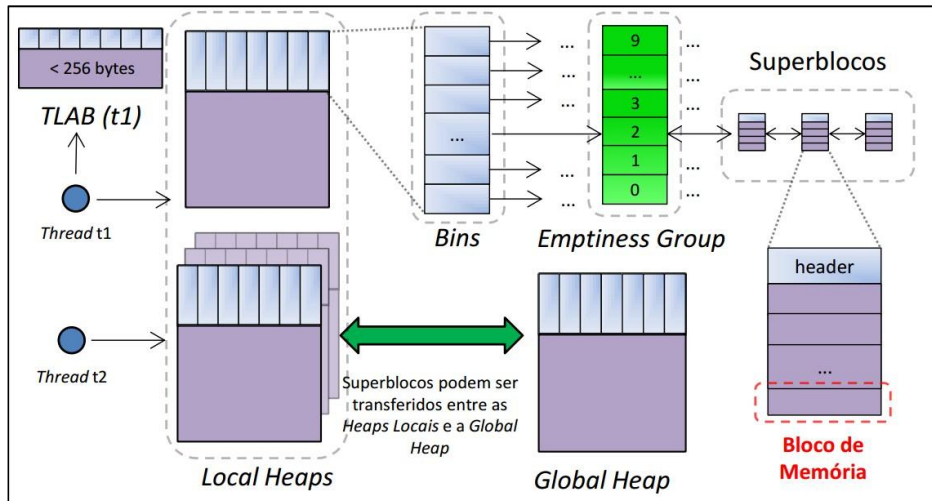


Figura 3.3. Estruturas de dados do alocador Hoard.

3.4.1 Superbloco

O Hoard gerencia a memória com uma estrutura de dados chamada de superbloco. Cada superbloco tem, por padrão, 64 KB e abriga blocos de memória de apenas um tamanho. Isso significa que alocações de diferentes tamanhos, quando não são arredondadas, são alocadas em superblocos distintos. O superbloco é formado por um cabeçalho que contém seus metadados e por um vetor de blocos de memória.

Os cabeçalhos de um superbloco contêm campos que indicam a qual *heap* este superbloco pertence, o tamanho dos objetos no vetor de blocos, informações da gerência interna do superbloco como o número de blocos livres e ponteiros que apontam para o superbloco anterior e para o próximo na lista encadeada em que está inserido.

Todos os blocos de memória de um superbloco ficam indexados em seu vetor e possuem um mesmo tamanho. No entanto, um superbloco abriga alocações de uma classe de tamanho definida pela estrutura de diretórios do Hoard. Qualquer alocação que não se encaixe (isto é, que não possua um superbloco para o seu tamanho) é arredondada para o tamanho acima mais próximo. Arredondar o tamanho requisitado simplifica a gerência da memória, mas deixa o alocador sujeito à fragmentação interna.

Um superbloco não pode ser usado por mais de uma *thread* ao mesmo tempo. Quando várias *threads* estão alocando o mesmo tamanho de memória simultaneamente, cada uma receberá memória de um superbloco diferente e este ficará em posse da *thread*, em uma área destinada apenas ao seu acesso. Como o

superbloco fica em posse da *thread*, o *false sharing* ocorre apenas em raras situações, quando um superbloco é movido para uma área compartilhada.

O maior bloco que o Hoard gerencia é normalmente metade do tamanho de um superbloco, por padrão 32 KB. Quando a aplicação requisita um tamanho maior que este, o bloco que ela recebe é alocado diretamente do sistema operacional, com a chamada de sistema *mmap*, e quando ocorre uma desalocação ele é devolvido para o sistema operacional com *munmap*. Nestes casos o Hoard começa a sofrer com o aumento do tempo de resposta das operações de alocação e desalocação de memória.

3.4.2 Bins

Os *bins* do Hoard são listas duplamente encadeadas de superblocos. Cada *bin* abriga apenas superblocos cujos blocos são da mesma classe de tamanho. Dentro de cada *bin* os superblocos são divididos em listas por taxa de preenchimento (número de blocos alocados), que são chamadas de *emptiness groups* (*EG*). À medida que ocorrem alocações (ou desalocações) de blocos, os superblocos são movidos de um *emptiness group* para outro. O índice do *emptiness group* indica a porcentagem de uso do superbloco, de modo que $EG_{[0]}$ aponta para superblocos vazios, enquanto $EG_{[9]}$ referencia um superbloco cheio. Quando ocorre uma alocação, Hoard dá preferência por retornar memória do superbloco mais cheio possível. Isso é feito para tentar garantir que os superblocos, ou fiquem completamente utilizados ou sejam esvaziados aos poucos pelas desalocações.

3.4.3 Implementação da *heap*

Para evitar a contenção de memória, o Hoard implementa três estruturas de dados: *Thread Local Allocation Buffer* (TLAB), *heap* local e a *heap* global. O TLAB é um buffer de 256 KB, exclusivo de cada *thread* da aplicação, para onde são direcionados requisições de até 256 bytes. Como essa estrutura é de acesso exclusivo da *thread*, não há a necessidade do uso de *locks* nas operações de alocação e desalocação, tornando as operações alocação/desalocação mais rápidas. Como em geral espera-se que as requisições de memória da maioria das aplicações sejam menores que 256 bytes, o alocador assume que a maioria das alocações sejam direcionadas para estes buffers.

A *heap* local é uma área semi-compartilhada entre as *threads*, que abriga superblocos que contenham blocos acima de 256 bytes. Cada *thread* é associada, por meio de uma função *hash*, a apenas uma das *heaps* locais que o Hoard mantém. No entanto, dependendo do número de *threads* que uma aplicação lança, mais de uma *thread* pode estar associada à mesma *heap* ao mesmo tempo. Ao todo são mantidas duas *heaps* locais por processador. Acredita-se que este número de *heaps* seja suficiente para minimizar a contenção no acesso uma vez que em um sistema multiprocessado, só executarão de fato ao mesmo tempo, um número de *threads* igual ao número de processadores.

A *heap* global funciona como um repositório onde são armazenados os superblocos movidos das *heaps* por processador que foram ficando vazios e de onde qualquer *thread* em execução pode buscar superblocos, quando não é

possível alocar da *heap* a qual está associada. O reaproveitamento dos superblocos proporcionado pelo uso da *heap* global ajuda a evitar que a aplicação busque mais memória do sistema operacional, o que além de aumentar o tempo de resposta das operações de alocação, contribuiria para o aparecimento do *memory blowup*.

O acesso à *heap* global pode causar contenção entre as threads. Cada vez que há retirada ou inclusão de um superbloco na *heap* global, esta *heap* é bloqueada pela primeira *thread* que fez a requisição. Neste caso, as outras *threads* são obrigadas a aguardar a liberação do *lock* para prosseguir sua execução.

3.5 Miser

Miser (Tannenbaum, 2010) é o alocador distribuído pela *Cilk Arts* como parte da linguagem programação *Cilk++*, desenvolvida para obter alto desempenho com aplicações *multithreaded*. Este alocador é baseado no código do alocador Hoard e visa proporcionar melhor desempenho às aplicações escritas com *Cilk++*. A Figura 3.4 apresenta as principais estruturas de dados do Miser.

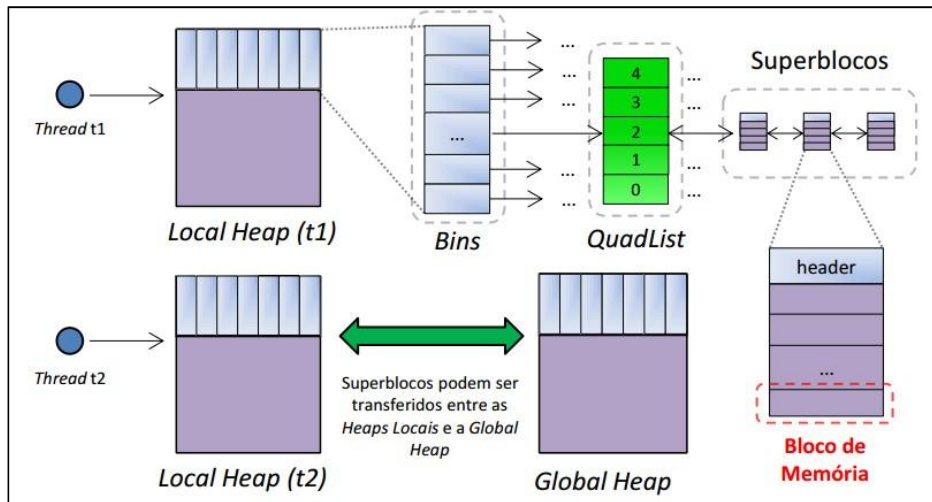


Figura 3.4. Estruturas de dados do alocador Miser.

3.5.1 Estruturas de Dados

Baseado no Hoard, o Miser também gerencia a memória usando superblocos, compostos pelo seu cabeçalho e por um vetor de blocos de mesmo tamanho. No entanto, as classes de tamanho dos superblocos abrigam apenas blocos de 8 a 256 bytes, como nos TLABs do Hoard. O tamanho das classes de tamanho que o Miser trabalha é definido em potências de dois. Isso significa que qualquer requisição cujo tamanho não é uma potência de dois, terá seu tamanho arredondado. Essa estratégia, embora simplifique o cálculo do tamanho do bloco destinado à alocação, gera fragmentação interna.

Os *bins* do Miser são listas encadeadas de superblocos por classe de tamanho, ordenadas por taxa de preenchimento através do *QuadList*. Miser coloca superblocos na *QuadList* de acordo com a quantidade de objetos alocados. Quando

um superbloco não possui memória disponível, ele é movido para o *QuadList*_[4], enquanto um superbloco praticamente vazio é mantido em *QuadList*_[0]. Superblocos completamente vazios são mantidos à parte, em uma lista duplamente encadeada chamada de *Empty List*. O Miser privilegia alocações dos superblocos mais cheios, estratégia semelhantemente usada pelo Hoard, para tentar otimizar o uso de memória, visando sempre encher um superbloco antes de utilizar o próximo.

O Miser cria uma *heap* local exclusiva por *thread*, o que impede a ocorrência de contenção nas alocações e desalocações direcionadas para estas estruturas. A implementação das *heaps*, além da contenção, previne a ocorrência de *false sharing*, porque um superbloco só pode pertencer a uma única *heap* ao mesmo tempo, evitando que mais de uma *thread* possa usá-la. Quanto ao número de *heaps*, o alocador tem uma limitação de 1024 *heaps* e a associação das *threads* com as *heaps* se dá por ordem de criação.

Este alocador também implementa uma *heap* global, que funciona como um repositório de superblocos que não estão em uso para tornar possível seu reaproveitamento, evitando o *memory-blowup*. No entanto, assim como no Hoard esse mecanismo gera a possibilidade de que a contenção e o *false-sharing* ocorram na *heap global*, uma vez que esta estrutura é compartilhada entre todas as *threads*.

Quando ocorre uma requisição acima de 256 bytes, esta é repassada para o alocador padrão do sistema operacional. Segundo o autor do Miser, não há necessidade de tratamento para tais tamanhos, pois em média, 98% das alocações feitas em aplicações de propósito geral serão de até 256 bytes, ou seja, praticamente todas as requisições de memória serão atendidas pelo Miser.

3.6 Jemalloc

Jemalloc foi desenvolvido por Jason Evans para ser o alocador padrão do *FreeBSD* (Evans, 2006) em substituição ao *PHKmalloc* (Kamp, 1998), que não tem bom desempenho em sistemas multiprocessados e *multithreading*. Também é parte do *NetBSD* (The NetBSD Foundation, 2014), das versões atuais do navegador Mozilla Firefox (Mozilla Foundation, 2014) e em vários componentes do Facebook. O objetivo desta implementação é manter bom desempenho com aplicações *multithreaded* em geral, evitando o aparecimento dos efeitos da fragmentação de memória e contenção de *heap*. A Figura 3.5 apresenta as principais estruturas de dados do alocador Jemalloc.

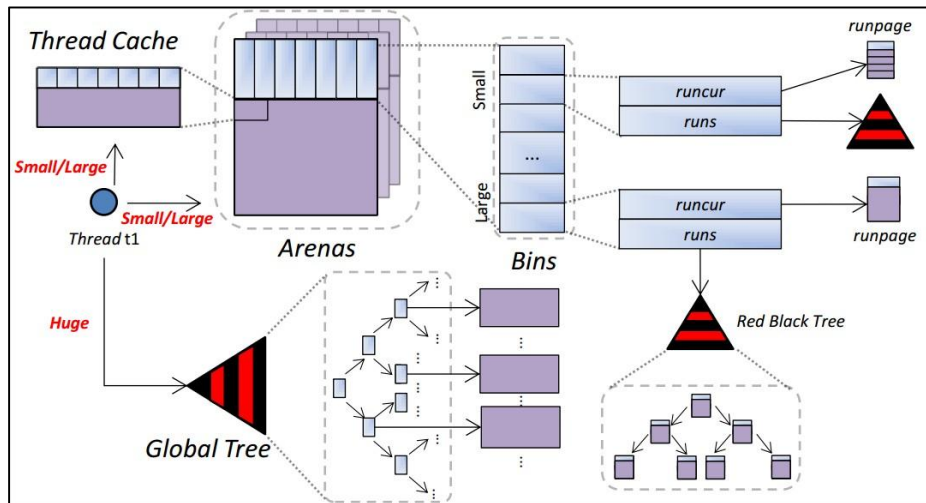


Figura 3.5. Estruturas de dados do alocador Jemalloc.

3.6.1 Estruturas de Dados

A estratégia do Jemalloc categoriza a alocação de memória em três principais classes de tamanho: *small* (alocações de 4 bytes até 4 KB), *large* (alocações de 4 KB até 4 MB) e *huge* (alocações maiores do que 4MB). Para atender as alocações classificadas como *small* e *large*, o Jemalloc utiliza estruturas de dados chamadas de *runpages*, ou simplesmente *run*. O tamanho mínimo do *run* é de uma página (normalmente 4 KB), portanto, para alocações na classe *small*, por consequência menor do que uma página, os *runs* são separados em objetos de igual tamanho e mapeados por um cabeçalho incluído no início do *run*. Para a classe *large*, cada objeto é armazenado em um *run* dedicado.

Assim como os alocadores apresentados até aqui, o Jemalloc organiza os seus *runs* em *bins*. Cada *bin* referencia *runs* que contenham blocos de um determinado tamanho. Dentro dos *bins*, os *runs* são mantidos, ordenados por endereço, em árvores *red-black*. O motivo desta ordenação é minimizar o número de *runs* que não foram completamente usados. Quando um *run* tem que ser removido de uma árvore, escolhe-se sempre o de menor endereço, tentando ao máximo manter a memória não utilizada agrupada no final da *arena*, o que minimiza os efeitos da fragmentação.

Cada *bin* mantém também um ponteiro para o *run* atualmente em uso, o *runcur*. Isso evita que seja necessário inserir e remover *runs* das árvores, ao acessar o *run* que está em uso, melhorando o tempo de resposta do algoritmo. Também facilita na busca pela região de memória a ser retornada quando a requisição se encaixa na classe *small*, uma vez que o *run* atual pode ter blocos de memórias livres.

As alocações da classe *huge*, por sua vez, são direcionadas para blocos de memória dedicados na *arena* e seus metadados são armazenados em uma única árvore *red-black* global. Como não são requisições muito comuns, espera-se que a contenção nesta estrutura seja mínima.

3.6.2 Implementação da *heap*

A *heap* do Jemalloc é implementada pelo conceito de *arenas*, de forma similar ao apresentado nos alocadores Ptmalloc2 e Ptmalloc3. Em sistemas multiprocessados, o Jemalloc cria, por padrão, quatro *arenas* por processador. Como algumas aplicações podem lançar mais *threads* do que isso, as *threads* são associadas às *arenas* usando o algoritmo de escalonamento *round-robin*, quando a *thread* faz sua primeira alocação de um objeto da classe *large* ou *small*. Este algoritmo permite balancear de forma mais justa o número de *threads* associadas a cada *arena*.

Para minimizar os efeitos da contenção de *heap*, o Jemalloc mantém também em cada *arena* um vetor de *thread-caches*, que vão sendo atribuídas às *threads* na medida em que são criadas. Estas *caches* são de uso exclusivo da *thread* e mantêm objetos de tamanho até 32 KB, compreendendo assim objetos da classe *small* e alguns objetos da classe *large*. Por ser uma estrutura exclusiva da *thread*, não há contenção de memória, logo não há necessidade de mecanismos de sincronização, tornando a operação de alocação/desalocação mais rápida.

Para evitar o *memory-blowup* causado pelo *thread-cache*, o Jemalloc implementa um *garbage collector* (GC) incremental, que utiliza as alocações como unidades de tempo e avalia o tempo em que o objeto fica alocado na *thread-cache*. Se o objeto ficar por muito tempo alocado, o GC o volta para a sua *arena* de origem, utilizando um algoritmo de decaimento exponencial. O objetivo dessa estratégia é o de manter na *thread-cache* apenas objetos que serão rapidamente alocados/desalocados.

3.7 Thread-Cache Malloc (TCMalloc)

O TCMalloc (Ghemawat e Menage, 2014) é distribuído como parte do *Google Performance Tools*, um conjunto de ferramentas que permite criar e analisar a performance de aplicações robustas, especialmente aplicações multithreaded. A implementação deste alocador tem como principal objetivo tornar mais rápidas as operações de alocação, reduzindo a contenção de *heap* com a implementação de *heaps* por *thread* (*thread-caches*). As principais estruturas de dados podem ser observadas na Figura 3.6.

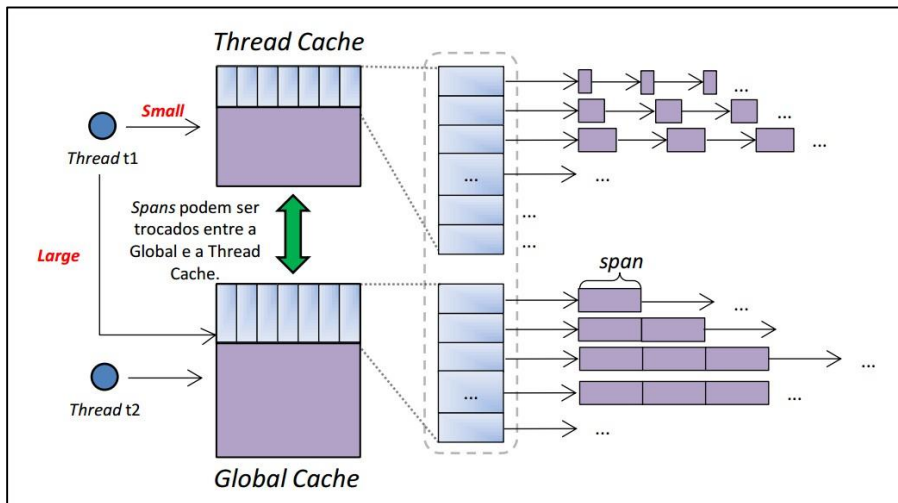


Figura 3.6. Estruturas de dados do alocador TCMalloc.

3.7.1 Estruturas de Dados

As alocações no TCMalloc são divididas em duas categorias: *small* (para alocações até 32 KB) e *large* (para alocações maiores do que 32 KB). As alocações classificadas como *small* são divididas em classes de tamanho, espaçadas de oito em oito bytes para tamanhos menores que um kilobyte, e espaçadas de 128 em 128 bytes para tamanhos de um kilobyte a 32 KB. A diferença dos alinhamentos é adotada para reduzir a fragmentação interna para tamanhos até um kilobyte, e para simplificar a gerência de tamanhos acima desse limiar. Os objetos pertencentes à mesma classe de tamanho são mantidos em listas encadeadas simples.

A inserção e remoção de objetos ocorre sempre no topo da lista. Esta estrutura permite que as operações de alocação sejam mais rápidas, uma vez que não há necessidade de busca dentro da lista e o número de ponteiros atualizados é o mínimo possível. Também, existe um ganho no fato de que o bloco alocado foi o último liberado e ainda pode estar na *cache* do processador.

Alocações classificadas com *large* são sempre arredondadas para múltiplos do tamanho de página (4 KB). Para requisições de tamanho entre uma e 255 páginas, cada tamanho é direcionado para uma lista encadeada de objetos específica. Objetos que contém 256 ou mais páginas são armazenados na mesma lista. Por se tratar de objetos não muito frequentemente alocados, o tamanho desta lista dificilmente será grande o suficiente para causar um atraso nas operações de alocação, caso sejam enquadradas nesta classe. Estas listas são parte da *heap* central mantido pelo TCMalloc.

3.7.2 Implementação da *heap*

A *heap* mantida pelo TCMalloc consiste em um conjunto de páginas contíguas representadas por objetos chamados de *spans*. Os *spans* livres são armazenados nas listas de objetos da classe *large* e podem ser usados para satisfazer requisições maiores que 32 KB, ou podem ser quebrados em objetos das classes *small*. Para

cada *thread* criada, o TCMalloc associa uma *thread-cache* que corresponde basicamente a um vetor onde cada entrada aponta para uma das listas de objetos das classes *small*. Como a *thread-cache* é específica de uma única *thread*, não há necessidade de *lock* e não haverá contenção, o que torna a operação de alocação mais rápida.

Visando minimizar o *memory blowup* causado pelas *thread-caches*, o TCMalloc implementa uma *heap* central. Assim como o Jemalloc, o TCMalloc utiliza um GC para reduzir o desperdício gasto pela memória da *thread*. Quando uma *thread cache* ultrapassa o limite de memória estipulado pelo alocador, o alocador utiliza o GC para mover blocos livres para a *Heap Global*. A *heap* central pode ser acessada por qualquer *thread* e é o único ponto onde há risco de haver contenção é no seu acesso, seja para alocar um objeto *large* ou para buscar um *span* para alguma classe *small*. No entanto espera-se que a maioria das alocações seja de tamanhos menores que KB e, portanto possam ser atendidas pela *thread-cache* sem correr este risco.

4. CARACTERIZAÇÃO DA ALOCAÇÃO DINÂMICA DE MEMÓRIA EM APLICAÇÕES

4.1 Introdução

Assim como apresentado na Seção 2.4, existem vários estudos experimentais que comparam alocadores de memória. Trabalhos como (Ferreira et al., 2011b) e (Costa et al., 2013) apresentaram um estudo empírico comparativo dos alocadores de memória utilizados nesta dissertação, contudo, em ambos os casos seus resultados possuem certa limitação de aplicabilidade. Em (Ferreira et al., 2011b), os resultados são aplicáveis apenas para aplicações com perfis de uso de memória similares ao da aplicação usada no estudo, cujas principais características foram a predominância de alocações de tamanho menor que 64 bytes e com o maior número de alocações ocorrendo no início da execução do programa. Já em (Costa et al., 2013), embora a abordagem utilizando uma carga sintética permita uma maior flexibilidade para testar os alocadores em diferentes condições experimentais, a definição da carga de trabalho foi principalmente baseada na experiência dos autores, pois não foram encontrados trabalhos na literatura que subsidiassem esta definição por meio de estudos de caracterização de alocações dinâmicas de memória em diferentes tipos de aplicações reais.

Objetivando suprir esta necessidade, por meio da identificação de diferentes padrões de alocação dinâmica, a fim de serem usados durante a geração de cargas de trabalho sintéticas, neste capítulo foi realizada uma caracterização de alocações dinâmicas de memória em sete aplicações reais.

4.2 Instrumentação

Para realizar a caracterização do comportamento das aplicações em termos da alocação dinâmica de memória, foi necessário instrumentar suas rotinas de alocação/desalocação de memória a fim de coletar dados em tempo de execução. Para tanto, optou-se por uma abordagem menos intrusiva, realizada sem modificação do código-fonte da aplicação. Para isso, foi desenvolvido um *wrapper* do alocador de memória, o qual será daqui em diante chamado de *DebugMalloc*. Este intercepta e redireciona as chamadas de rotinas de alocação e desalocação de memória, realizando a coleta dos valores dos seus parâmetros e em seguida repassando-os para o alocador padrão (ver Figura 4.1). Com este tipo de instrumentação foi possível coletar dados com precisão para cada rotina de

alocação/desalocação, de forma menos intrusiva do que outras ferramentas, tais como *SystemTap* (SystemTap, 2012) e *Ptrace* (Linux, 2010). Para que o *DebugMalloc* seja executado é realizada a ligação dinâmica do seu código com a aplicação, em tempo de execução, por meio da variável de ambiente LD_PRELOAD que aponta para a biblioteca compartilhada contendo o *DebugMalloc*.

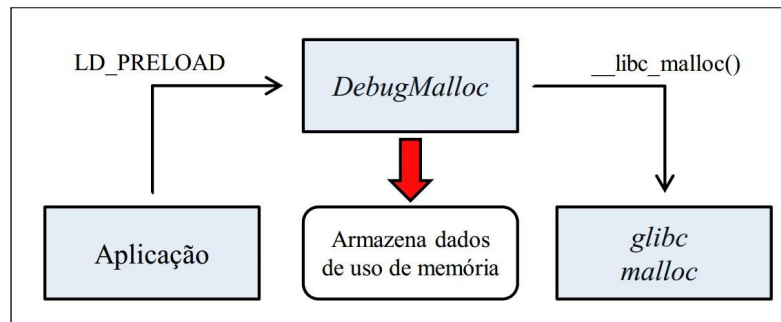


Figura 4.1. Diagrama de funcionamento do DebugMalloc.

Ao ser acionado, o DebugMalloc coleta um conjunto de dados para cada uma das operações de alocação e desalocação realizadas pelas aplicações. As Tabelas 4.1 e 4.2 apresentam os dados coletados nas operações de alocação e desalocação, respectivamente. Os dados coletados pelo DebugMalloc ficam armazenados na memória principal durante todo o experimento. Tal estratégia foi adotada para minimizar o efeito do acesso ao disco durante a execução das aplicações. Ao final do experimento o DebugMalloc salva os dados em disco. Vale ressaltar que esta abordagem possui uma limitação. Por estar ligado à aplicação como uma biblioteca, o DebugMalloc precisa guardar os dados da caracterização no disco, antes da finalização da aplicação. Dessa forma, em alguns casos, a caracterização foi feita até momentos antes da finalização da aplicação, e portanto, não contemplando a usual liberação da memória no final de sua execução. Essa limitação nos impediu de realizar uma análise sobre as alocações não desalocadas de cada aplicação.

No estudo de caracterização, não foi considerado o custo computacional da instrumentação, pois o foco do trabalho foi na identificação das operações e seus parâmetros, o que independe do tempo de execução das operações.

Tabela 4.1. Dados coletados por operação de alocação.

Dados	Descrição
Tamanho (em bytes)	Tamanho da requisição.
Tipo de operação	Rotinas de alocação requisitadas: <i>malloc</i> ^a , <i>calloc</i> e <i>realloc</i> .
Tempo (em milisegundos)	Instante em que a alocação foi requisitada.
Endereço	Endereço de memória retornado pelo alocador padrão.

^a O operador *new* chama internamente *malloc*, portanto, todo uso do *new* foi categorizado como *malloc*.

Tabela 4.2. Dados coletados por operação de desalocação.

Dados	Descrição
Tempo (em milissegundos)	Instante em que a desalocação foi requisitada.
Endereço	Endereço de memória que será liberado pela aplicação.

4.3 Aplicações Avaliadas

Na etapa de caracterização da memória, foram selecionadas sete aplicações para estudo. A escolha das aplicações se deu pelos seguintes critérios:

- A aplicação deve ser amplamente usada e relevante para o seu contexto. Esse critério foi avaliado através da quantidade de usuários, tempo de desenvolvimento e relevância da aplicação dentro do grupo de aplicações em que ela está inserida.
- A aplicação deve ser executada no sistema operacional Linux. Para manter a homogeneidade dos testes, estes foram executados apenas com o sistema operacional Linux.
- A aplicação deve ser programada nas linguagens C/C++. Tal critério se faz necessário devido à maneira escolhida para instrumentação, pois o *DebugMalloc* foi desenvolvido para interceptar as chamadas para o alocador padrão da *glibc* (Gloger, 2006); a biblioteca padrão para programas escritos em C/C++ no Linux.
- A aplicação deve utilizar o alocador padrão da *glibc*, atualmente o *Ptmalloc2*. Algumas aplicações não usam o alocador padrão e trazem o seu próprio alocador de memória, visando atender melhor ao seu perfil de alocação de memória. O *DebugMalloc* intercepta as chamadas apenas para o alocador padrão.
- A aplicação deve permitir a automatização das suas principais operações. Todos os testes foram automatizados para serem replicados sem intervenção do usuário, evitando qualquer influência não controlada.

As aplicações escolhidas contemplam duas categorias distintas de programas: *Desktop* e *Servidor*. Foram utilizadas duas aplicações da categoria *Servidor* e cinco da categoria *Desktop*. As duas aplicações *Servidor* escolhidas para a caracterização foram:

MySQL: é um gerenciador de banco de dados largamente usado, com mais de 100 milhões de cópias distribuídas (Oracle Corporation, 2014). É considerado o banco de dados *open-source* mais popular do mundo e vem sendo desenvolvido desde 1995. A versão utilizada na caracterização foi a *MySQL Community Server 5.6.12*.

Cherokee: é um *web server open-source*, leve e de alto desempenho (Ortega, 2014). Embora possua pouco tempo de desenvolvimento, alguns estudos (Barea, 2011) já apontam o Cherokee como um dos melhores *web servers* em termos de desempenho, tanto para conteúdo estático quanto dinâmico.

Utilizou-se neste estudo a versão estável 1.0. O *web server* mais utilizado, o Apache, não pôde ser utilizado neste estudo pois possui alocador próprio.

As cinco aplicações *Desktop* escolhidas para este estudo de caracterização foram:

CodeBlocks: é um ambiente integrado para desenvolvimento de software (IDE) com suporte para as linguagens C, C++ e Fortran (Elah, 2014). Vem sendo desenvolvido desde 2005 e a versão utilizada neste estudo foi a 13.2.

VLCPayer: é um reprodutor multimídia livre, *open-source*, multiplataforma e um arcabouço que reproduz a maioria dos arquivos de mídia (VideoLAN, 2014). Possui mais de 17 anos de desenvolvimento e a versão utilizada neste estudo foi a 2.1.4.

Octave: é uma linguagem interpretada de alto-nível, desenvolvida para realizar computações numéricas (GNU, 2012). Esta provê desde soluções numéricas até funcionalidades de manipulação de gráficos, semelhante ao Matlab. Vem sendo desenvolvido desde 1988 e a versão utilizada foi a 3.8.1.

Inkscape: é um editor profissional de gráficos vetoriais (Owens, 2013) com suporte para múltiplas plataformas. Possui mais de 10 anos de desenvolvimento e a versão utilizada neste estudo foi a 0.48.4.

Lynx: é um navegador *web* baseado em interface textual (Dickey, 2013). Vem sendo desenvolvido desde 1992 e a versão utilizada neste estudo foi a 2.8.7rel.2. Das aplicações escolhidas, apenas o Lynx possui uma relevância reduzida em seu contexto. Os navegadores *web* mais utilizados não puderam ser utilizados neste estudo, em sua maioria, porque não utilizam o alocador padrão da *glibc*.

4.4 Planejamento Experimental

A caracterização das alocações dinâmicas foi feita utilizando um cenário típico de uso, específico para cada aplicação. Cada teste de caracterização foi replicado trinta vezes com o propósito de minimizar a influência de erros experimentais nos resultados analisados (Montgomery, 2000). Os dados analisados foram obtidos através da média das replicações. A seguir uma descrição dos cenários de teste de cada aplicação.

4.4.1 MySQL

Para caracterizar o uso da memória do MySQL foi utilizado o banco de dados de teste Sakila-Database (Foundation, 2014b). O Sakila-Database é um banco de dados funcional, provido pela equipe de desenvolvimento do MySQL, que simula uma locadora de filmes de grande porte. O banco de dados possui 22 tabelas e utiliza todas as principais estruturas de dados e buscas do MySQL, tais como *views*, *stored procedures* e *triggers*. Para realizar as consultas foi usado o MySQLSlap (Foundation, 2014a), uma aplicação de teste que realiza um conjunto pré-determinado de operações no banco de dados de forma automatizada.

Para testar o MySQL foi criado um cenário de teste composto por três etapas: a primeira etapa consiste na criação do banco de dados Sakila, feito por uma única conexão cliente; na segunda etapa é realizado um conjunto de consultas e alterações no banco de dados feitos por 50 clientes simultâneos, onde cada cliente

realiza 36 operações entre consultas, alterações e remoções de registros. Na terceira etapa do experimento, o banco de dados de teste é apagado do disco.

4.4.2 Cherokee

Para caracterizar o uso de memória do Cherokee foi utilizado o aplicativo Apache-Bench (Apache Software Foundation, 2014a), o qual permite automatizar o acesso a páginas *web*. Foi criado um cenário de teste com vinte clientes realizando 50 acessos simultâneos à página administrativa do Cherokee.

4.4.3 CodeBlocks

O caso de teste escolhido para a caracterização do CodeBlocks consiste na inicialização de um programa e o carregamento completo de uma área de trabalho contendo o código do projeto do próprio CodeBlocks. Foi usado o código da versão 13.12 do CodeBlocks, sendo que o código e artefatos do projeto ocupam aproximadamente 20 MB de espaço em disco. Esta pode ser considerada uma aplicação de grande porte.

4.4.4 VLCPlayer

Para caracterizar o uso de memória do VLCPlayer foram criados dois casos de teste. No primeiro caso, foi realizada a execução ininterrupta de uma faixa de áudio com a duração de 5 minutos e 34 segundos. No segundo caso de teste foi feita a execução de um curta-metragem em vídeo com alta resolução com a duração de 9 minutos e 56 segundos. As Tabelas 4.3 e 4.4 apresentam em detalhes as especificações de cada arquivo executado.

Tabela 4.3. Especificações do arquivo de áudio.

Nome da Faixa	Symphony No. 5 - Allegro con brio
Autor	Ludwig Van Beethoven
Tempo de Execução	5:34
Tamanho:	10,2 MB
Taxa de bits:	256 kbps

Tabela 4.4. Especificações do arquivo de vídeo

Nome do Vídeo:	Bick Buck Bunny
Formato do Vídeo:	Ogg (<i>Ogging</i>)
Tempo de Execução:	9:56
Tamanho:	187,8 MB
Qualidade:	720p (HD)

4.4.5 Octave

O Octave dá suporte para automação de suas funcionalidades através da execução de um *script* de sua linguagem. Para caracterizar as alocações dinâmicas

de memória do Octave, foi feita a execução de um *script* com o método de eliminação de Gauss para escalonar uma matriz tridiagonal de ordem 50. Tal algoritmo é comumente utilizado para resolução de problemas lineares (Smith, 1986), uma das principais funcionalidades do Octave.

4.4.6 Inkscape

O Inkscape disponibiliza apenas um pequeno conjunto de funcionalidades que podem ser automatizadas, entre elas a inicialização e o carregamento de imagens e projetos para edição. Com essa limitação, o caso de teste escolhido foi a inicialização completa do Inkscape seguida do carregamento de uma imagem para a edição. A imagem possui resolução de 1920x108 pixels e tamanho de 280 KB.

4.4.7 Lynx

O Lynx possui nativamente um suporte para automatização do acesso às páginas que pode ser feito através da execução de um script. Para a caracterização das alocações de memória no Lynx foi executado um script que realizava o acesso a um conjunto de cinco páginas. Para escolher o conjunto de páginas usadas, foram selecionadas as cinco páginas mais acessadas da Internet segundo o Alexa Rank (Alexa, 2014).

4.4.8 Bancada de Testes

Todos os experimentos foram realizados em uma bancada de testes composta de um computador *multicore* (Intel Core i5 2410M), 6 GB de memória RAM, com Linux (Kernel 3.11.6-4-desktop) e distribuição OpenSuse 13.1. A Figura 4.2 apresenta a topologia de processador e memória *cache* do computador utilizado.

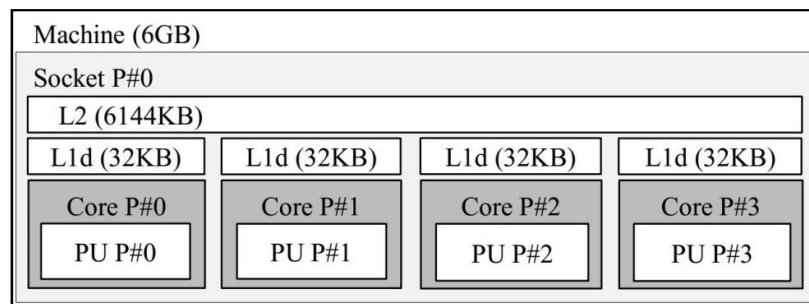


Figura 4.2. Topologia do processador usado nos experimentos.

4.5 Aspectos Avaliados

Na análise dos resultados da caracterização de cada aplicação foram analisados os seguintes aspectos: total de memória alocada, tamanhos alocados, rotinas de alocação requisitadas, memória liberada, tempo de retenção dos blocos alocados e os padrões de uso das alocações de longa duração. Nesta seção, estes

aspectos serão apresentados com mais detalhes, evidenciando a sua importância no estudo exploratório dos alocadores.

4.5.1 Total de Memória Alocada

O total de memória alocada é um dos fatores básicos que definem o uso da memória de uma aplicação. Compreender o total da memória alocada contribui para o estabelecimento da carga da aplicação sintética a ser utilizada para avaliar os alocadores. Para analisar esse aspecto, foram considerados: quantidade de alocações realizadas, tamanho médio de cada alocação e total da memória alocada durante cada caso de teste.

4.5.2 Tamanhos Alocados

Os tamanhos alocados foram analisados visando estabelecer um perfil de alocação de cada uma das aplicações. Como apresentado na Seção 3, a estratégia de alocação dos alocadores depende muito do tamanho requisitado pela aplicação. Com isso, a sua caracterização é de extrema importância para o estabelecimento de um experimento fidedigno de avaliação dos alocadores. Adicionalmente, tal informação pode ser utilizada como uma fonte para o desenvolvimento de novas estratégias de alocação, que visem explorar melhor o perfil das aplicações em questão.

Para avaliar os tamanhos alocados de cada aplicação, foi analisada a distribuição dos tamanhos de cada aplicação. Além disso, foi considerada também a quantidade de tamanhos distintos alocados, bem como o percentual das alocações dos dez tamanhos mais alocados pela aplicação.

4.5.3 Rotinas de Alocação

Cada rotina realiza a alocação da memória de uma forma específica e seu uso depende da forma como a memória será utilizada, variando de acordo com a necessidade de cada aplicação. Portanto, cada alocador implementa de forma particular cada uma das rotinas de alocação, e conhecer quais rotinas são mais utilizadas é um fator importante para a criação de um experimento de avaliação dos alocadores.

Foram analisadas as três principais rotinas usadas para alocação dinâmica de memória: *malloc*, *calloc* e *realloc*. Lembrando que o operador *new* chama a *malloc*. Vale ressaltar aqui que a liberação de memória é feita apenas com uma rotina, a *free*.

4.5.4 Memória Liberada

A liberação ou desalocação de memória é um fator muito importante no comportamento de uma aplicação, do ponto de vista da alocação dinâmica de memória. A análise da liberação de memória das aplicações foi feita apoiando-se no número de desalocações realizadas e no percentual das desalocações com ponteiros nulos. A desalocação de ponteiros nulos é um defeito (*bug*) de software e

ocorre quando a aplicação tenta desalocar uma porção de memória que foi previamente liberada ou que não havia sido alocada. Não é incomum isso provocar uma falha na aplicação. Vale ressaltar que a análise das alocações não desalocadas não foram consideradas em todos os casos neste estudo, pois devido à limitação do DebugMalloc (ver Seção 4.2), alguns casos de teste foram interrompidos antes da finalização da aplicação, não contemplando a liberação da memória ao final de sua execução.

4.5.5 Tempo de Retenção

O tempo de retenção de um bloco de memória alocado é o tempo entre a sua alocação e desalocação. Este tempo é um fator muito importante para a análise e projeto de alocadores de memória, uma vez que ele tem influência na fragmentação da memória, um dos principais problemas no uso da alocação dinâmica (Wilson et al., 1995). Portanto, caracterizar esse tempo é útil no estudo dos padrões de alocação de memória. Para analisar o tempo de retenção dos blocos de memória alocados pelas aplicações investigadas, os tempos de retenção de cada bloco foram classificados em três categorias:

Alocações de curta duração: Alocações cujo tempo de retenção é de até 100 milissegundos.

Alocações de média duração: Alocações cujo tempo de retenção está entre 100 milissegundos e 1 segundo.

Alocações de longa duração: Alocações cujo tempo de retenção é maior do que 1 segundo. Estas alocações representam, na sua maioria, estruturas de dados que permanecem alocadas durante toda a execução da aplicação.

As faixas de valores utilizadas para distinguir os tempos de retenção foram definidas com base nos próprios dados experimentais. Depois da categorização dos blocos, a análise do tempo de retenção das aplicações foi feita através da porcentagem encontrada das alocações de curta, média e longa duração, com relação ao total das alocações.

4.5.6 Padrões de Uso das Alocações de Longa Duração

As alocações de longa duração representam estruturas de dados que podem permanecer alocadas durante todo o tempo de vida de uma aplicação. Por essa razão, este aspecto foi analisado com mais detalhes de forma isolada. As alocações de longa duração influenciam diretamente na fragmentação externa do alocador de memória (Wilson et al., 1995). Portanto, identificar como as aplicações utilizam este tipo de alocação pode contribuir com o estudo sobre os alocadores, levantando informações que podem ajudar a estabelecer uma estratégia que minimize os efeitos da fragmentação.

De acordo com o comportamento das alocações de longa duração durante a execução, as aplicações foram analisadas e categorizadas em dois padrões distintos de comportamento:

Pico: A aplicação realiza a maior parte das alocações de longa duração em um conjunto pequeno e determinado de operações, evidenciado por um pico nas alocações de longa duração. Fora da área do pico, as alocações de

longa duração se mantêm em um nível aproximadamente constante, mostrando que a maior parte das alocações após o pico são alocações de curta ou média duração.

Uso crescente: A aplicação incrementa as alocações de longa duração gradativamente, evidenciado pelo crescimento aproximadamente linear do número de alocações de longa duração. Neste caso, parte das alocações realizadas ao longo do experimento não é desalocada imediatamente, aumentando o uso de memória durante o seu período de execução. Teoricamente, este perfil de alocação tende a gerar mais fragmentação externa do alocador, pois parte das alocações não será imediatamente desalocada, gerando espaços vagos entre as alocações de longa duração.

Pela grande quantidade de operações, as análises foram realizadas quantificando de forma cumulativa as alocações de longa duração a cada 1000 operações de alocação/desalocação.

4.6 Análise dos Resultados por Aplicações

Nesta seção será apresentada uma análise dos resultados da caracterização das alocações dinâmicas de memória obtida em cada aplicação.

4.6.1 MySQL

A Tabela 4.5 apresenta os resultados obtidos na caracterização da alocação dinâmica de memória no MySQL. O MySQL alocou no total cerca de 686 MB durante o teste, em mais de 467 mil operações de alocações. Com relação aos tamanhos alocados, foram alocados no total 775 tamanhos diferentes, no entanto é possível observar na Figura 4.3 que a maior parte das alocações estão concentradas nos tamanhos de 1144 bytes (cerca de 60%), 128 bytes (8%) e 280 bytes (6%), os quais representam 75% das alocações. Se essa análise for ampliada, será possível constatar que apenas os dez tamanhos mais alocados no MySQL representam quase 88% do total de suas alocações. Tal comportamento mostra que, embora o MySQL realize um grande número de alocações distintas, as alocações são concentradas em um conjunto pequeno de tamanhos. Isso sugere que as mesmas estruturas de dados tenham sido alocadas e realocadas durante o experimento. Além disso, o MySQL praticamente só alocou memória utilizando a rotina *malloc*, obtendo um percentual de 99,91% de uso.

Tabela 4.5. Alocações, tamanhos alocados e rotinas de alocação do MySQL.

Alocações			Tamanhos Alocados		Rotinas de Alocação		
Número de Alocações	Memória Alocada (MB)	Tamanho Médio Alocado (Bytes)	Total de Tamanhos Alocados	% dos 10 Tamanhos mais Alocados	malloc	realloc	calloc
467.348	686	1.467,40	755	87,9	99,91%	0,06%	0,03%

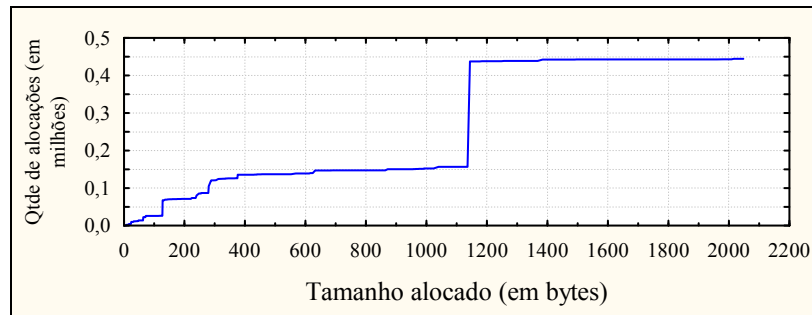


Figura 4.3. Quantidade acumulada dos tamanhos alocados no MySQL.

Por ser uma aplicação do tipo *Servidor*, o caso de teste do MySQL pôde ser finalizado antes do término do experimento. Com isso, ao final do teste a memória alocada pelo MySQL foi liberada pela própria aplicação. É possível notar que o número de desalocações feitas ultrapassou o número de alocações em cerca de 5%. Isso ocorreu devido ao elevado número de alocações nulas realizadas pelo MySQL, cerca de 11% das liberações de memória foram feitas passando um ponteiro nulo para o DebugMalloc.

Tabela 4.6. Resultados obtidos na caracterização do MySQL.

Desalocações		Tempo de Retenção		
Número de Desalocações	Desalocações Nulas	Alocações de Curta Duração	Alocações de Média Duração	Alocações de Longa Duração
493.965	11,80%	87,65%	5,31%	7,04%

Com relação ao tempo de retenção das alocações (ver Tabela 4.6), o MySQL alocou em sua maioria estruturas de dados temporárias, sendo a maior parte delas (87,65%) desalocadas em até 100 milissegundos. Das alocações realizadas, apenas 7,04% foram mantidas por mais de um segundo e representam as estruturas de dados mantidas durante a maior parte da execução do MySQL. Note ainda que o MySQL realiza a grande parte de suas alocações de longa duração no início do experimento, mantendo o nível de alocações de longa duração aproximadamente constante no resto do experimento (ver Figura 4.4). Dessa forma, o MySQL se enquadrado no padrão *Pico*.

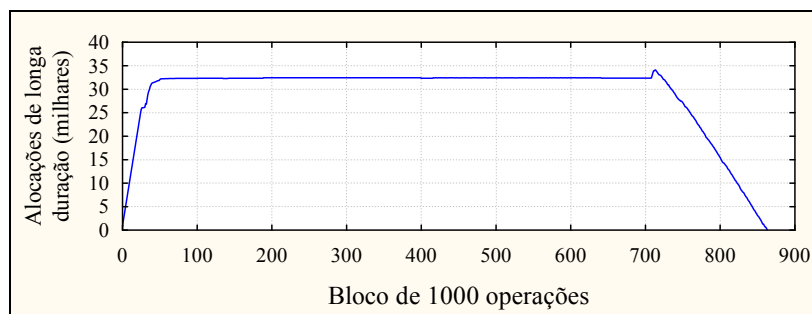


Figura 4.4. Alocações de longa duração agrupadas por blocos de 1000 operações (alocações/desalocações) da aplicação MySQL.

4.6.2 Cherokee

O Cherokee alocou 14 MB durante o caso de teste, em 31.727 alocações, com o tamanho médio de 420 bytes por alocação (Tabela 4.7). Com relação ao tamanho alocado, o Cherokee alocou um total de 114 tamanhos diferentes, no entanto, mais de 92% das alocações foram concentradas nos dez tamanhos mais alocados da aplicação. Através da Figura 4.5 é possível observar que a maior parte das alocações foi feita com requisições de 264 bytes (cerca de 80%) e que o restante das alocações foram feitas com tamanhos pequenos (abaixo de 64 bytes). Isso mostra que possivelmente uma única estrutura de dados foi alocada e realocada continuamente durante o experimento. A rotina mais utilizada para a alocação foi a *malloc* (87,49%), seguida da *realloc* (12,38%).

Tabela 4.7. Alocações, tamanhos alocados e rotinas de alocação do Cherokee.

Alocações			Tamanhos Alocados		Rotinas de Alocação		
Número de Alocações	Memória Alocada (MB)	Tamanho Médio Alocado (Bytes)	Total de Tamanhos Alocados	% dos 10 Tamanhos mais Alocados	malloc	realloc	calloc
31.727	14	420,13	114	92,12	87,49%	12,38%	0,14%

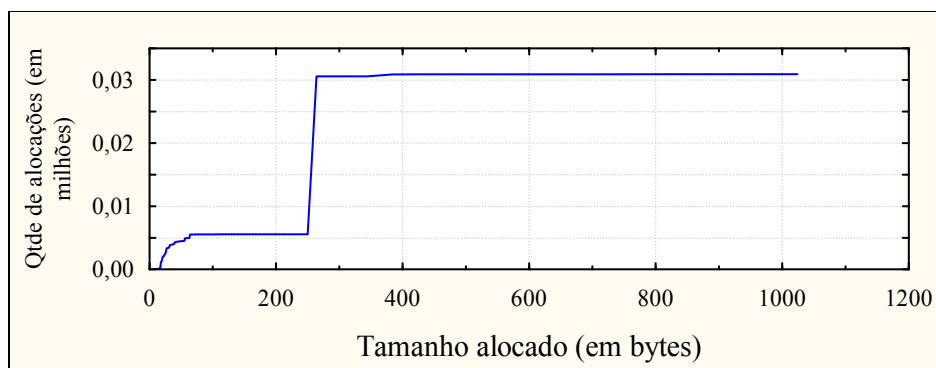


Figura 4.5. Quantidade acumulada dos tamanhos alocados pelo Cherokee.

Assim como o MySQL, o Cherokee foi finalizado apropriadamente no final do experimento, e por isso desalocou 91,44% das alocações realizadas, como mostra a Tabela 4.8. Dessa forma, apenas 8,56% das alocações não foram desalocadas e apenas 0,02% das desalocações foram feitas com ponteiros nulos. A maior parte das alocações foi categorizada como de curta duração (85,16%), enquanto menos de 10% das alocações permaneceram alocadas por mais de um segundo. Isso mostra que a maior parte das alocações do Cherokee foi para uso temporário.

Ainda com relação às alocações de longa duração, através da Figura 4.6 é possível observar que a aplicação alocou a maior parte de suas alocações de longa duração no início de sua execução. A partir das 10.000 operações de alocação/desalocação o Cherokee realizou praticamente apenas alocações de média ou curta duração. Com isso, o Cherokee foi enquadrado no padrão *Pico*.

Tabela 4.8. Desalocações e tempo de retenção do Cherokee.

Desalocações		Tempo de Retenção		
Número de Desalocações	Desalocações Nulas	Alocações de Curta Duração	Alocações de Média Duração	Alocações de Longa Duração
29.017	0,02%	85,16%	5,28%	9,56%

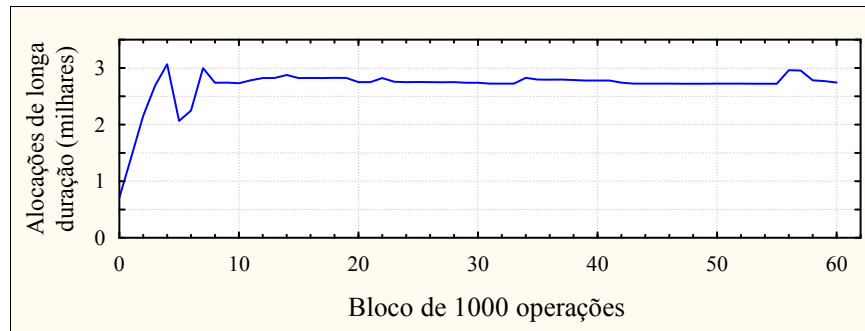


Figura 4.6. Alocações de longa duração agrupadas por blocos de 1000 operações (alocações/desalocações) da aplicação Cherokee.

4.6.3 CodeBlocks

Dentre as aplicações caracterizadas, o CodeBlocks foi a que obteve o maior consumo de memória. A Tabela 4.9 apresenta os resultados. É possível notar que o CodeBlocks alocou quase 2,5 GB com mais de 12 milhões de alocações. A aplicação alocou memória com uma grande diversidade de tamanhos, tendo sido 3.806 tamanhos diferentes. No entanto, assim como as aplicações já analisadas, a maior parte de suas alocações (86,56%) está concentrada em um grupo de dez tamanhos distintos. Além disso, através da distribuição dos tamanhos apresentada na Figura 4.7 é possível observar que a maioria das alocações do CodeBlocks é feita com tamanhos menores do que 128 bytes. A alocação do CodeBlocks foi feita em sua maioria pela rotina *malloc* (92,51%), seguida pela *realloc* (7,18%).

Tabela 4.9. Alocações, tamanhos alocados e rotinas de alocação do CodeBlocks.

Alocações			Tamanho Alocado		Rotinas de Alocação		
Número de Alocações	Memória Alocada (MB)	Tamanho Médio Alocado (Bytes)	Total de Tamanhos Alocados	% dos 10 Tamanhos mais Alocados	malloc	realloc	calloc
12.412.302	2.493	200,92	3.806	86,56	92,51%	7,18%	0,31%

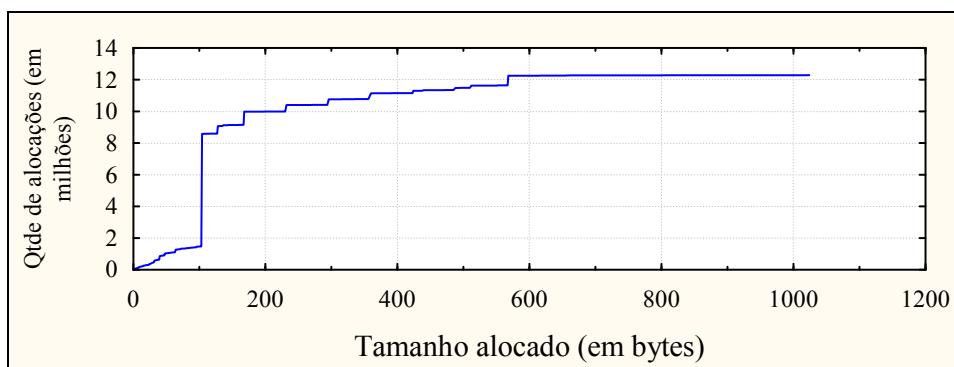


Figura 4.7. Quantidade acumulada dos tamanhos alocados pelo CodeBlocks.

Com relação às desalocações, devido às limitações nas funcionalidades de automatização dos testes do CodeBlocks, este foi interrompido antes de sua finalização normal. Tal característica do teste pode justificar, em partes, porque 13,38% de suas alocações não foram desalocadas (Tabela 4.10). Dentre as suas desalocações, apenas 2,27% foram feitas com ponteiros nulos. Assim como nas aplicações até aqui analisadas, a maior parte (85%) das alocações do CodeBlocks foi para uso temporário, sendo desalocadas em até um segundo (somadas as alocações de curta e média duração).

Tabela 4.10. Desalocações e tempo de retenção do CodeBlocks.

Desalocações		Tempo de Retenção		
Número de Desalocações	Desalocações Nulas	Alocações de Curta Duração	Alocações de Média Duração	Alocações de Longa Duração
11.001.975	2,27%	81,96%	3,74%	14,31%

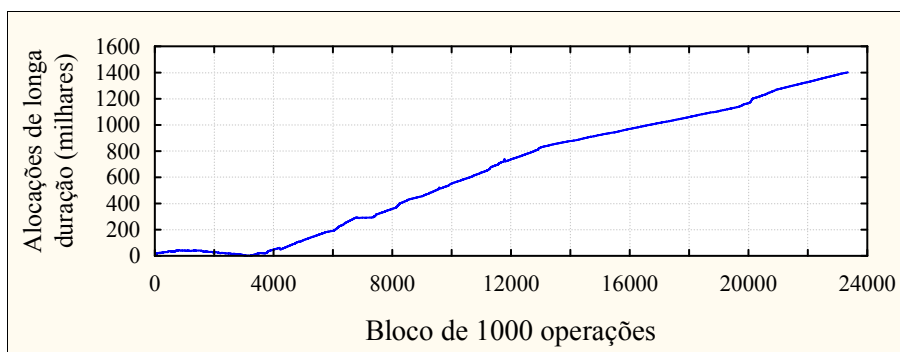


Figura 4.8. Alocações de longa duração agrupadas por blocos de 1000 operações (alocações/desalocações) da aplicação CodeBlocks.

Os quase 15% das alocações de longa duração do CodeBlocks foram alocadas durante todo o experimento, como mostra a Figura 4.8. A partir do bloco 4000, as alocações de longa duração iniciam um crescimento aproximadamente linear até o final do experimento. Por esse motivo, o CodeBlocks foi enquadrado no padrão *Uso Crescente*.

4.6.4 VLCPlayer (áudio)

O teste feito com o VLCPlayer executando um arquivo de áudio resultou em 138.747 alocações, com um total de 335 MB alocados (Tabela 4.11). Note que o VLCPlayer(áudio) apresentou uma distribuição maior com relação aos seus tamanhos alocados. É possível observar na Figura 4.9 que, embora a maior parte das alocações foi realizada em até 100 bytes, há dois outros intervalos de tamanhos que contribuíram significativamente para as alocações. Os tamanhos mais alocados foram 72 bytes e 9.392 bytes com um percentual de 18,8% e 18,3% cada. Com uma maior distribuição dos tamanhos alocados, a porcentagem da concentração nos dez tamanhos alocados foi de 76,11%, um percentual menor do que encontrado na média das aplicações. Com relação às rotinas de alocação, o VLCPlayer é uma das aplicações que melhor distribui as suas formas de alocação, com *malloc* sendo a mais utilizada (75,40%), seguida da *calloc* (19,89%) e por último a *realloc* (4,71%).

Tabela 4.11. Alocações, tamanhos alocados e rotinas de alocação do VLCPlayer(áudio).

Alocações			Tamanho Alocado		Rotinas de Alocação		
Número de Alocações	Memória Alocada (MB)	Tamanho Médio Alocado (Bytes)	Total de Tamanhos Alocados	% dos 10 Tamanhos mais Alocados	malloc	realloc	calloc
138.747	335	2.410,82	911	76,41	75,40%	4,71%	19,89%

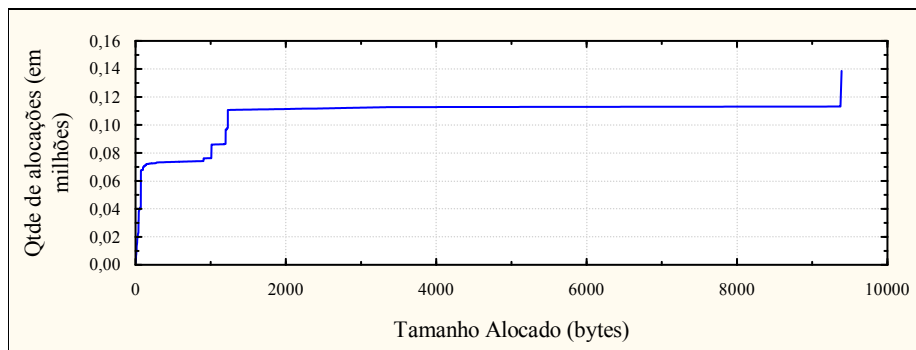


Figura 4.9. Quantidade acumulada dos tamanhos alocados pelo VLCPlayer (áudio).

A Tabela 4.12 apresenta os resultados do VLCPlayer (áudio) com relação à sua desalocação e tempo de retenção. É possível observar que VLCPlayer (áudio) realizou 120.051 alocações, dentre as quais apenas 2,37% foram chamadas com ponteiro nulo. Outra característica que deve ser ressaltada é que mais de 36% das alocações foram classificadas como de média e longa duração. Comparando esses resultados com as outras aplicações, é possível observar que o VLCPlayer (áudio) retém mais a memória alocada do que a média encontrada nas demais aplicações testadas (apenas 25% das alocações são de média e longa duração na média observada neste estudo).

Tabela 4.12. Desalocações e tempo de retenção do VLCPlayer (áudio).

Desalocações		Tempo de Retenção		
Número de Desalocações	Desalocações Nulas	Alocações de Curta Duração	Alocações de Média Duração	Alocações de Longa Duração
120.051	2,37%	63,43%	20,57%	15,99%

As alocações de longa duração do VLCPlayer (áudio) foram alocadas em sua maioria no primeiro conjunto de alocações do experimento. Após o bloco 50, a maior parte das alocações do VLCPlayer (áudio) foi para uso temporário e foram desalocadas imediatamente. Por esse motivo, o VLCPlayer (áudio) foi enquadrado no padrão Pico (ver Figura 4.10).

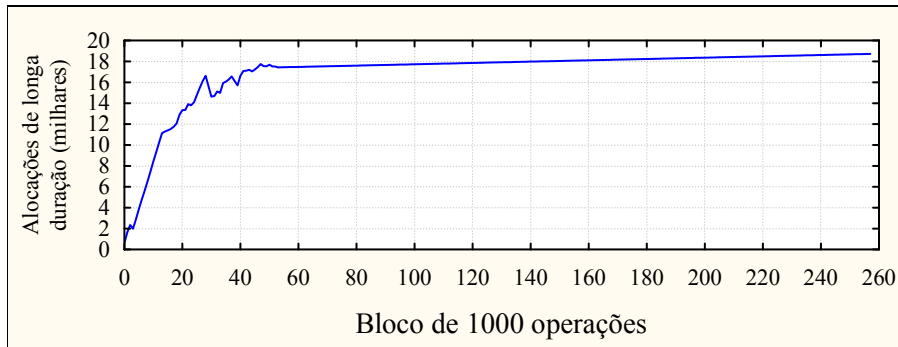


Figura 4.10. Alocações de longa duração agrupadas por blocos de 1000 operações (alocações/desalocações) da aplicação VLCPlayer (áudio).

4.6.5 VLCPlayer (vídeo)

O VLCPlayer (vídeo) utilizou 1,8 GB, alocados em 1.513.223 requisições de alocação (Tabela 4.13). Dentre as aplicações testadas, o VLCPlayer executado com um arquivo de vídeo apresentou o maior número de alocações com tamanhos distintos, alocando 4020 tamanhos diferentes. Por conta dessa alta distribuição, a porcentagem dos dez tamanhos mais alocados do VLCPlayer (áudio) foi de apenas 59,27% das alocações, um dos percentuais mais baixos dentre as aplicações consideradas neste estudo. Ainda assim, é possível ver na Figura 4.11 que a maior parte das alocações foi feita com tamanhos pequenos de até 200 bytes. Diferente do teste feito com o áudio, o VLCPlayer (vídeo) utilizou em quase 80% de suas alocações a rotina *malloc*, seguido pela *calloc* com 14,79%, e por último pela *realloc* com 5,49% das alocações.

Tabela 4.13. Alocações, tamanhos alocados e rotinas de alocação do VLCPlayer (vídeo).

Alocações			Tamanhos Alocados		Rotinas de Alocação		
Número de Alocações	Memória Alocada (MB)	Tamanho Médio Alocado (Bytes)	Total de Tamanhos Alocados	% dos 10 Tamanhos mais Alocados	malloc	realloc	calloc
1.513.223	1.885	1.245,42	4020	59,27	79,72%	5,49%	14,79%

O VLCPlayer (vídeo) desalocou 86,62% de suas alocações durante o seu caso de teste, e 2,05% das requisições de liberação foram feitas com ponteiros nulos, como mostra a Tabela 4.14. Com relação ao tempo de retenção, assim como no teste feito com o arquivo de áudio, o VLCPlayer (vídeo) manteve por mais tempo os seus blocos de memória alocados, com 51,26% das alocações sendo classificadas como média e longa duração. Uma possível explicação para o tempo de retenção alto obtido com os testes envolvendo o VLCPlayer é de que tais alocações foram feitas para um sistema de *buffer* de áudio/vídeo e que por isso demora mais tempo para serem desalocadas.

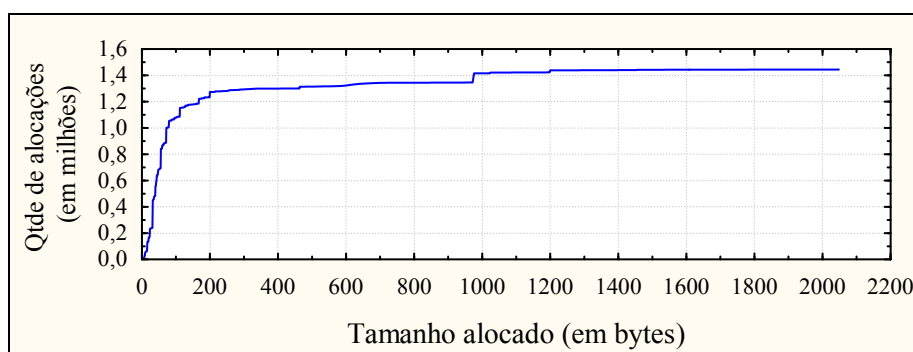


Figura 4.11. Quantidade acumulada dos tamanhos alocados pelo VLCPlayer (vídeo).

Tabela 4.14. Desalocações e tempo de retenção do VLCPlayer (vídeo).

Desalocações		Tempo de Retenção		
Número de Desalocações	Desalocações Nulas	Alocações de Curta Duração	Alocações de Média Duração	Alocações de Longa Duração
1.397.594	2,05%	49,84%	16,88%	33,28%

A Figura 4.12 apresenta as alocações de longa duração acumuladas e agrupadas a cada bloco de 1000 operações de alocação/desalocação. É possível ver que durante todo o experimento o VLCPlayer (vídeo) realizou continuamente

alocações com mais de um segundo de retenção. Por conta disso, o VLCPlayer (vídeo) foi enquadrado no padrão *Uso Crescente*.

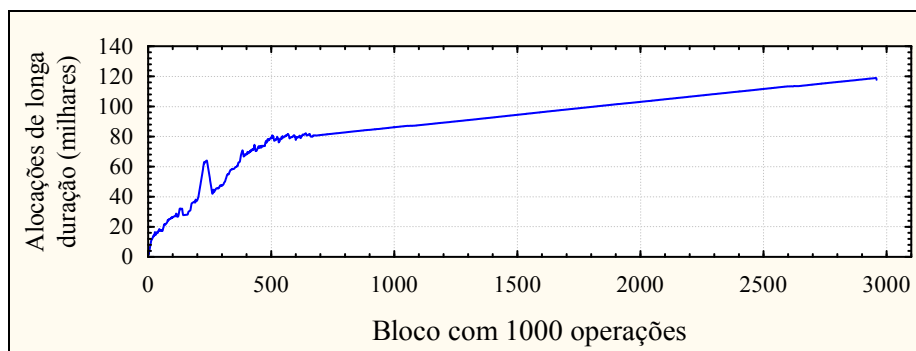


Figura 4.12. Alocações de longa duração agrupadas por blocos de 1000 operações (alocações/desalocações) da aplicação VLCPlayer (vídeo).

4.6.6 Octave

O Octave alocou cerca de 100 MB durante o seu caso de teste, divididos entre 579.606 alocações (ver Tabela 4.15). O Octave alocou 789 tamanhos diferentes, no entanto, a maior parte das suas alocações foi feita com tamanhos até 64 bytes, como mostra a Figura 4.13. Além disso, 70,14% das alocações foram feitas com os dez tamanhos mais alocados pela aplicação que estão situados quase inteiramente na faixa entre 16 e 64 bytes, com exceção das alocações feitas com 512 bytes que obtiveram a nona colocação dos dez tamanhos mais alocados. As alocações do Octave foram feitas quase inteiramente pela rotina *malloc*, que foi utilizada em 99,71% das alocações.

Tabela 4.15. Alocações, tamanhos alocados e rotinas de alocação do Octave.

Alocações			Tamanhos Alocados		Rotinas de Alocação		
Número de Alocações	Memória Alocada (MB)	Tamanho Médio Alocado (Bytes)	Total de Tamanhos Alocados	% dos 10 Tamanhos mais Alocados	malloc	realloc	calloc
579.606	100	172,07	789	70,14	99,71%	0,08%	0,21%

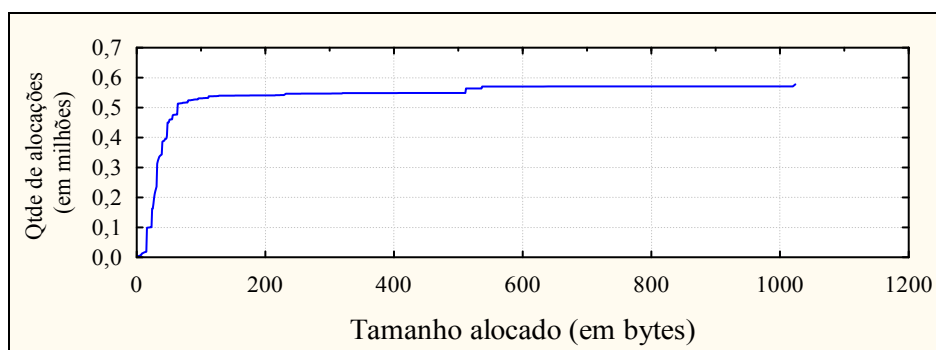


Figura 4.13. Quantidade acumulada dos tamanhos alocados pelo Octave.

O Octave desalocou apenas 80,4% das suas alocações (ver Tabela 4.16), dentre as quais menos de 1% foram feitas com ponteiros nulos. Essa baixa taxa de desalocação é justificada em partes pela interrupção do *Octave* ao final do experimento. Quase 80% das alocações foram desalocadas em até um segundo um percentual comum, dentro da média das aplicações testadas.

Tabela 4.16. Desalocações e tempo de retenção do Octave.

Desalocações		Tempo de Retenção		
Número de Desalocações	Desalocações Nulas	Alocações de Curta Duração	Alocações de Média Duração	Alocações de Longa Duração
464.206	0,96%	74,81%	4,49%	20,70%

Cerca de 20% das alocações foram retidas por mais de um segundo pelo Octave. O Octave realizou as alocações de longa duração durante todo o experimento, comportamento evidenciado pelo crescimento aproximadamente linear apresentado na Figura 4.14. Consequentemente, o Octave foi enquadrado como sendo do padrão *Uso Crescente*.

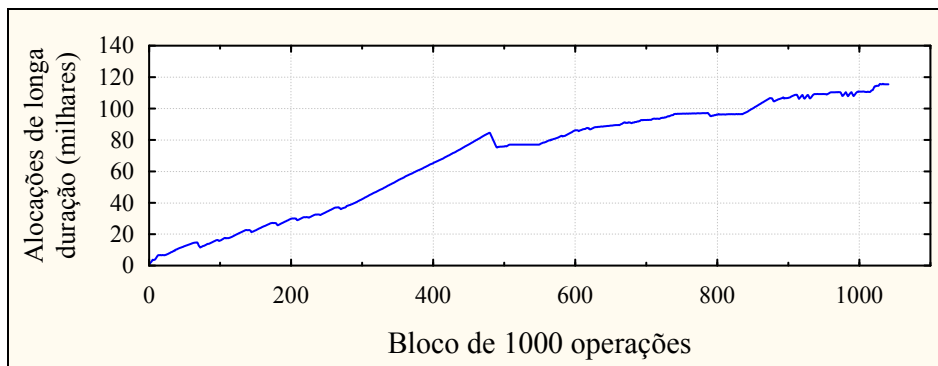


Figura 4.14. Alocações de longa duração agrupadas por blocos de 1000 operações (alocações/desalocações) da aplicação Octave.

4.6.7 Inkscape

Dentre as aplicações avaliadas, o Inkscape foi a segunda aplicação com o maior número de alocações realizadas (Tabela 4.17). No entanto, como o tamanho médio apresentado foi de apenas 46 bytes, o total alocado pelo Inkscape foi de apenas 564 MB. Com esses dados já é possível traçar um perfil de distribuição do Inkscape, o qual realiza em sua maioria alocações de até 64 bytes, como mostra a Figura 4.15. No total, o Inkscape alocou 1441 tamanhos diferentes, e quase 74% de suas alocações estão concentradas nos dez tamanhos mais alocados. O Inkscape utilizou na maior parte de suas alocações a rotina *malloc* com 95,61% das alocações, seguida pela *realloc* (2,93%) e pela *calloc* (1,46%).

Tabela 4.17. Alocações, tamanhos alocados e rotinas de alocação do Inkscape.

Alocações			Tamanhos Alocados		Rotinas de Alocação		
Número de Alocações	Memória Alocada (MB)	Tamanho Médio Alocado (Bytes)	Total de Tamanhos Alocados	% dos 10 Tamanhos mais Alocados	malloc	realloc	calloc
12.172.463	564	46,31	1441	73,99	95,61%	2,93%	1,46%

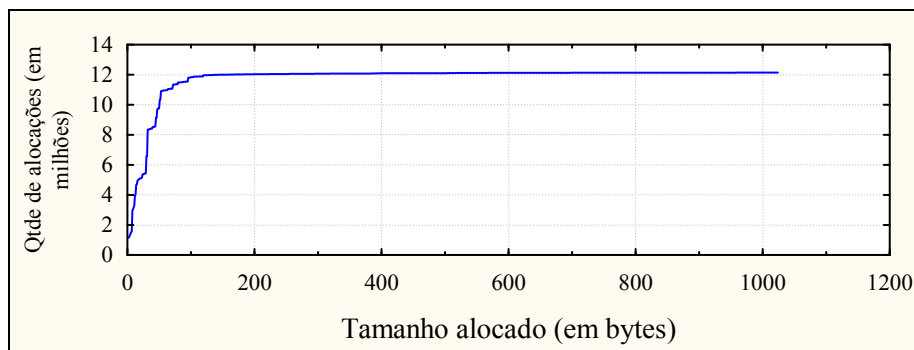


Figura 4.15. Quantidade acumulada dos tamanhos alocados pelo Inkscape.

Embora o caso do teste do Inkscape tenha sido interrompido após a inicialização de sua área de trabalho com a figura escolhida para o teste, apenas 6,20% de suas alocações não foram desalocadas (Tabela 4.18). Além disso, apenas 0,11% das desalocações foram feitas com ponteiros nulos. Com um alto índice de desalocação, mesmo tendo seu caso de teste interrompido, o Inkscape foi a aplicação que menos reteve blocos de memória, com quase 95% tendo sido desalocados em até um segundo.

Tabela 4.18. Desalocações e tempo de retenção do Inkscape.

Desalocações		Tempo de Retenção		
Número de Desalocações	Desalocações Nulas	Alocações de Curta Duração	Alocações de Média Duração	Alocações de Longa Duração
11.430.033	0,11%	89,49%	4,24%	6,27%

As alocações de longa duração representaram apenas 6,27% das alocações e foram alocadas em sua maioria no primeiro conjunto de operações de alocação do experimento. Devido a esse comportamento, o Inkscape foi enquadrado no padrão *Pico* (ver Figura 4.16).

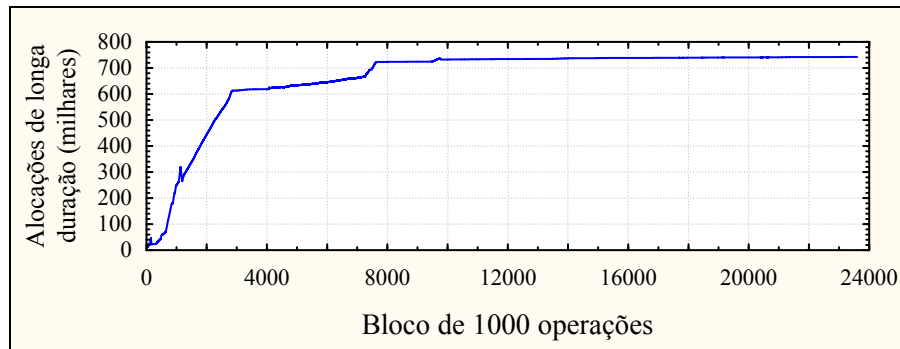


Figura 4.16. Alocações de longa duração agrupadas por blocos de 1000 operações (alocações/desalocações) da aplicação Inkscape.

4.6.8 Lynx

Dentre as aplicações testadas, o Lynx foi a aplicação com a menor quantidade de memória alocada (9MB), utilizando para isso 31.303 operações de alocação (Tabela 4.19). A distribuição dos tamanhos foi outro comportamento atípico encontrado no Lynx, com 554 tamanhos diferentes, o Lynx distribuiu melhor as suas alocações, apenas 50,37% foram concentradas nos dez tamanhos mais usados. Embora mais bem distribuídas, através da Figura 4.17 é possível identificar que a maior parte das suas alocações foi feita com tamanhos de até 100 bytes. Outra característica atípica é que embora a *malloc* (72,84%) tenha sido a rotina mais utilizada, o Lynx ainda utiliza com certa frequência as rotinas *realloc* (15,72%) e *calloc* (11,44%), diferente dos respectivos percentuais de uso encontrados nas demais aplicações avaliadas.

Tabela 4.19 Alocações, tamanhos alocados e rotinas de alocação do Lynx.

Alocações			Tamanhos Alocados		Rotinas de Alocação		
Número de Alocações	Memória Alocada (MB)	Tamanho Médio Alocado (Bytes)	Total de Tamanhos Alocados	% dos 10 Tamanhos mais Alocados	malloc	realloc	calloc
31.303	9	268,51	554	50,37	72,84%	15,72%	11,44%

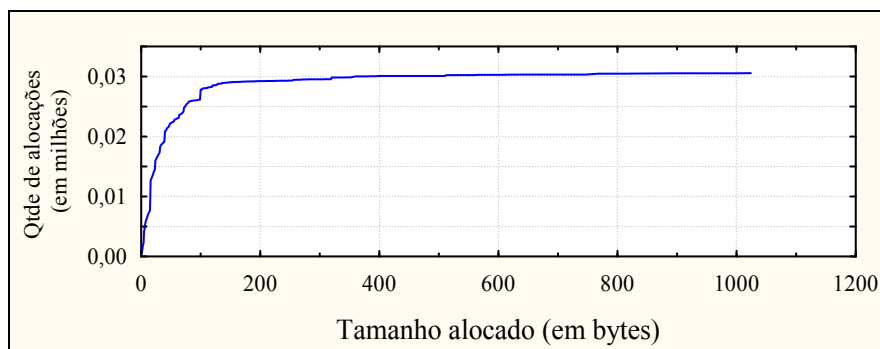


Figura 4.17. Quantidade acumulada dos tamanhos alocados pelo Lynx.

O caso de teste do Lynx foi finalizado antes de seu fechamento apropriado. Por conta disso, cerca de 55% das alocações não foram desalocadas (ver Tabela 4.20). Menos de 1% das desalocações foram feitas com ponteiro nulo. O Lynx foi a aplicação que mais reteve alocações por mais de um segundo (56,98%). Uma explicação para o alto tempo de retenção do Lynx se deve ao seu perfil de uso. Mesmo com um teste automatizado, o tempo de acesso do Lynx depende do acesso à rede que pode levar mais do que um segundo em determinados casos. O acesso às páginas na Internet pode ser considerado um processo lento maior do que 1 segundo), de modo que boa parte das alocações do Lynx poderiam ser desalocadas apenas após o acesso a uma nova página.

Tabela 4.20. Desalocações e tempo de retenção do Lynx.

Desalocações		Tempo de Retenção		
Número de Desalocações	Desalocações Nulas	Alocações de Curta Duração	Alocações de Média Duração	Alocações de Longa Duração
14.234	0,84%	40,75%	2,27%	56,98%

Analisando mais profundamente o comportamento da aplicação com relação às alocações de longa duração, é possível observar que este tipo de alocação foi feito durante todo o experimento realizado (ver Figura 4.18). Dessa forma, foi considerado que o Lynx enquadra-se no padrão *Uso Crescente*.

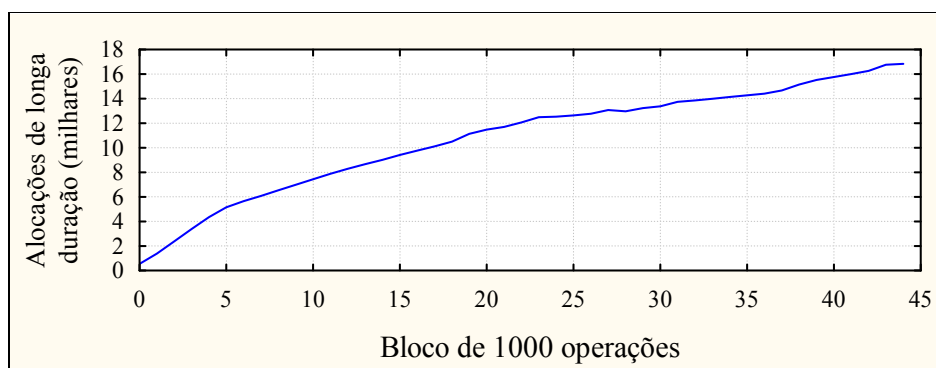


Figura 4.18. Alocações de longa duração agrupadas por blocos de 1000 operações (alocações/desalocações) da aplicação Lynx.

4.7 Análise Comparativa dos Resultados

Nesta seção será apresentada a análise comparativa dos resultados obtidos em cada caracterização, destacando os padrões de uso de memória encontrados.

4.7.1 Total de Memória Alocada

Observou-se uma considerável diversidade na quantidade de memória alocada e no número de alocações feitas pelas aplicações (ver Tabela 4.21). A

memória total alocada por cada aplicação variou entre 9 MB com 30.000 alocações (Lynx) e 2,4 GB (CodeBlocks) alocados em mais de 12,4 milhões de alocações. Neste estudo, a aplicação que mais alocou memória foi também a que mais vezes requisitou o alocador. No entanto, o caso de teste com o Inkscape mostrou que embora este tenha requisitado 12,1 milhões de alocações, a sua memória total utilizada foi apenas de 564MB, devido ao seu baixo tamanho médio de alocação.

Tabela 4.21. Memória alocada por aplicação.

Aplicação	Número de Alocações	Memória Alocada (MB)	Tamanho Médio Alocado (Bytes)
MySQL	467.348	686	1.467,40
Cherokee	31.727	14	420,13
CodeBlocks	12.412.302	2.493	200,92
VLCPlayer (áudio)	138.747	335	2.410,82
VLCPlayer (vídeo)	1.513.223	1.885	1.245,42
Octave	579.606	100	172,07
Inkscape	12.172.463	564	46,31
Lynx	31.303	9	268,51

4.7.2 Tamanhos Alocados

A maior parte das alocações das aplicações está concentrada em uma faixa bem definida de tamanhos. Todas as aplicações *Desktop* testadas obtiveram a maioria das alocações com tamanhos de até 128 bytes, enquanto as aplicações MySQL e Cherokee, ambas do tipo *Servidor*, apresentaram a maior parte das alocações com tamanhos 1144 e 264 bytes, respectivamente. Vale ressaltar que o teste feito com o VLCPlayer (áudio) também obteve uma considerável parcela das alocações com tamanhos de 1000 até 1230 e 9392 bytes.

As aplicações alocaram, em média, 1548 tamanhos distintos em seu tempo de execução, no entanto, a maior parte das alocações está concentrada em um grupo de dez tamanhos. Em média, os dez tamanhos mais alocados representam 75,2% do total das alocações, variando de 50% com o Lynx até 92% com o Cherokee (ver Figura 4.19).

Geradores de carga de trabalho sintética, para testes de alocadores de memória, tendem a tratar o tamanho das alocações de memória de forma aleatória, alocando tamanhos randômicos para cada requisição (Costa et al., 2013) ou fixando um tamanho único para as alocações (Berger et al., 2000) (Michael, 2004). O comportamento observado neste estudo sugere que uma junção destas duas práticas oferece uma carga de trabalho mais próxima de uma aplicação real, ao se fixar um pequeno conjunto de tamanhos como a maior parte das alocações e aleatorizar, de forma homogênea, as demais alocações de memória.

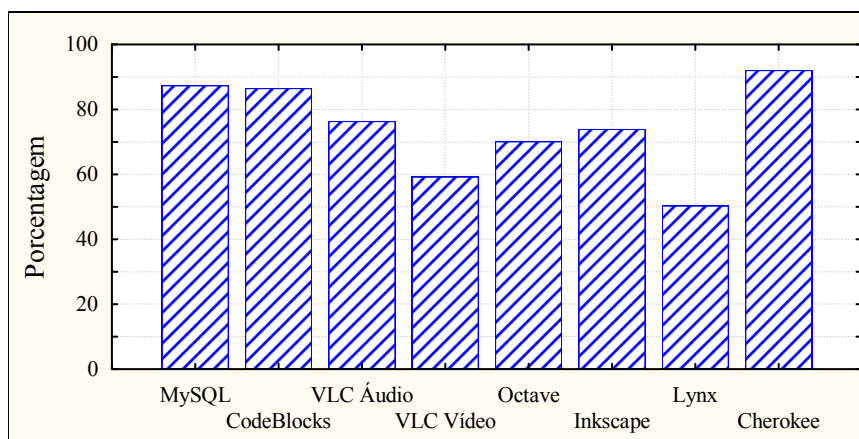


Figura 4.19. Porcentagem dos dez tamanhos mais alocados por aplicação.

4.7.3 Rotinas de Alocação

Em todas as aplicações a rotina mais usada foi a *malloc*, com média de 87,9% do total das alocações, seguida pela *realloc* com 6,07% e *calloc* com 6,03%. De fato, apenas as aplicações VLCPlayer e Lynx utilizaram as outras rotinas com maior frequência (ver Tabela 4.22).

Tabela 4.22. Porcentagem do uso das rotinas de Alocação.

Aplicação	malloc	realloc	calloc
MySQL	99,91	0,06	0,03
Cherokee	87,49	12,38	0,14
Code Blocks	92,51	7,18	0,31
VLCPlayer (áudio)	75,40	4,71	19,89
VLCPlayer (vídeo)	79,72	5,49	14,79
GNU Octave	99,71	0,08	0,21
Inkscape	95,61	2,93	1,46
Lynx	72,84	15,72	11,44

4.7.4 Memória Liberada

Os resultados mostraram que o banco de dados MySQL foi a aplicação que mais realizou desalocações com ponteiros nulos, representando quase 12% de suas desalocações, seguido pelo VLCPlayer (áudio) com 2,37%, CodeBlocks com 2,27% e o VLCPlayer (vídeo) com 2,05%. As demais aplicações obtiveram percentuais de desalocações nulas abaixo de 1% (ver Tabela 4.23).

Tabela 4.23. Desalocação por aplicação.

Aplicação	Número de Desalocações	% de Desalocações Nulas
MySQL	493.965	11,80
Cherokee	29.017	0,02
CodeBlocks	11.001.975	2,27
VLCPlayer (áudio)	120.051	2,37
VLCPlayer (vídeo)	1.397.594	2,05
Octave	464.206	0,96
Inkscape	11.430.033	0,11
Lynx	14.234	0,84

4.7.5 Tempo de Retenção

Os resultados mostraram que, em média, 71,6% das alocações foram classificadas como de curta duração, enquanto 7,85% foram classificadas como média duração, ou seja, quase 80% das alocações são desalocadas em até 1 segundo (ver Figura 4.20). No caso das aplicações *Servidor*, essa média aumenta para 86%. Esse resultado mostra que a maioria das alocações é para uso temporário, ou seja, tempo de retenção curto.

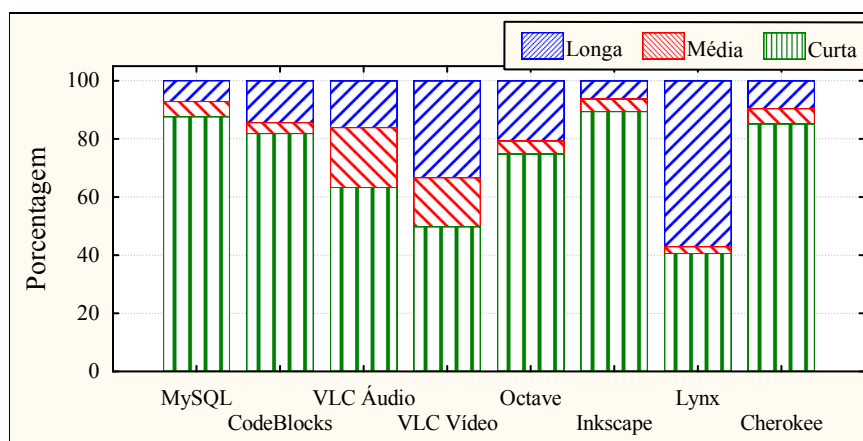


Figura 4.20. Proporção das alocações de curta, média e longa duração.

As alocações de longa duração representam em média 20,51% e constituem em sua maioria de estruturas de dados utilizadas durante a maior parte do tempo de execução das aplicações. As aplicações também foram analisadas de acordo com o seu uso das alocações de longa duração, categorizadas em dois padrões distintos *Pico* e *Uso Crescente*. De todas as aplicações analisadas, quatro se enquadraram no padrão *Uso Crescente*, a saber: CodeBlocks, Lynx, Octave e VLCPlayer. As outras quatro aplicações se enquadraram no padrão *Pico*: o VLCPlayer (áudio), MySQL, Cherokee e Inkscape.

4.8 Considerações Finais

Neste capítulo foi apresentada uma caracterização experimental do uso da alocação dinâmica de memória em sete aplicações reais. Observa-se que diferentes aplicações apresentaram padrões similares de alocação de memória: mais de 75% das alocações concentraram-se em um grupo de dez tamanhos diferentes. Além disso, todas as aplicações *Desktop* avaliadas realizaram, predominantemente, alocações menores do que 100 bytes, enquanto as duas aplicações *Servidor*, MySQL e Cherokee, apresentaram a maioria das alocações de 1144 e 264 bytes, respectivamente.

A rotina de alocação mais acionada pelas aplicações foi a *malloc*, em 87,9% dos casos, seguida da *realloc* (6,07%) e *calloc* (6,03%). Com relação ao tempo de retenção dos blocos alocados, 71,6% dos blocos foram desalocados em até cem milissegundos e 20,5% levaram mais de um segundo para serem liberados (alocações de longa duração). Dois padrões distintos de uso de memória foram observados através das alocações de longa duração: o padrão *Uso Crescente*, observado nas aplicações CodeBlocks, Lynx, Octave e VLCPlayer (vídeo), e o padrão *Pico*, observado nas aplicações VLCPlayer (áudio), MySQL, Cherokee e Inkscape.

Os resultados deste trabalho foram utilizados para configurar diferentes padrões de alocação de memória durante a geração de cargas de trabalho sintéticas, as quais foram usadas no experimento de comparação dos alocadores de memória. Além disso, os padrões encontrados no uso das alocações dinâmicas de memória podem servir para a melhoria dos algoritmos dos alocadores, visando explorar melhor as regularidades encontradas. Tal experimento serviu para investigar o comportamento dos alocadores de memória, a ser descrito no próximo capítulo.

5. ANÁLISE EXPERIMENTAL

COMPARATIVA DE ALOCADORES DE MEMÓRIA EM NÍVEL DO USUÁRIO

5.1 Introdução

Trabalhos anteriores (ex. (Matias et al., 2011) e (M. Masmano, 2006)) têm avaliado, experimentalmente, diferentes alocadores. Analisando os trabalhos nestes estudos, foi observado que seus resultados possuem uma limitação em sua aplicabilidade. Vários trabalhos se baseiam na execução de uma aplicação de teste, como aquelas usadas em *benchmarks* (ex. SPEC CPU), a fim de medir o desempenho do alocador frente a uma determinada carga de trabalho. O problema desta abordagem é a falta de controle sobre parâmetros importantes para se avaliar um alocador, tais como o *tamanho das alocações*, o *número de threads*, o *número de alocações por threads*, a ordem das alocações, entre outros. A impossibilidade de controlar esses parâmetros impede uma análise mais sofisticada dos resultados, objetivando conhecer os efeitos que mudanças nestes parâmetros acarretam no desempenho do alocador para uma dada carga de trabalho em uma dada configuração de sistema (ex. número de CPUs) (Costa et al., 2013).

Neste sentido, este capítulo apresenta a comparação de seis alocadores de memória por meio de experimentos controlados e planejados estatisticamente. A comparação dos alocadores teve como base o tempo de execução e o consumo de memória dos alocadores.

5.2 Instrumentação usada

No experimento de comparação dos alocadores foram avaliados os seguintes alocadores: Hoard (v3.8) (Berger et al., 2000), Ptmalloc2 (Gloger, 2006), Ptmalloc3 (Gloger, 2006), TCMalloc (v1.5) (Ghemawat e Menage, 2014), Jemalloc (v2.0.1) (Evans, 2006) e Miser (Tannenbaum, 2010).

Os experimentos foram conduzidos em uma bancada de teste composta de um computador *multicore* (AMD Opteron™ Processor 6212), 64 GB de RAM, com o sistema operacional Linux (Kernel 3.11.10-7-default) e distribuição openSUSE 13.1 (x86_64). A Figura 5.1 ilustra a topologia de um dos quatro *sockets* do processador usado nos testes. Cada *socket* possui oito núcleos e para cada núcleo existem quatro níveis de *cache*, com os níveis L2 e L1i compartilhados por dois núcleos e o

nível L3 compartilhado entre quatro núcleos do *socket*. Neste tipo de estudo, conhecer a topologia do processador e memória *cache* é importante, pois alguns algoritmos avaliados foram projetados para minimizar problemas de *false sharing* ao compartilhar linhas de *cache* (Berger et al., 2000). A topologia completa pode ser visualizada no Apêndice A.

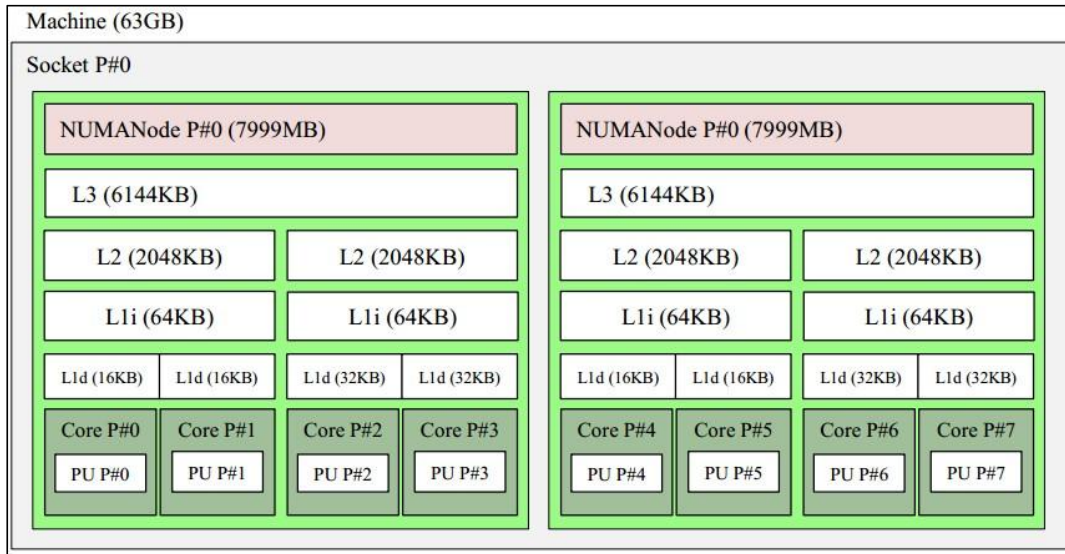


Figura 5.1. Topologia de um dos quatro *sockets* do processador utilizado nos testes.

Como o objetivo deste trabalho foi a realização de experimentos controlados, utilizou-se uma aplicação de teste, criada para esse propósito, onde foi controlado diferentes fatores relacionados com as operações de alocação.

Os fatores controlados no programa de teste foram *número de alocações* (NA), *tamanho das alocações* (TA), *número de threads* (NT) e o *padrão de distribuição das alocações de longa duração* (AL). O programa de teste foi dividido em duas etapas: preparação do experimento e a execução do experimento. Na preparação do experimento o tempo de execução não é registrado e o programa de teste gera um *script* com as informações de cada alocação, o qual é lido e executado durante a segunda etapa. O *script* definido na etapa de preparação possui duas informações para cada alocação do experimento: *tamanho da alocação* e *tempo de retenção*.

Com relação ao tempo de retenção, assim como no teste feito com o arquivo de áudio, o VLCPlayer (vídeo) manteve por mais tempo os seus blocos de memória alocados, com 51,26% das alocações sendo classificadas como média e longa duração. Uma possível explicação para o tempo de retenção alto obtido com os testes envolvendo o VLCPlayer é de que tais alocações foram feitas para um sistema de *buffer* de áudio/vídeo e que por isso demora mais tempo para serem desalocadas.

Com relação ao tempo de retenção, foi observado na caracterização de memória das aplicações que cerca de 20% das alocações foram de longa duração e tiveram o seu tempo de retenção superior a um segundo (ver Seção 4.7.5).

Normalmente, as alocações de longa duração são alocações que se mantêm alocadas durante todo o tempo de execução da aplicação. No experimento de avaliação dos alocadores este aspecto foi simplificado, visando emular os diferentes tempos de retenção sem a necessidade da medição de tempo para cada alocação, o que poderia influenciar o tempo do experimento. Cada alocação foi categorizada entre *alocação de curta duração* ou *alocação de longa duração*. No programa de teste usado nos experimentos deste capítulo, as *alocações de curta duração* foram desalocadas imediatamente após a sua alocação, enquanto *alocações de longa duração* foram desalocadas apenas no final do experimento. Por conta dessa simplificação, nos tratamentos com o tempo de execução mais curto é possível que as *alocações de longa duração* não tenham o tempo de retenção superior a 1 segundo.

A Figura 5.2 apresenta o pseudocódigo do programa utilizado no experimento de comparação dos alocadores. Na execução do experimento (linhas 15-32), cada uma das NT *threads* processam NA/NT requisições de alocação, alocando o tamanho especificado na preparação do experimento para cada alocação (linha 20). Depois de alocada, a porção de memória é preenchida com zeros (linhas 21-23). O algoritmo então verifica se a porção de memória será desalocada imediatamente (linha 24). Caso essa verificação seja verdadeira, o algoritmo desaloca a porção de memória em seguida (linha 25). Caso contrário, o algoritmo adiciona a porção de memória no vetor de alocações de longa duração (linha 27). Ao final de todas as alocações, as alocações de longa duração são liberadas (linhas 29-31). O objetivo do teste foi avaliar o desempenho de cada alocador com respeito ao tempo de execução das operações de alocação e liberação de memória, bem como o uso da memória no atendimento das requisições. Este último critério objetivou considerar o *overhead* de alocação, bem como efeitos de fragmentações interna e externa da *heap* (Randell, 1969).

Além do controle de parâmetros que afetam o comportamento do alocador sendo avaliado, os testes foram planejados para mensurar, principalmente, o trabalho realizado pelo alocador. Nosso objetivo é medir o tempo e o espaço gasto na execução das rotinas e armazenamento das estruturas de dados de cada alocador avaliado, desconsiderando qualquer uso que as aplicações possam fazer com a memória alocada, já que este uso não envolve o alocador. Em trabalhos anteriores, além das rotinas do alocador, os resultados incorporavam o tempo gasto para realizar diferentes operações de escrita e leitura na memória. Nesses casos, corre-se o risco dos tempos de uso da memória predominar sobre os de processamento das estruturas de dados do alocador, levando a conclusões incorretas. Este último é o que interessa para a comparação dos algoritmos de alocação.

Para cada alocador avaliado, executou-se a aplicação de teste instruindo o *dynamic linker* do Linux para carregar, em tempo de execução, o alocador a ser testado. Esse procedimento é realizado por meio da variável de ambiente LD_PRELOAD, a qual é configurada antes da execução da aplicação de teste para cada alocador. Desta forma, ao iniciar a aplicação utiliza-se um alocador diferente para cada teste. As métricas usadas na avaliação de desempenho dos alocadores foram o tempo total de execução e o tamanho do processo na memória principal (RSS), obtidos no final de cada execução.

Pseudocódigo: Algoritmo de Avaliação dos Alocadores

```
1  UMA_Test (ti, ta, na, nt, al)
2    ti: tamanho mínimo da alocação
3    tf: tamanho máximo da alocação
4    na: número de alocações
5    nt: número de threads
6    al: padrão das alocações de longo prazo
7    v: vetor com as instruções (size e dealloc) para cada alocação
8
9    v = Prepare_Experiment(ti, tf, na, nt, al);
10   for i in range(1..nt)
11     thread_create (Experiment(na/nt, v));
12   end for
13 end UMA_Test
14
15 Function Experiment(n, v)
16   a: vetor de blocos de memória alocados
17   d: vetor de blocos de memória de longa duração
18   cont: contador para o vetor d
19   for i in range(1..n)
20     a[i] = malloc (v[i].size);
21     for each position c in a[i]
22       a[i][c] = 0;
23     end for
24     if (v[i].dealloc == 1) then
25       free (a[i]);
26     else then
27       d[cont++] = a[i];
28     end for
29     for i in range (1..cont)
30       free(d[i]);
31     end for
32   end Function
```

Figura 5.2. Pseudocódigo da etapa de execução do experimento do programa de teste.

5.3 Planejamento Experimental

Os experimentos foram planejados com o método DOE (Montgomery, 2000), objetivando analisar seus resultados estatisticamente. Os valores adotados para os níveis dos fatores *número de alocações* (NA), *tamanho das alocações* (TA) e o *padrão das alocações de longa duração* (AL) foram baseados na caracterização das alocações dinâmicas de memória apresentada no Capítulo 4. Na caracterização de memória realizada com as aplicações foi possível observar uma variabilidade considerável do *número de alocações*. As aplicações realizaram de 30000 (Lynx e Cherokee) até 12,5 milhões de alocações (CodeBlocks e VLCPlayer vídeo). Desse modo, foi possível organizar as aplicações em cinco grupos diferentes de acordo

com o seu *número de alocações*, como mostra a Tabela 5.1. Para minimizar o tempo total do experimento, foram consideradas como nível do NA, apenas as três últimas faixas de alocação G3, G4 e G5, respectivamente arredondados para 500 mil, 1 milhão e 10 milhões de alocações.

Tabela 5.1. Aplicações agrupadas de acordo com o seu *número de alocações*.

Grupo	Número de Alocações (aproximado)	Aplicações	Nível adotado para a avaliação dos alocadores (NA)
G1	30 mil	Lynx e Cherokee	Não adotado
G2	150 mil	VLCPlayer (áudio)	Não adotado
G3	500 mil	Octave, Inkscape e MySQL	500 mil
G4	1,5 milhão	VLCPlayer (vídeo)	1 milhão
G5	12,5 milhões	CodeBlocks	10 milhões

Com base no perfil das aplicações na etapa de caracterização de memória, foram estabelecidos para o experimento de avaliação dos alocadores três níveis do fator TA: alocações pequenas (de 1 a 64 bytes), alocações médias (de 64 a 512 bytes) e alocações grandes (de 512 a 2048 bytes). Com exceção do VLCPlayer (áudio), todas as aplicações caracterizadas tiveram uma predominância das alocações definidas em apenas um nível, como mostra a Tabela 5.2. O VLCPlayer (áudio) foi a única aplicação que realizou as suas alocações de forma mais homogênea em todos os três níveis de TA: alocações pequenas (29,44%), alocações médias (28,5%) e alocações grandes (42,31%).

No estudo de caracterização descrito no Capítulo 4, foi possível estabelecer dois padrões com relação ao uso das alocações de longa duração, o padrão *Pico* e o *Uso Crescente* (ver Seção 4.5.6). Também, foi definido que 20% das alocações realizadas no experimento de comparação dos alocadores seriam de longa duração. No nível *Pico*, as alocações de longa duração foram todas realizadas nas primeiras operações de alocação, enquanto no padrão *Uso Crescente* essas alocações foram distribuídas uniformemente durante todo o experimento.

Tabela 5.2. Aplicações caracterizadas com concentração de pelo menos 70% de suas alocações em um dos níveis definidos para o TA.

Nível do TA	Intervalo do Tamanho das Alocações	Aplicações
Alocações pequenas	1..64 bytes	Octave, Lynx, Inkscape, VLCPlayer (vídeo)
Alocações médias	64..512 bytes	CodeBlocks, Cherokee
Alocações grandes	512..2048 bytes	MySQL

Neste planejamento, além dos três fatores supramencionados, outros dois fatores foram inseridos no experimento para avaliar o comportamento dos alocadores com relação ao paralelismo: o *número de threads* (NT) e o *número de processadores* (NP). Controlar estes fatores é importante, pois a maioria dos alocadores testados é projetada para explorar recursos de multiprocessamento. O controle do NT foi realizado dentro da aplicação de teste, enquanto o NP foi controlado através da chamada de sistemas *taskset* (Linux, 2014), que atribui ao processo a afinidade com determinado(s) processador(es). A escolha dos níveis do NP e NT foi realizada se baseando em trabalhos anteriores (Matias et al., 2011) (Ferreira et al., 2011a) e na experiência dos autores. A Tabela 5.3 lista todos os fatores do experimento de comparação dos alocadores e seus respectivos níveis.

Tabela 5.3. Fatores e níveis do projeto experimental.

Fatores	Níveis
<i>Número de alocações</i> (NA)	500 mil, 1 milhão, 10 milhões
<i>Tamanho das alocação</i> (TA)	16..64 bytes, 64..512 bytes, 512..2048 bytes
<i>Número de threads</i> (NT)	1, 2, 4, 8, 16, 32
<i>Número de processadores</i> (NP)	1, 2, 4, 8, 16, 32
<i>Padrão das alocações de longa duração</i> (AL)	<i>Pico, Uso Crescente</i>

Na análise dos resultados, foi utilizada a técnica de análise de variância (ANOVA) (Montgomery, 2000) para identificar os principais fatores e interações de fatores que afetam as variáveis respostas de interesse (tempo de execução e consumo de memória). Para as interações identificadas como significativas na análise de variância, realizou-se a comparação múltipla de médias pelo teste de Tukey (Montgomery, 2000). Essas comparações foram realizadas ao nível de 5% de significância (ver Apêndice B).

Neste estudo, o planejamento experimental seguiu um projeto fatorial completo (Montgomery, 2000) com fatores variando a dois e seis níveis, o que resultou em um total de 648 tratamentos por alocador. Cada tratamento é um teste para uma dada combinação de fatores e níveis. Pelo elevado número de tratamentos por alocador, adotou-se a abordagem gráfica para apresentar a diferença entre as médias dos tratamentos. Todos os gráficos apresentados neste capítulo foram gerados a partir de interações significativas entre os fatores, observados na análise de variância, com o nível de significância de 5%. Em todos os gráficos, é apresentada uma barra com o intervalo de confiança de 95% para a média.

Além da representação gráfica, o universo de 648 tratamentos por alocador teve que ser reduzido para facilitar a análise dos resultados. Visando apresentar os resultados relevantes para o contexto da aplicação prática dos alocadores, alguns fatores foram agrupados e analisados apenas em conjunto com suas interações:

NA x TA: os fatores *número de alocações* e *tamanho das alocações* foram analisados em conjunto. Estes fatores representam a carga de trabalho do alocador, uma vez que o total requisitado pela aplicação em termos de espaço de memória pode ser expresso diretamente pela multiplicação entre estes dois fatores. Nesta análise espera-se que tanto o tempo de execução quanto o consumo de memória de cada alocador sejam proporcionais aos níveis de NA e TA.

NT x NP: os fatores *número de threads* e *número de processadores* são fatores essenciais para se avaliar o desempenho dos alocadores em ambientes multiprocessados. Portanto, essa análise será apresentada em conjunto sempre que a interação for significativa para a variável resposta do alocador.

Cada tratamento foi replicado trinta vezes, descartando as duas primeiras replicações para eliminar possíveis efeitos de carga do programa, comumente presentes nas primeiras execuções. Para cada tratamento calculou-se a média aritmética dos vinte e oito valores obtidos para ambas as variáveis resposta. Entre a execução de um tratamento e outro, o sistema operacional foi reinicializado a fim de evitar a influência das condições de controle de um tratamento sobre outro. Foram executados 19.440 testes por alocador, sendo um total de 116.640 testes para todo o experimento de comparação.

5.4 Análise dos Resultados por Alocador

Nesta seção serão apresentados os resultados do experimento por alocador. Nesta primeira análise, não foi feita uma comparação entre os alocadores, contudo, os resultados individuais podem ser utilizados para compreender experimentalmente o comportamento dos alocadores frente a diferentes cenários de execução.

5.4.1 Tempo de Execução

Foram avaliados os fatores que mais influenciaram o tempo de execução do alocador, seu comportamento com o aumento da carga de trabalho (fatores NA e TA) e com relação ao aumento do *número de threads* e *número de processadores* (NT e NP). O desempenho do alocador face ao crescimento dos fatores que influenciam o paralelismo (NT e NP) evidencia a eficiência de sua estratégia na resolução do problema de contenção de memória. Vale ressaltar que o paralelismo será observado comparando o tempo de execução obtido nos testes com o acréscimo do número de *threads* e processadores.

5.4.1.1 Ptmalloc2

A análise de variância dos resultados obtidos com o Ptmalloc2 apresentou uma influência significativa dos fatores NA, TA, NP e NT no desempenho do alocador. Apenas o fator AL não influenciou o tempo de execução significativamente. Como esperado, o aumento da carga de trabalho (NA e TA) resultou em um acréscimo significativo no tempo de execução (ver Figura 5.3).

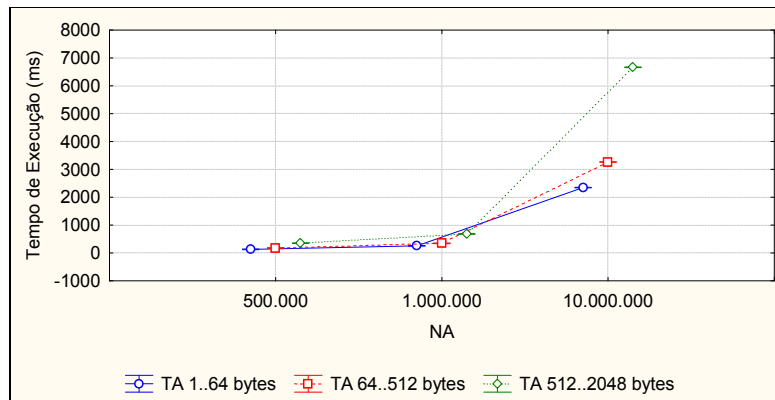


Figura 5.3. Tempo de Execução do Ptmalloc2 por fatores de carga de trabalho(NA x TA).

O alocador Ptmalloc2 apresentou queda de desempenho com o aumento nos níveis dos fatores de paralelismo, como mostra a Figura 5.4. De fato, analisando apenas os fatores NP e NT, o menor tempo obtido foi para testes com uma única *thread* (NT=1), independentemente do *número de processadores*. Ainda com relação aos ganhos do paralelismo (Figura 5.5 e Figura 5.6), analisando os quatro fatores, cujos efeitos sobre o tempo de execução foram considerados estatisticamente significativos, é possível observar que a queda no desempenho do Ptmalloc2 ocorre quando o *número de alocações* é muito grande (NA=10 milhões) e quando são alocados tamanhos acima de 64 bytes (TA=64..512 bytes, 512..2048 bytes). O Ptmalloc2, por sua vez, obteve um ganho no paralelismo nos testes feitos com tamanhos pequenos e com até quatro *threads* (TA=1..64 bytes, NT={1, 2, 4}).

O Ptmalloc2 utiliza a estratégia de *arenas* para evitar a contenção de memória em aplicações *multithreaded* (ver Seção 3.2.1). Os resultados apresentados mostraram que tal estratégia é ineficiente para a maior parte dos casos. Independente da carga da aplicação (NA e TA), na maior parte dos cenários, o pior desempenho do Ptmalloc2 ocorreu com os níveis mais altos dos fatores NT e NP (NT=32 e NP=32).

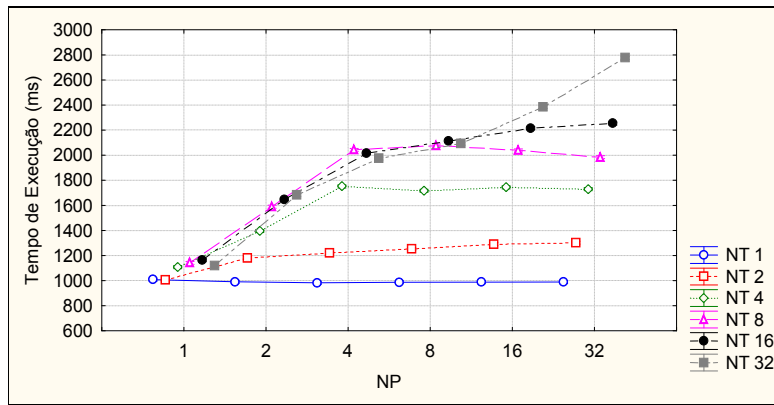


Figura 5.4. Tempo de execução do Ptmalloc2 por número de processadores e número de threads (NP x NT).

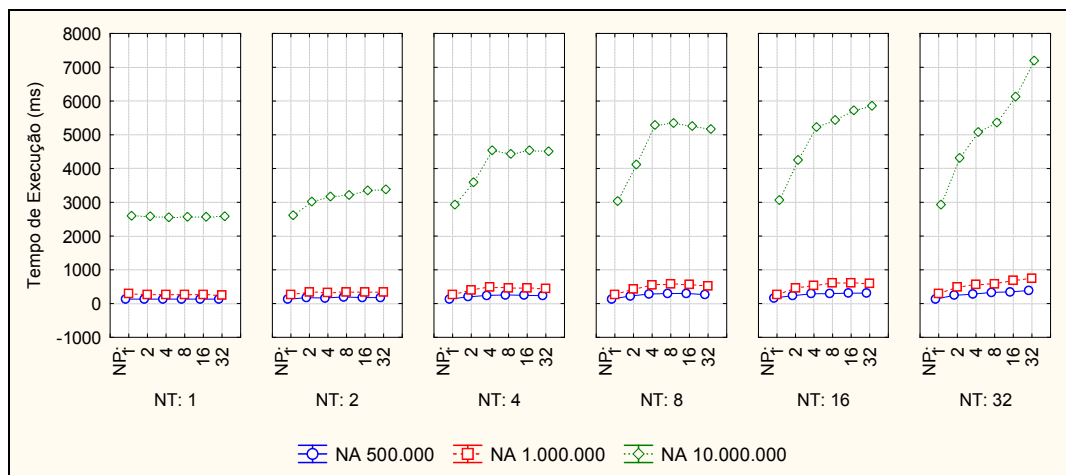


Figura 5.5. Tempo de execução do Ptmalloc2 por número de processadores, threads e número de alocações (NP x NT x NA).

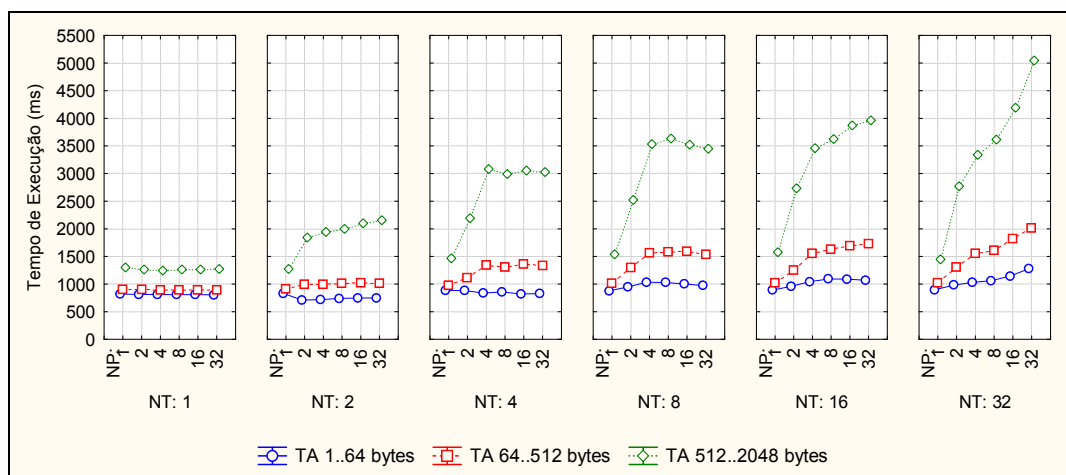


Figura 5.6. Tempo de execução do Ptmalloc2 por número de processadores, threads e tamanho das alocações (NP x NT x TA).

5.4.1.2 Ptmalloc3

Quase todos os fatores testados influenciaram significativamente o tempo de execução do Ptmalloc3. Apenas o AL não obteve uma influência significativa (ver Apêndice C). O desempenho do Ptmalloc3 foi proporcional ao nível dos fatores de carga de trabalho (NA e TA). Vale ressaltar que o Ptmalloc3 apresentou uma queda de desempenho nos testes com dez milhões de alocações (NA=10 milhões), como mostra a Figura 5.7.

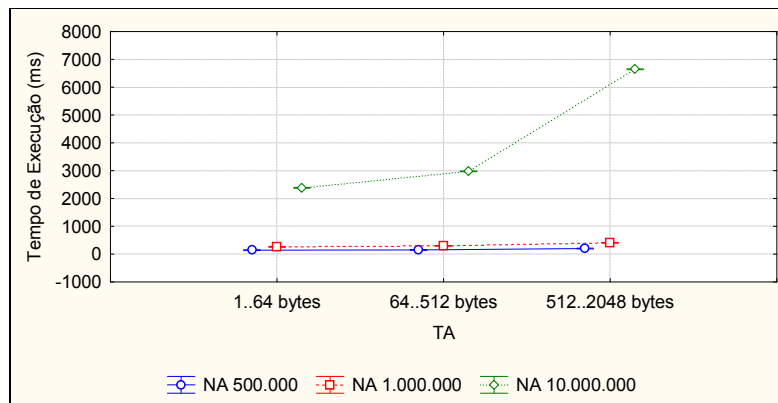


Figura 5.7. Tempo de execução do Ptmalloc3 por fatores de carga de trabalho (NA x TA).

Assim como o Ptmalloc2, o Ptmalloc3 também não obteve um bom desempenho com o aumento do nível do paralelismo nos testes (NT e NP). Este obteve uma queda de desempenho significativa nos testes com duas *threads* (NT=2). O melhor desempenho foi obtido nos testes com uma única *thread* (NT=1). Observando as Figura 5.9 e Figura 5.10, é possível constatar que essa queda de desempenho com o aumento do paralelismo se deu principalmente nos testes com a maior carga de trabalho (NA=10 milhões e TA=512..2048 bytes). Além disso, nota-se que em testes com o mesmo *número de threads*, o pior caso do alocador foi obtido com dois processadores (NP=2).

Assim como o seu antecessor, o Ptmalloc3 utiliza as *arenas* para contenção de memória. O Ptmalloc3 também obteve um ligeiro ganho de desempenho com o paralelismo quando testado com até quatro *threads* alocando apenas tamanhos pequenos (NT={1, 2, 4} e TA=1..64 bytes). Para todos os outros cenários, no entanto, esta estratégia se mostrou ineficiente para o paralelismo das aplicações em ambientes multiprocessados.

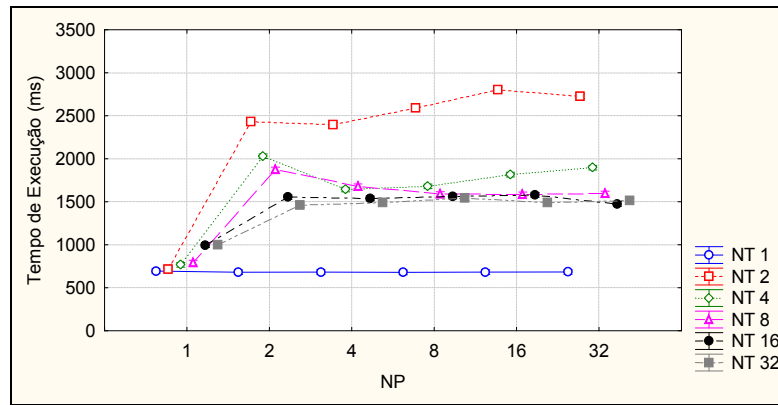


Figura 5.8. Tempo de execução do Ptmalloc3 por número de processadores e número de threads (NP x NT).

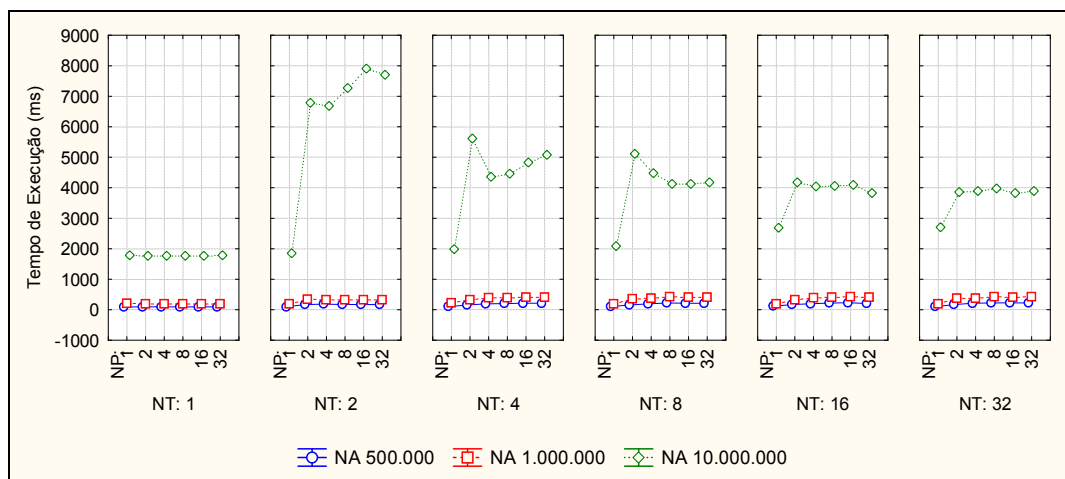


Figura 5.9. Tempo de execução do Ptmalloc3 por número de processadores, número de threads e número de alocações (NP x NT x NA).

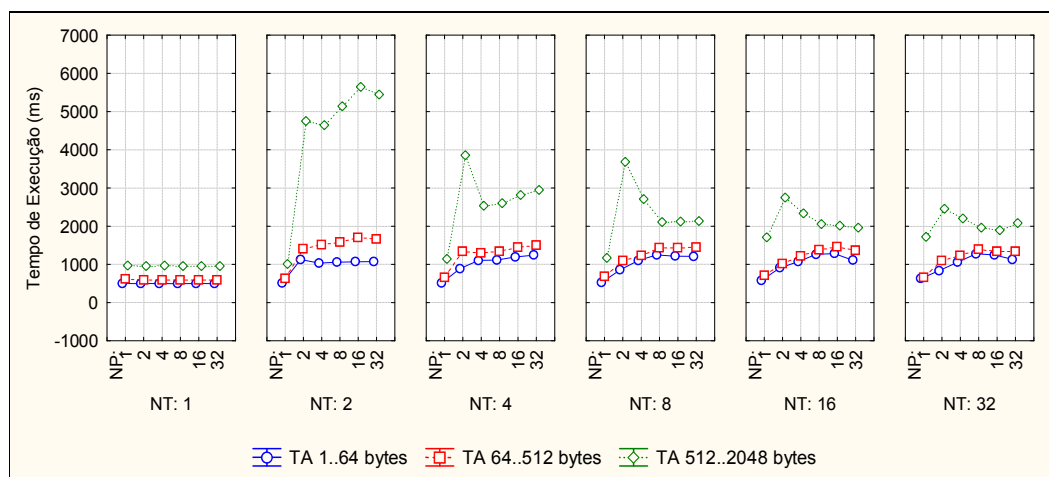


Figura 5.10. Tempo de execução do Ptmalloc3 por número de processadores, número de threads e tamanho das alocações (NP x NT x TA).

5.4.1.3 Hoard

Por meio da análise de variância foi possível observar que quase todos os fatores (NA, TA, NP e NT) influenciaram significativamente o tempo de execução do alocador Hoard. Apenas o AL não apresentou influência significativa (ver Apêndice C). Como esperado, o aumento do nível carga de trabalho (NA e TA) resultou em um acréscimo significativo no tempo de execução (Figura 5.11).

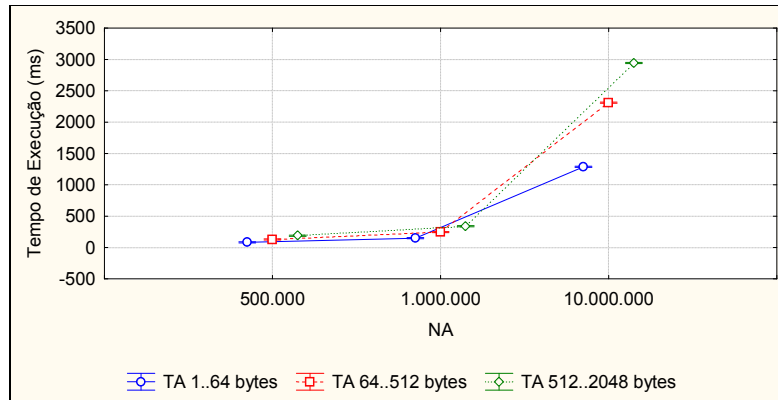


Figura 5.11. Tempo de execução do Hoard por fatores de carga de trabalho (NA x TA).

O Hoard obteve um bom desempenho com o acréscimo dos níveis dos fatores de paralelismo (ver Figura 5.12). Com a exceção dos testes feitos com mais de 16 *threads* (NT={16, 32}) o tempo de execução do Hoard caiu na medida em que mais *threads* e processadores foram adicionados no experimento. Esse ganho no desempenho, no entanto, atinge o seu limite a partir de oito processadores (NP={8, 16, 32}), onde o tempo de execução do Hoard sugere estabilizar, independente do acréscimo dos fatores NT e NP. Além disso, através dos resultados obtidos nas Figura 5.13 e Figura 5.14 é possível observar que o ganho no desempenho com o paralelismo do alocador ocorre em todos os diferentes níveis de carga de trabalho (NA e TA).

Como descrito na Seção 3.4.3, o Hoard utiliza três áreas de memória (TLAB, *Local Heap* e *Global Heap*) para minimizar a contenção das *threads*. Esta estratégia se mostrou eficiente nos resultados obtidos nesta avaliação. De forma geral, o menor tempo de execução foi obtido com 32 processadores (NP=32). No entanto, em cenários com mais de 16 *threads*, houve uma queda brusca de desempenho com dois processadores (NT={16, 32} e NP=2). Portanto, os resultados sugerem que o Hoard não seja indicado para ambientes *dual-core*. Uma das explicações possíveis para esse resultado está na política de criação das chamadas *Local Heap*. Essa estrutura é utilizada com o objetivo de reduzir a contenção de memória entre as *threads*, e o alocador cria duas *Local Heap* por processador. Com dois processadores o alocador cria apenas quatro *Local Heap*, uma quantidade baixa para uma aplicação com 16 e 32 *threads*, gerando um tempo maior de contenção.

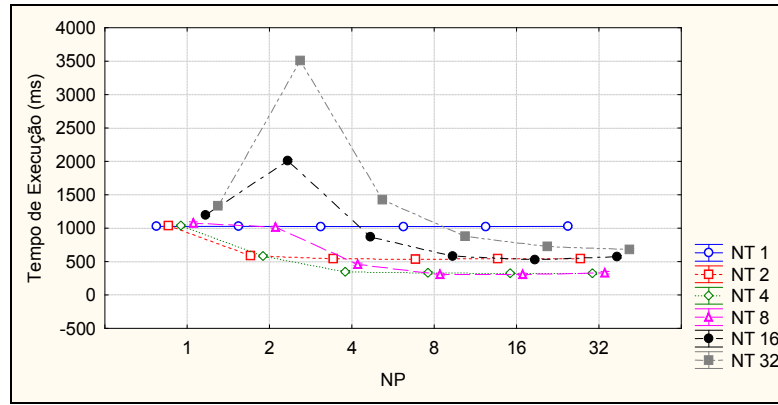


Figura 5.12. Tempo de execução do Hoard por *número de processadores* e *número de threads* (NP x NT).

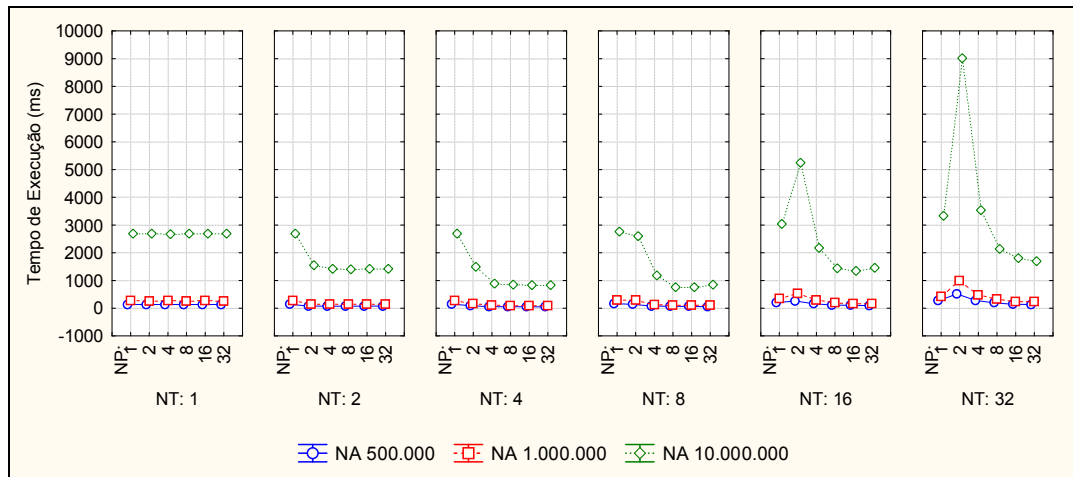


Figura 5.13. Tempo de execução do Hoard por *número de processadores*, *número de threads* e *número de alocações* (NP x NT x NA).

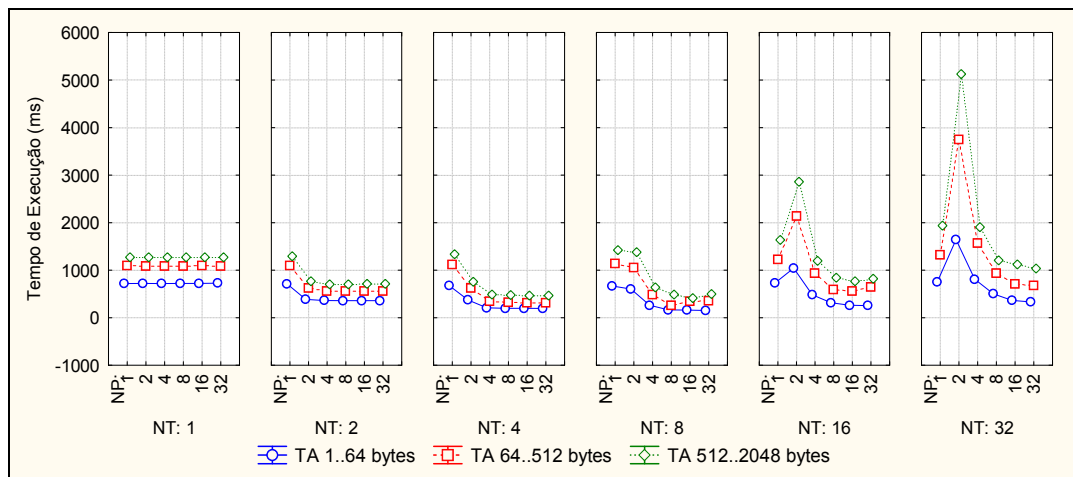


Figura 5.14. Tempo de execução do Hoard por *número de processadores*, *número de threads* e *tamanho das alocações* (NP x NT x TA).

5.4.1.4 Miser

Com exceção do AL, todos os outros fatores adotados no experimento (NA, TA, NT e NP) influenciaram significativamente o tempo de execução do alocador Miser (ver Apêndice C). Este tempo também se mostrou proporcional ao nível de carga de trabalho aplicado ao alocador, de modo que o pior desempenho foi obtido com os mais altos níveis de NA e TA (ver Figura 5.15).

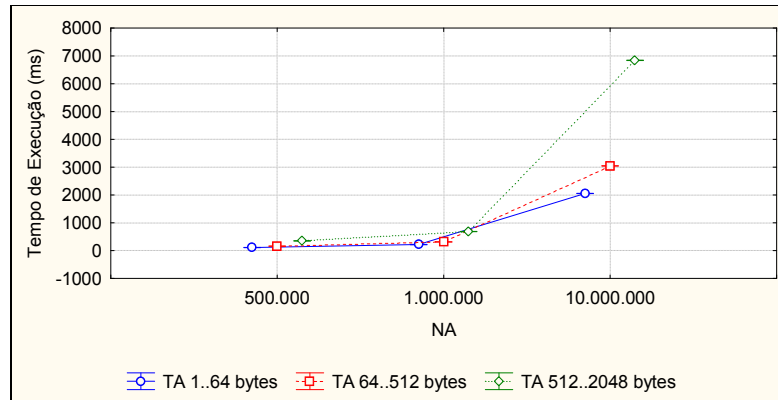


Figura 5.15. Tempo de execução do Miser por fatores de carga de trabalho (NA x TA).

O Miser não apresentou ganho de desempenho com o paralelismo nos experimentos. O seu melhor desempenho ocorreu nos experimentos com uma única *thread* (NT=1), independentemente do *número de processadores* (ver Figura 5.16). O tempo de execução do Miser cresceu com o aumento dos níveis dos fatores NT e NP. Analisando esse comportamento com relação aos outros dois fatores NA e TA (Figura 5.17 e Figura 5.18), é possível observar que o Miser perde ainda mais desempenho quando os níveis de NA e TA estão altos. Isso ocorre principalmente porque o Miser não gerencia alocações acima de 256 bytes, repassando-as para o sistema operacional. O resultado apresentado na Figura 5.18 evidencia ainda mais a diferença entre os níveis TA=1..64 bytes, completamente gerenciado pelo alocador, e o TA=512..2048 bytes repassado ao sistema operacional.

O Miser utiliza boa parte das estruturas do Hoard, com a exceção do TLAB, o *buffer* por *thread*. Isso pode explicar a queda de desempenho deste alocador com o aumento do paralelismo, uma vez que o Miser se comportou melhor com aplicações *single-threaded*.

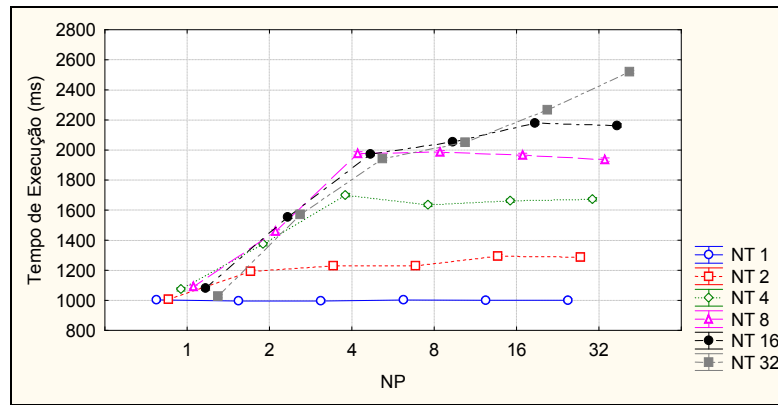


Figura 5.16. Tempo de execução do Miser por *número de processadores e número de threads* (NP x NT).

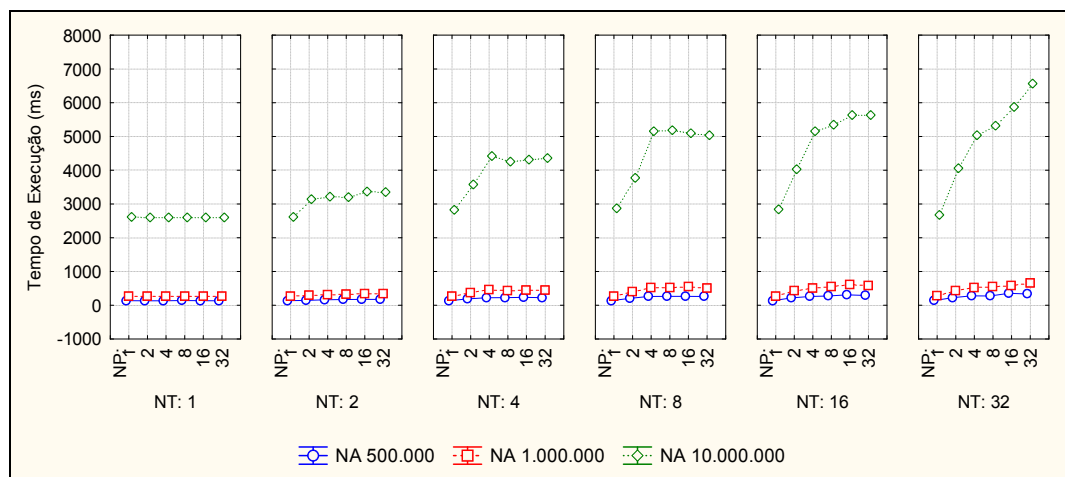


Figura 5.17. Tempo de execução do Miser por *número de processadores, número de threads e número de alocações* (NP x NT x NA).

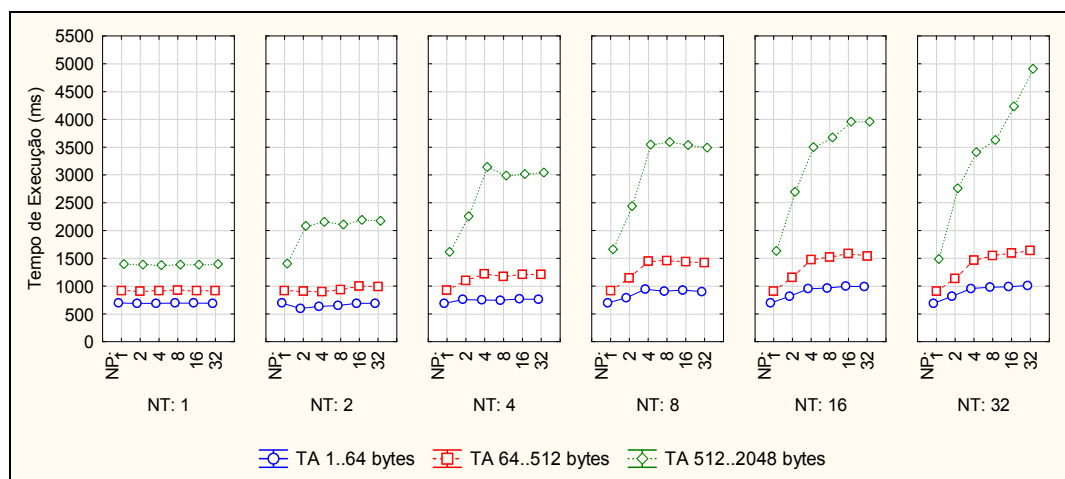


Figura 5.18. Tempo de execução do Miser por *número de processadores, número de threads e tamanho das alocações* (NP x NT x TA).

5.4.1.5 Jemalloc

Assim como nos demais alocadores, os fatores NA, TA, NT e NP influenciaram significativamente o tempo de resposta do Jemalloc. Apenas o fator AL não influenciou significativamente (ver Apêndice C). Como mostra a Figura 5.19, o tempo de resposta obtido foi proporcional ao aumento da carga de trabalho (NA e TA).

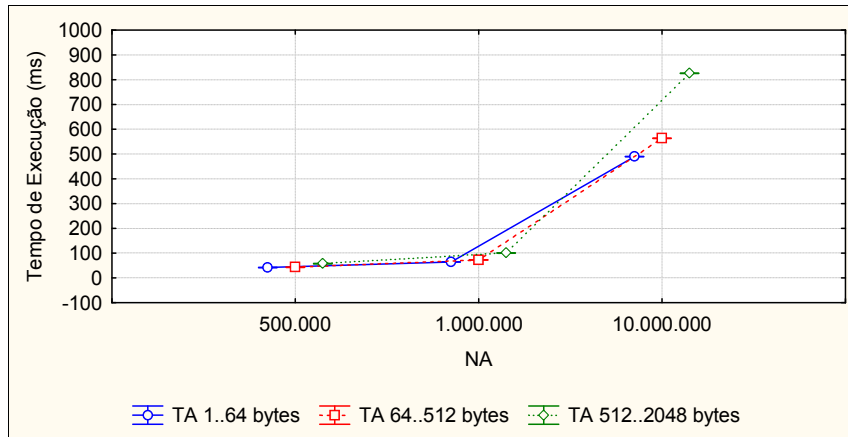


Figura 5.19. Tempo de execução do Jemalloc por fatores de carga de trabalho (NA x TA).

Diferente dos demais, o Jemalloc apresentou ganho consistente no desempenho com o aumento do paralelismo. O aumento nos níveis de *threads* e de processadores resultaram em uma redução no tempo de execução do alocador (ver Figura 5.20), o qual obteve como pior desempenho os testes realizados com uma única *thread* (NT=1) e/ou com um único processador (NP=1). É possível observar também que os testes feitos com 10 milhões de alocações obtiveram o maior ganho proporcional de desempenho com o acréscimo de *threads* e processadores (Figura 5.21). A Figura 5.22 mostra também, que independentemente do tamanho predominante da alocação da aplicação, o Jemalloc consegue aproveitar bem os múltiplos processadores em aplicações *multithreaded*.

O Jemalloc utiliza três estruturas distintas para minimizar a contenção de memória, o *Thread-Cache*, as *arenas* e a *Global Tree*. A utilização da *Thread-Cache* em conjunto com as *arenas* se mostrou eficiente em todos os cenários avaliados neste experimento, obtendo um ganho de desempenho com o aumento dos fatores de paralelismo. Os testes, no entanto, não cobriram o acesso à *Global Tree* uma vez que não foram alocados tamanhos maiores do que 4MB.

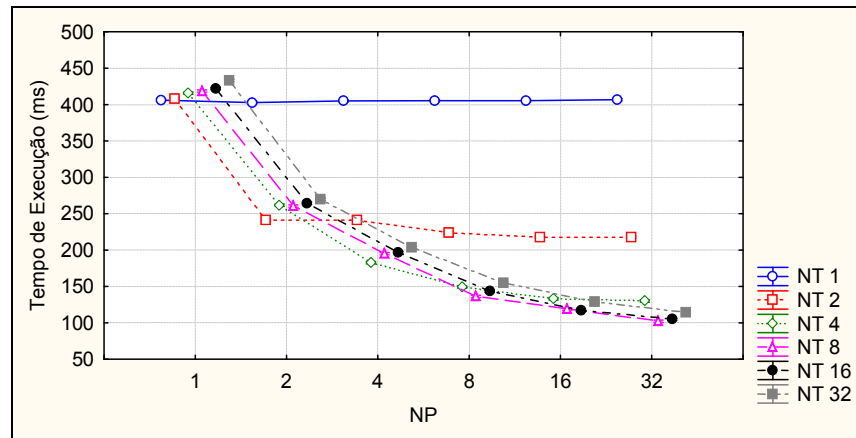


Figura 5.20. Tempo de execução do Jemalloc por número de processadores e número de threads (NP x NT).

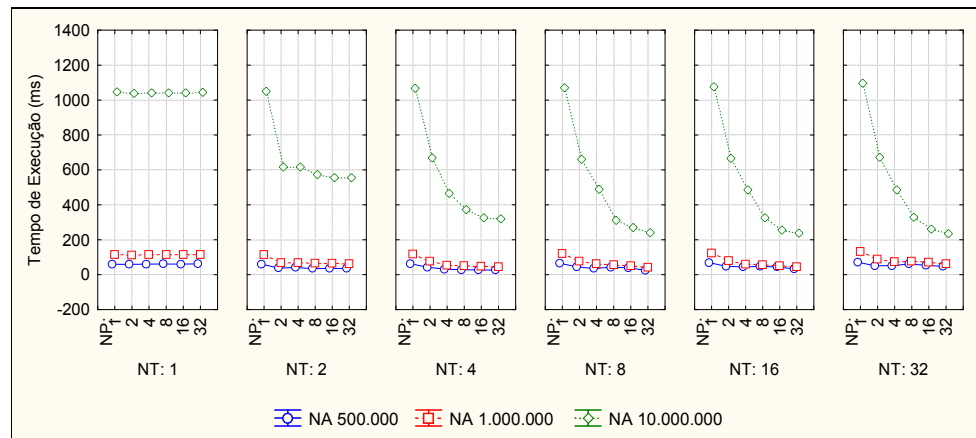


Figura 5.21. Tempo de execução do Jemalloc por número de processadores, número de threads e número de alocações (NP x NT x NA).

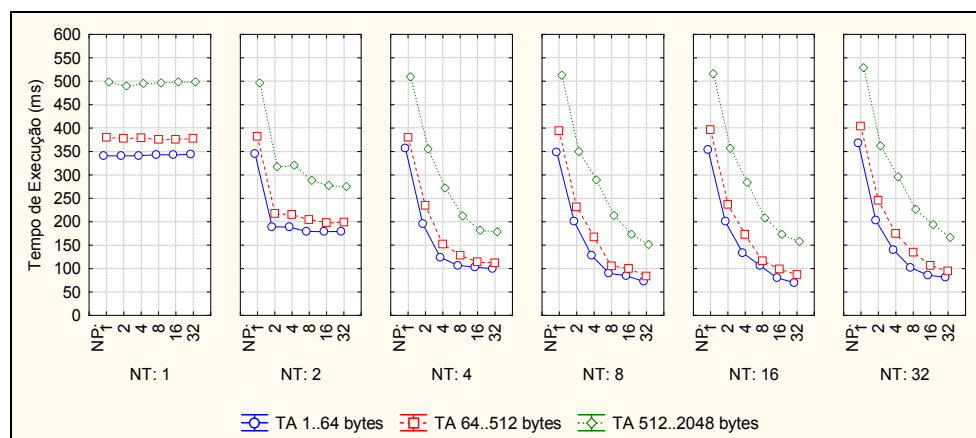


Figura 5.22. Tempo de execução do Jemalloc por número de processadores, número de threads e número de alocações (NP x NT x NA).

5.4.1.6 TCMalloc

Os fatores NA, TA, NP e NT influenciaram significativamente o tempo de resposta do alocador TCMalloc. Apenas o AL não obteve uma influência significativa no tempo de execução do alocador (ver Apêndice C). O aumento da carga do trabalho também influenciou o tempo de resposta do alocador (ver Figura 5.23), de forma que o maior tempo de execução foi obtido nos maiores níveis das variáveis NA e TA (NA=10 milhões e TA=512..2048 bytes).

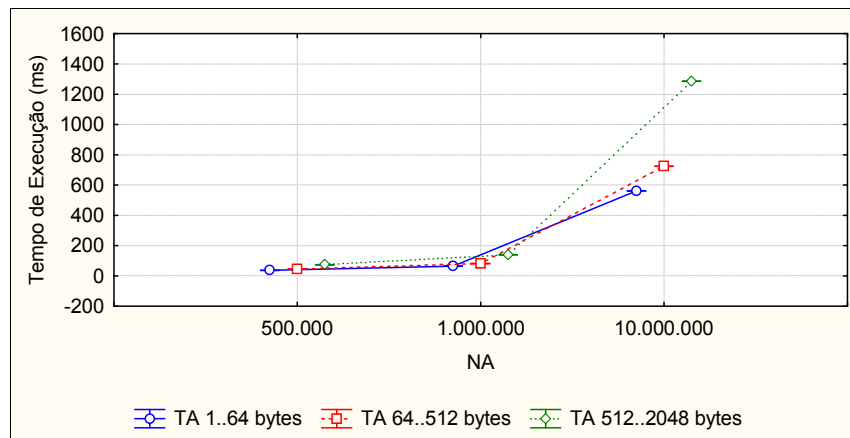


Figura 5.23. Tempo de execução do TCMalloc por fatores de carga de trabalho (NA x TA).

Assim como o Jemalloc, o TCMalloc mostrou um ganho no desempenho com relação ao paralelismo nos testes com múltiplas *threads* e múltiplos processadores (ver Figura 5.24). De forma geral, o pior tempo de execução do alocador foi obtido em cenários *single threaded* (NT=1) e/ou em cenários com um único processador (NP=1). No entanto, é possível notar um limite na escalabilidade do TCMalloc. Testes com mais de oito *threads* começam a perder o desempenho quando combinados em ambientes com mais de 8 processadores (NT={16, 32} e NP={16, 32}).

O TCMalloc utiliza dois níveis de memória para minimizar a contenção das *threads*: o *Thread-Cache* e o *Global-Cache*. Os resultados neste experimento apontaram que esta estratégia é eficiente em aproveitar os benefícios do paralelismo. O ganho do desempenho pôde ser encontrado nos testes com diferentes níveis de NA e TA (ver Figura 5.25 e Figura 5.26). Isso mostra que independente do perfil de alocação da aplicação, o alocador ganha em performance quando em um ambiente multiprocessado e sendo usado em uma aplicação com múltiplas *threads*.

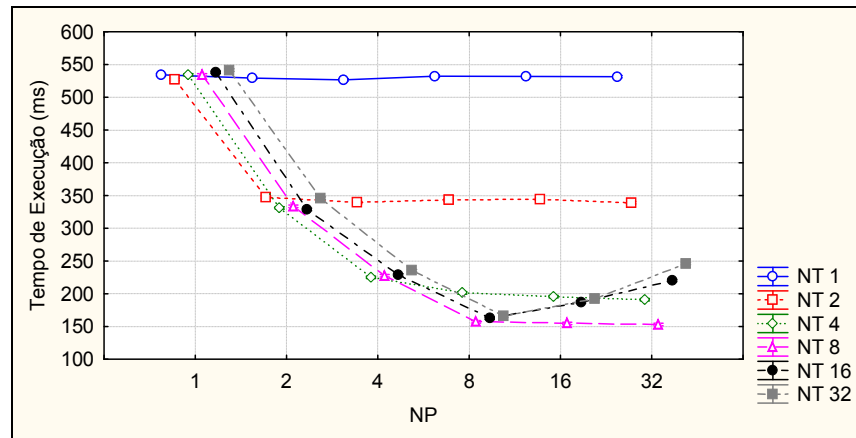


Figura 5.24. Tempo de execução do TCMalloc por número de processadores e número de threads (NP x NT).

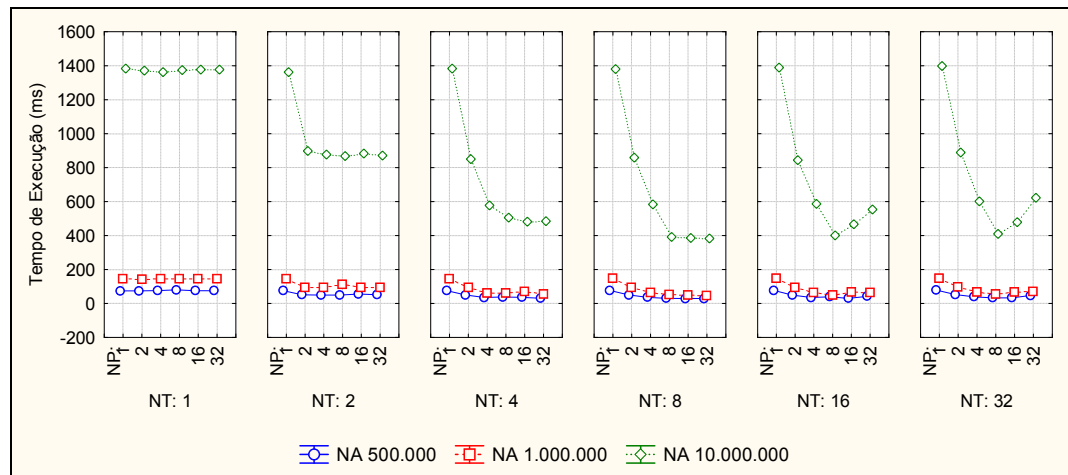


Figura 5.25. Tempo de execução do TCMalloc por número de processadores, número de threads e número de alocações (NP x NT x NA).

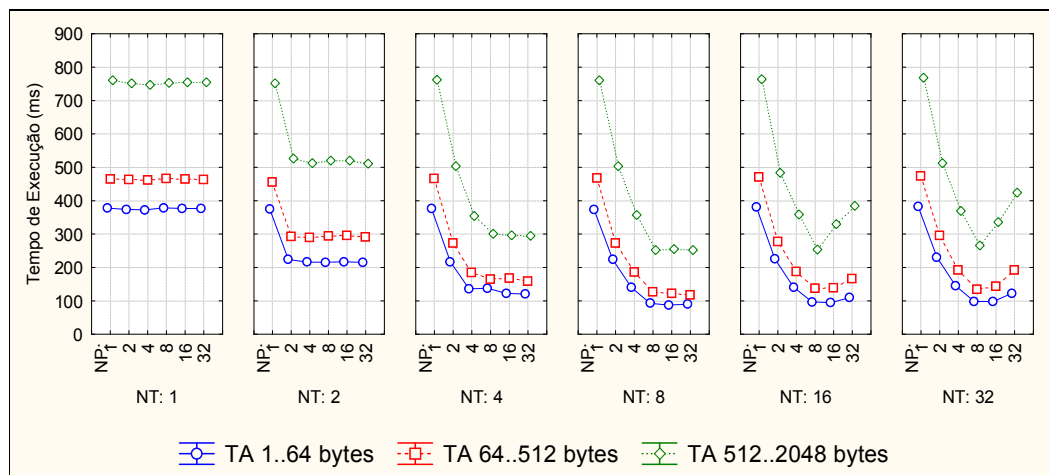


Figura 5.26. Tempo de execução do TCMalloc por número de processadores, número de threads e tamanho das alocações (NP x NT x TA).

5.4.2 Consumo de Memória

Nesta seção serão apresentados os resultados do experimento de avaliação dos alocadores com relação ao consumo de memória. Serão discutidos os fatores que mais influenciaram o consumo de memória dos alocadores, bem como o seu comportamento com o aumento da carga de trabalho (fatores NA e TA) e o consumo de memória relativo aos padrões de alocação de longa duração (AL).

5.4.2.1 Ptmalloc2

Com exceção do *número de processadores*, todos os outros fatores influenciaram significativamente no consumo de memória (ver Apêndice D). Como esperado, o consumo de memória do Ptmalloc2 foi proporcional à sua carga de trabalho (ver Figura 5.27). Além disso, o alocador obteve o maior consumo de memória nos testes feitos com o padrão *Uso Crescente*. Interessante notar que o Ptmalloc2 apresentou o maior consumo de memória nos testes com uma única *thread* (ver Figura 5.28). Esse comportamento contradiz o esperado pela análise dos algoritmos de alocação (ver Seção 3.2), uma vez que o alocador cria múltiplas *arenas* para evitar a contenção nos testes com múltiplas *threads*. O melhor cenário de consumo de memória do Ptmalloc2 ocorreu com a menor carga de trabalho (NA=500 mil e TA=1..16 bytes), com oito *threads* e com as alocações de longa duração no padrão *Pico* (ver Figura 5.29 e Figura 5.30).

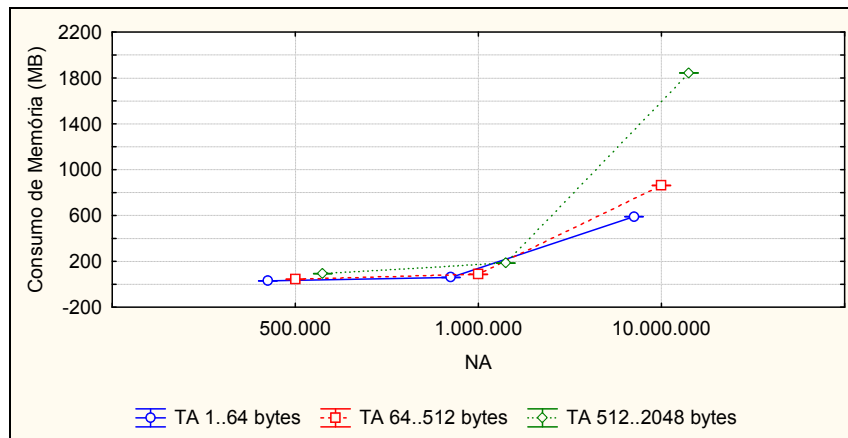


Figura 5.27. Consumo de memória do Ptmalloc2 por *número de alocações* e *tamanhos da alocações* (NA x TA).

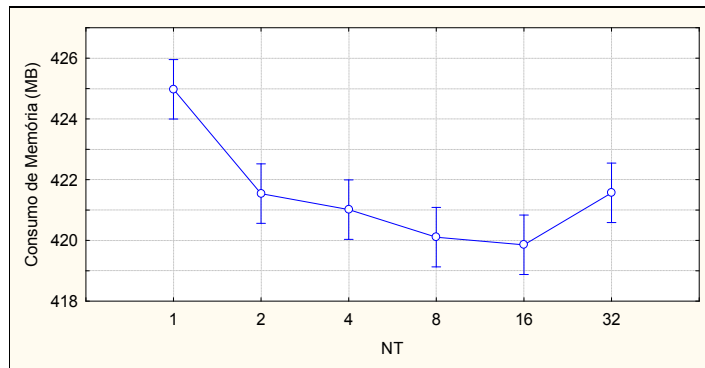


Figura 5.28. Consumo de memória do Ptmalloc2 por número de threads (NT).

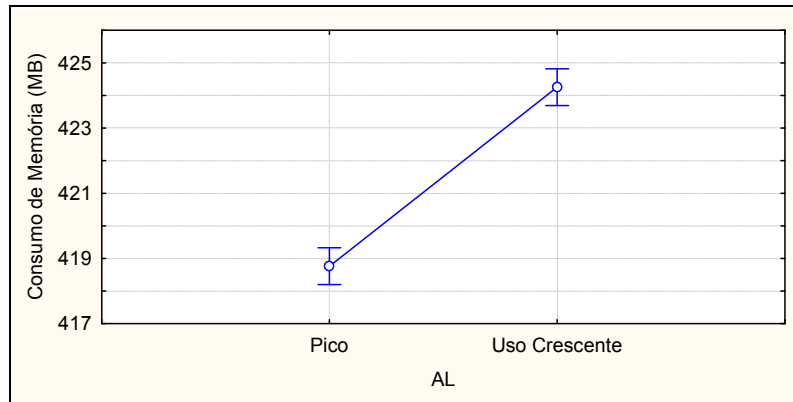


Figura 5.29. Consumo de memória do Ptmalloc2 por padrão de alocações de longa duração (AL).

5.4.2.2 Ptmalloc3

Apenas o *número de processadores* não influenciou significativamente o consumo de memória do alocador Ptmalloc3 (ver Apêndice D). O consumo de memória desse alocador foi proporcional aos níveis dos fatores de carga de trabalho (NA e TA) como mostra a Figura 5.30.

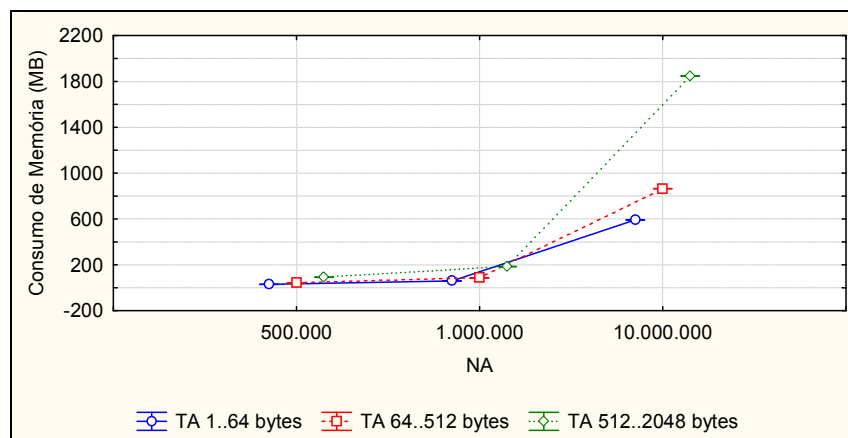


Figura 5.30. Consumo de memória do Ptmalloc3 por número de alocações e tamanhos da alocações (NA x TA).

Analisando o fator AL separadamente, é possível constatar que o Ptmalloc3 obteve um menor consumo de memória com o padrão *Pico* (ver Figura 5.31), enquanto com relação ao *número de threads* o menor consumo foi obtido com menos de 32 *threads* (NT={1,2,4,8,16}), que apresentam valores estatisticamente semelhantes (ver Figura 5.32). O pior consumo foi obtido com 32 *threads* no padrão *Uso Crescente* (NT=32 e AL=*Uso Crescente*).

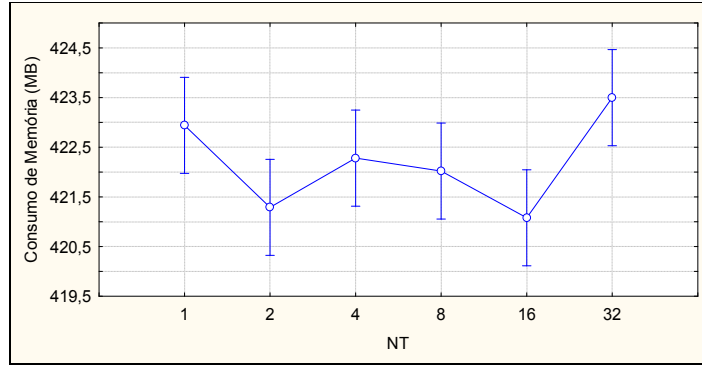


Figura 5.31. Consumo de memória do Ptmalloc3 por *número de threads* (NT).

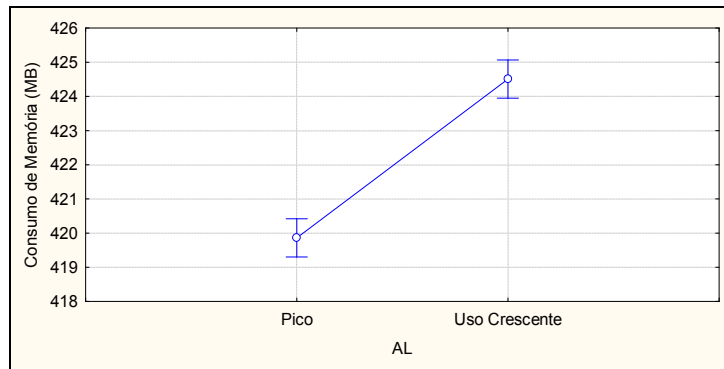


Figura 5.32. Consumo de memória do Ptmalloc3 por *padrão das alocações de longa duração* (AL).

5.4.2.3 Hoard

Diferentemente dos outros alocadores, no Hoard todos os fatores testados influenciaram o consumo de memória do alocador, incluindo o *número de processadores* (ver Apêndice D). O Hoard teve o seu consumo de memória proporcional aos níveis da carga de trabalho (NA e TA), como mostra a Figura 5.33. Como todos os fatores influenciaram significativamente o alocador, neste caso foi realizada a análise do consumo de memória em relação ao *número de processadores* (Figura 5.34). É possível notar que o consumo de memória foi maior para 32 núcleos e menor para um ambiente *dual-core* (NP=2). Este resultado pode ser observado na Figura 5.34.

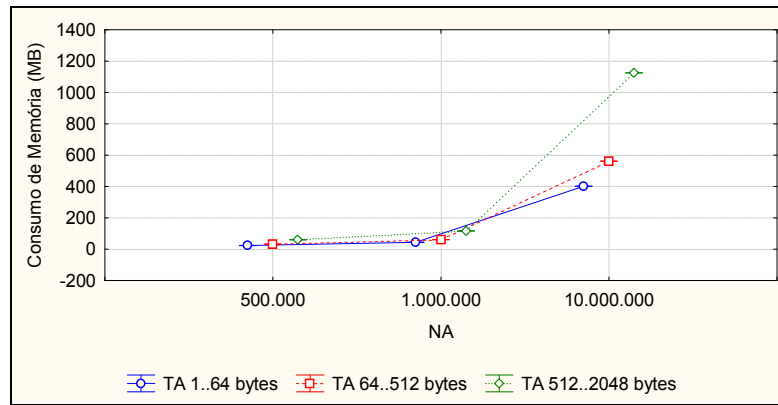


Figura 5.33. Consumo de memória do Hoard por número de alocações e tamanhos das alocações (NA x TA).

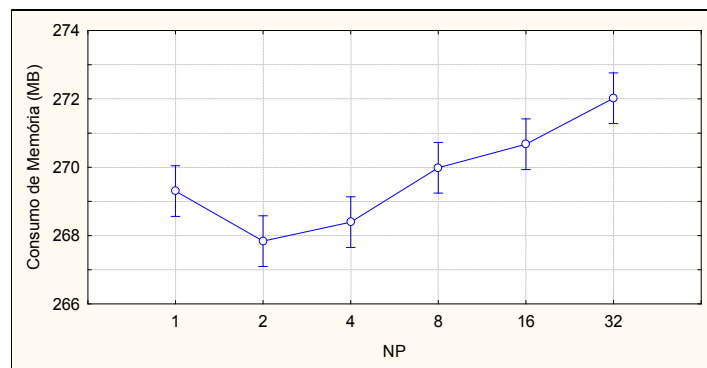


Figura 5.34. Consumo de memória do Hoard por número de processadores (NP).

Observa-se na Figura 5.35 que o Hoard gradativamente reduz o seu consumo de memória quando os testes são realizados com até quatro *threads*, aumentando o seu consumo a partir de 8 *threads*. Com relação ao fator AL, o padrão de alocação *Pico* apresentou um nível de consumo médio 4,5 vezes menor do que o nível *Usa Crescente*, uma diferença mais expressiva do que a encontrada nos demais alocadores investigados (ver Figura 5.36). Este resultado mostra que o Hoard utiliza mais eficientemente a memória para aplicações que realizam suas alocações de longa duração no primeiro conjunto de alocações.

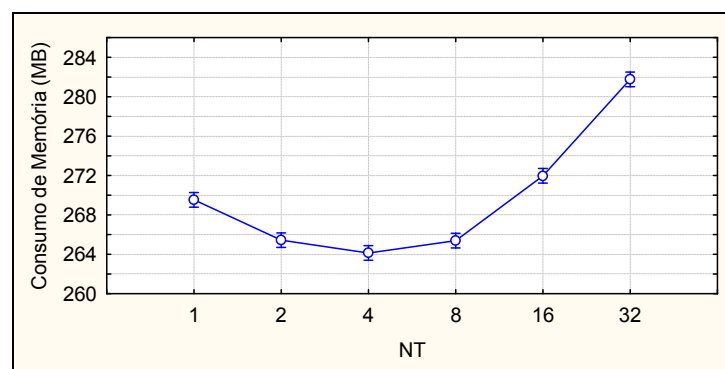


Figura 5.35. Consumo de memória do Hoard por número de threads (NT).

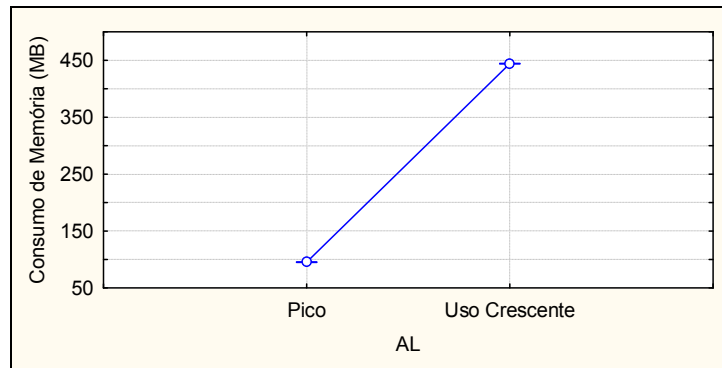


Figura 5.36. Consumo de memória do Hoard por padrão das alocações de longa duração (AL).

5.4.2.4 Miser

Seguindo o padrão encontrado nos alocadores já analisados, apenas o *número de processadores* não influenciou significativamente o consumo de memória do alocador (ver Apêndice D). É possível observar na Figura 5.37 que o consumo de memória do Miser foi proporcional aos níveis dos fatores de carga de trabalho (NA e TA).

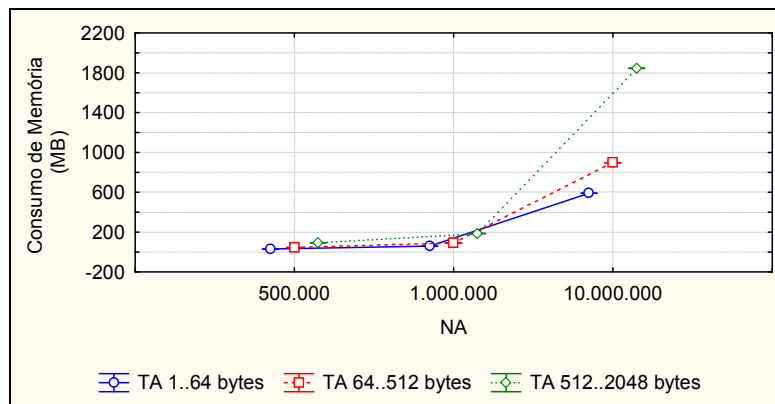


Figura 5.37. Consumo de memória do Miser por número de alocações e tamanhos da alocações (NA x TA).

O consumo de memória do Miser decai com o acréscimo do *número de threads* (ver Figura 5.38), de modo que o alocador teve o seu maior consumo nos testes feitos com uma *thread* (NT=1) e o menor consumo com quatro *threads* (NT=4). Com relação ao padrão das alocações de longo prazo, o padrão *Pico* apresentou o menor consumo de memória (ver Figura 5.39).

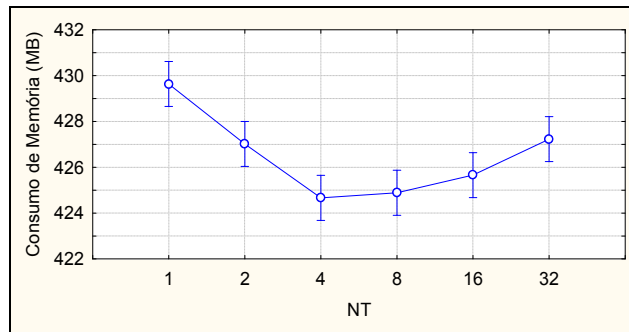


Figura 5.38. Consumo de memória do Miser por número de threads (NT).

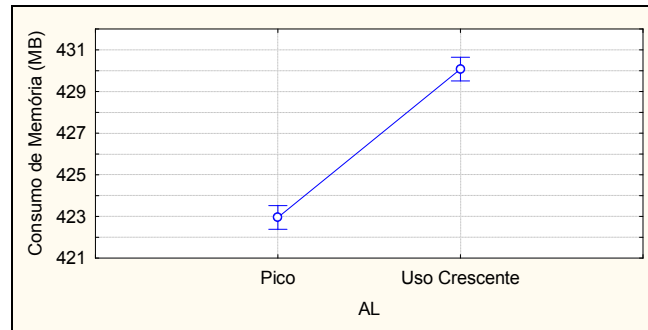


Figura 5.39. Consumo de memória do Miser por padrão das alocações de longa duração (AL).

5.4.2.5 Jemalloc

O Jemalloc teve o seu consumo de memória influenciado significativamente por todos os fatores com exceção do número de processadores (ver Apêndice D). A Figura 5.40 apresenta o consumo de memória do Jemalloc em relação aos fatores de carga de trabalho (NA x NT).

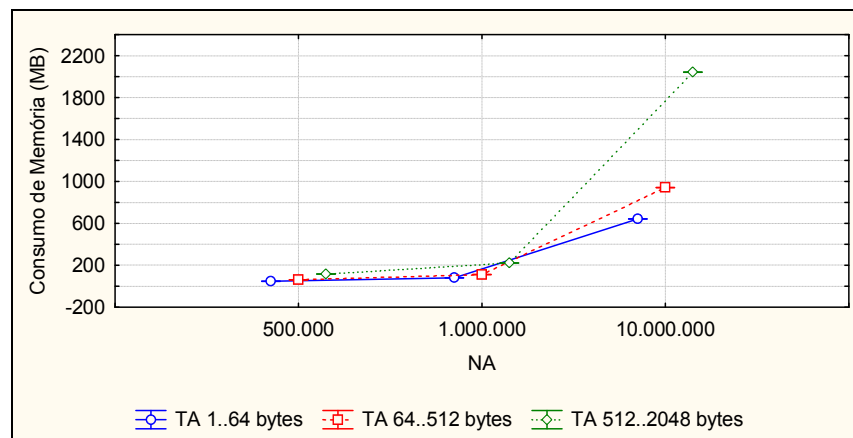


Figura 5.40. Consumo de memória do Jemalloc por número de alocações e tamanhos das alocações (NA x TA).

Com base na Figura 5.421, é possível notar que o menor consumo de memória do Jemalloc foi obtido com duas threads (NT=2) e o maior consumo com 32 threads (NT=32). Esse alocador obteve um padrão de consumo diferente do

resultado obtido pelos demais alocadores na mudança de nível do AL. O padrão *Uso Crescente* apresentou um consumo de memória menor do que o *Pico* (ver Figura 5.432).

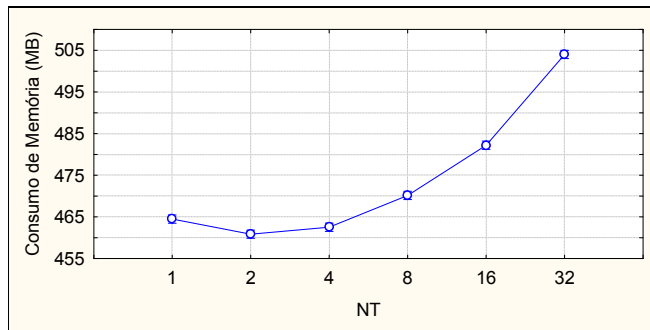


Figura 5.41. Consumo de memória do Jemalloc por *número de threads* (NT).

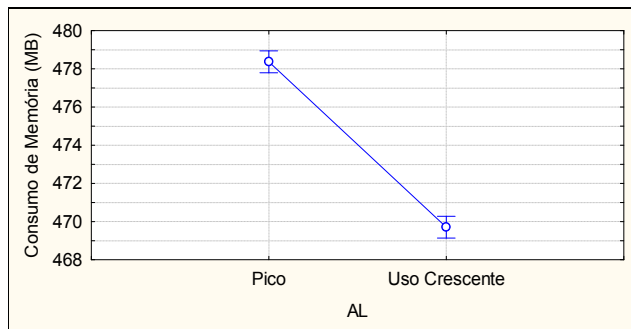


Figura 5.42 Consumo de memória do Jemalloc por *padrão das alocações de longa duração* (AL).

5.4.2.6 TCMalloc

O TCMalloc teve o seu consumo de memória influenciado significativamente por quase todos os fatores combinados nos testes. Apenas o *número de processadores* não influenciou significativamente o seu consumo de memória (ver Apêndice D). A Figura 5.43 apresenta o consumo de memória do alocador com respeito aos níveis adotados nos fatores de carga de trabalho (NA e TA). Note que o consumo de memória do alocador foi proporcional ao tamanho e ao *número de alocações* do teste.

Analisando apenas o fator NT, o menor consumo de memória do alocador foi obtido com 16 e 32 *threads* (Figura 5.44). Esse resultado, atípico, mostra que o TCMalloc tende a utilizar menos memória em uma aplicação com múltiplas *threads*. Além disso, semelhante ao Jemalloc, o TCMalloc apresentou um menor consumo com o padrão de alocações de longa duração *Uso Crescente* (veja Figura 5.45).

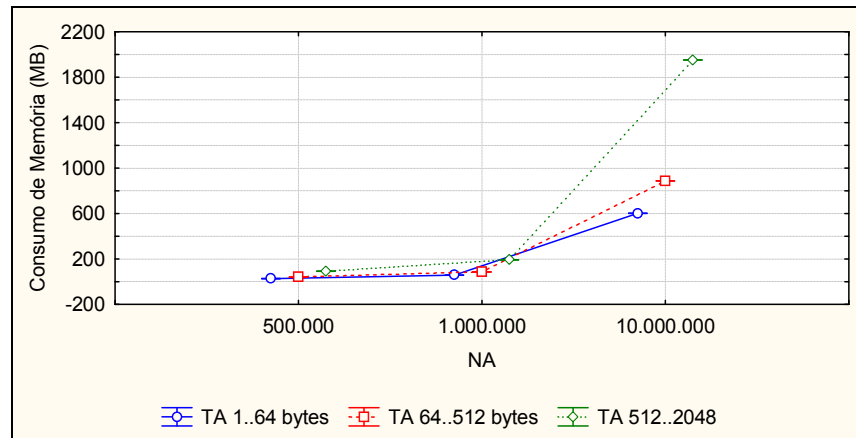


Figura 5.43. Consumo de memória do TCMalloc por *número de alocações* e *tamanhos das alocações* (NA x TA).

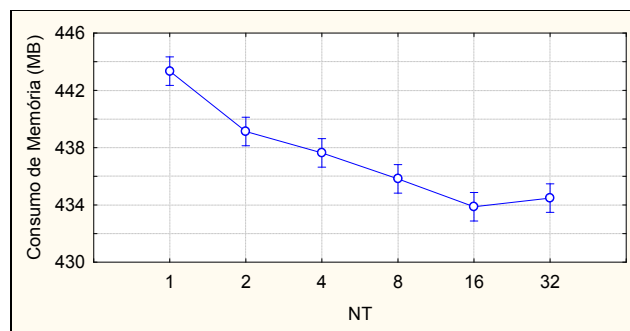


Figura 5.44. Consumo de memória do TCMalloc por *número de threads* (NT).

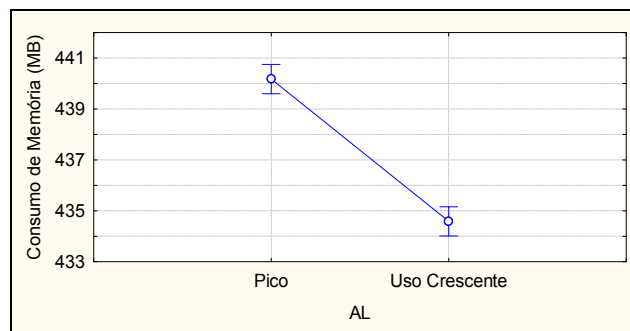


Figura 5.45. Consumo de memória do TCMalloc por *padrão das alocações de longa duração* (AL).

5.5 Análise Comparativa dos Alocadores

Nesta seção será apresentada uma análise comparativa dos resultados obtidos pelos alocadores. Será estabelecido um *ranking* dos alocadores em termos de tempo de execução e consumo de memória. O resultado desta análise pode ser utilizado para a escolha de um alocador mais adequado para determinado perfil de operação da aplicação, no tocante às operações de alocação dinâmica de memória.

5.5.1 Tempo de Execução

De forma geral, o alocador que obteve o menor tempo médio de execução, em todos os experimentos, foi o Jemalloc (Figura 5.46). O TCMalloc também obteve um bom desempenho na média e foi o segundo alocador mais rápido, seguido do Hoard, Ptmalloc3, Miser e Ptmalloc2. Vale ressaltar que o alocador padrão da *glibc*, Ptmalloc2, obteve o pior desempenho. Com a média dos tempos de execução de cada alocador, é possível dividir os alocadores em três grupos distintos de acordo com uma aproximação de seu desempenho: o grupo dos alocadores mais rápidos composto por Jemalloc e TCMalloc, o segundo grupo com o Hoard que apresentou um desempenho mediano e o terceiro e mais lento grupo dos alocadores, composto pelo Ptmalloc2, Ptmalloc3 e Miser.

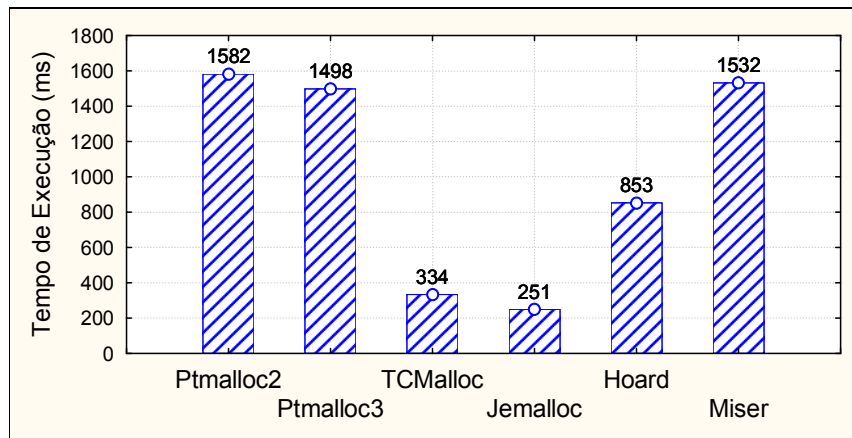


Figura 5.46. Tempo de execução médio por alocador de memória.

Também é importante analisar o desempenho dos alocadores relativo ao fator *número de processadores* (NP). Esta análise pode ser usada para decidir previamente qual alocador é mais indicado para um ambiente de execução específico, quando não se tem conhecimento do perfil de alocação de uma aplicação. Considerando os resultados dessa análise, sumarizados na Figura 5.47, é possível concluir que o Jemalloc se comporta melhor em todos os ambientes de execução, seguido de perto pelo TCMalloc. O Ptmalloc2 apresentou o pior desempenho em todos os cenários, com exceção dos ambientes *dual-core* (NP=2), onde o seu sucessor (Ptmalloc3) apresentou o maior tempo de execução (pior desempenho). Todos os alocadores que obtiveram ganhos de desempenho com mais de um processador implementam uma estrutura de memória exclusiva, por thread, a *Thread-Cache*. Essa estrutura possibilita à *thread* alocar e desalocar a memória mais rapidamente, uma vez que não há a necessidade de mecanismos de contenção. Essa pode ser uma das explicações da superioridade do desempenho dos alocadores Jemalloc, TCMalloc e Hoard, com o acréscimo do número de processadores.

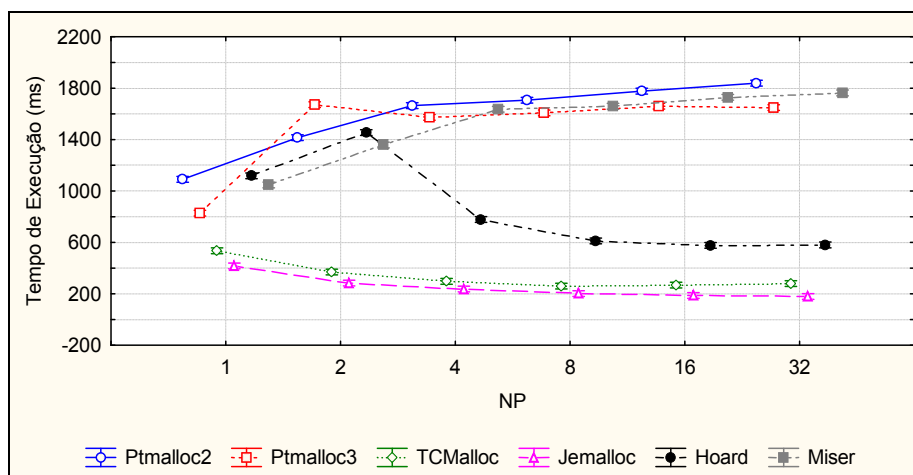


Figura 5.47. Tempo de execução médio por alocador agrupado pelo número de processadores (NP).

Analisando os fatores de paralelismo em conjunto (NP x NT), é possível observar os mesmos resultados da análise feita apenas com o fator NP. O grupo dos alocadores mais rápidos, compostos pelo Jemalloc e TCMalloc foram superiores em todas as combinações NP x NT executadas neste experimento (ver Figura 5.48). O Hoard não se comportou bem com ambientes *dual-core* com mais de 8 *threads* (NP=2 e NT={16, 32}), mas apresentou desempenho mediano nos outros casos. Um resultado que deve ser destacado é o comportamento do Ptmalloc3 quando utilizado com duas *threads* (NT=2), o qual obteve um tempo de execução muito acima dos outros alocadores. Neste experimento, os alocadores que apresentaram um ganho de performance com o aumento do paralelismo foram também os alocadores com a menor média de tempo de execução.

Analisando o tempo de resposta dos alocadores pelo fator NA é possível observar que a diferença de desempenho entre os alocadores cresce quando se aumenta o número de alocações do experimento. Os alocadores Jemalloc e TCMalloc mantêm um desempenho muito acima dos outros alocadores nos testes com 10 milhões de alocações (NA=10.000.000). O Jemalloc chega a ser em média 6,3 vezes mais rápido do que alocadores como o Ptmalloc2, Ptmalloc3 e Miser no nível mais alto do fator NA.

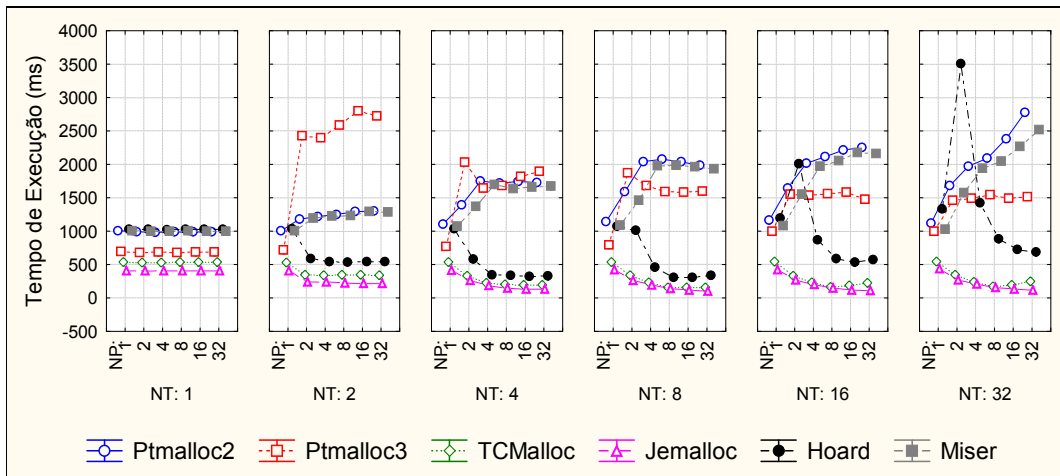


Figura 5.48. Tempo de execução médio por alocador agrupado pelo número de processadores e número de threads (NP x NT).

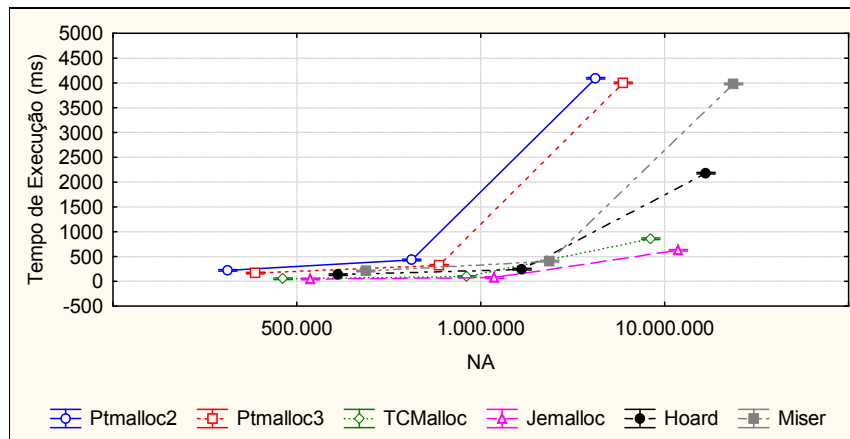


Figura 5.49. Tempo de execução médio por alocador agrupado pelo número de alocações (NA).

Analisando a Figura 5.49 é possível comparar o tempo de resposta dos alocadores apenas em relação aos *tamanhos das alocações* (TA). Os resultados observados anteriormente se repetem, ou seja, o Jemalloc e o TCMalloc apresentam um desempenho superior nos três níveis de tamanhos de alocação, com uma diferença no tempo de execução especialmente maior nos níveis mais altos (TA={64..512 bytes, 512..2048 bytes}). Vale ressaltar que tanto o Jemalloc quanto o TCMalloc tratam alocações de até 2048 bytes como alocações pequenas. Dessa forma, durante todo o experimento, as alocações realizadas acessaram uma mesma estrutura de dados desses alocadores. O mesmo não é válido para os alocadores Ptmalloc2, Ptmalloc3 e Miser, os alocadores mais lentos nos testes feitos com TA=512..2048 bytes. Para estes alocadores, as requisições acima de 256 bytes são consideradas grandes e acessam uma estrutura de busca mais lenta, que privilegia a economia de memória. Esse pode ter sido um fator decisivo para a baixa performance nos testes feitos com o nível mais alto dos *tamanhos das alocações* (TA) apresentado na Figura 5.50.

Com base nesses resultados é possível estabelecer um *ranking* com respeito ao tempo de execução dos alocadores. A Tabela 5.4 apresenta um ranking geral dos alocadores, estabelecido através do tempo médio de execução dos alocadores em todo o experimento. Além do ranking, a tabela apresenta a diferença de tempo de execução médio de cada alocador comparado ao tempo do alocador mais rápido, o Jemalloc. O TCMalloc, por exemplo, foi 33% mais lento do que o alocador mais rápido no experimento.

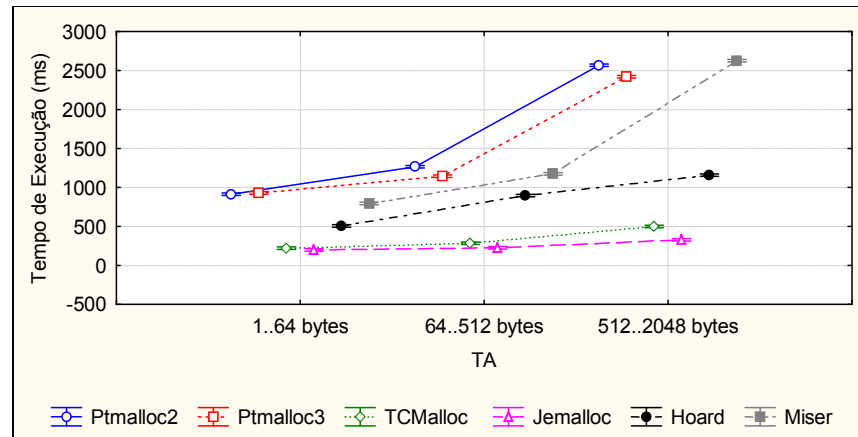


Figura 5.50. Tempo de execução médio por alocador agrupado pelo *tamanho das alocações (TA)*.

Tabela 5.4. Ranking dos alocadores pelo tempo médio de execução.

Ranking	Alocador	Comparação do tempo de execução com o alocador mais rápido ^a
#1	Jemalloc	-
#2	TCMalloc	+33,06%
#3	Hoard	+239,8%
#4	Ptmalloc3	+496,8%
#5	Miser	+510,4%
#6	Ptmalloc2	+530,7%

^a O cálculo foi feito com base na diferença entre a média geral do tempo de execução de cada alocador, dividido pelo tempo médio de execução do Jemalloc.

5.5.2 Consumo de Memória

Analisando o consumo de memória médio dos alocadores, em todos os cenários de avaliados, pode-se observar que o Hoard apresentou, destacadamente, o menor consumo de memória. Na Figura 5.51 é possível observar que o Hoard obteve uma economia considerável de memória, com cerca de 40% de memória

economizada comparada à média dos outros alocadores. O segundo alocador com o menor consumo foi o Ptmalloc2, seguido pelo Ptmalloc3, Miser, TCMalloc e por último o Jemalloc, o qual apresentou o maior consumo de memória entre os alocadores avaliados.

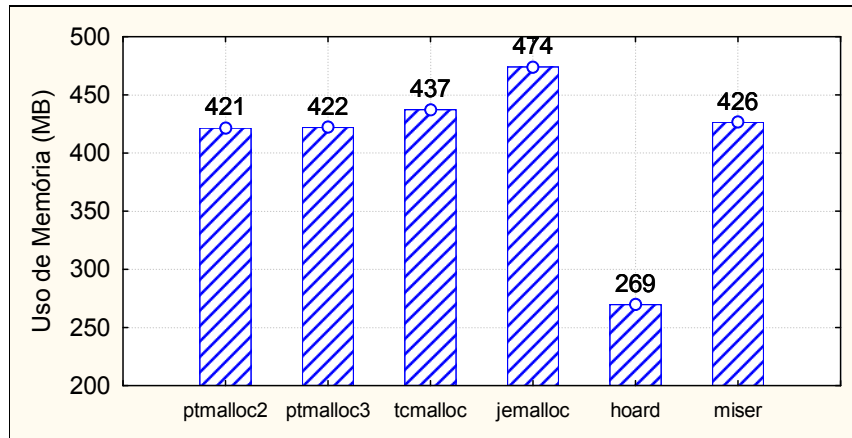


Figura 5.51. Consumo médio de memória por alocador.

Analisando o consumo de memória dos alocadores obtidos nos diferentes níveis do fator AL, é possível observar que o alocador Hoard economiza muita memória no padrão *Pico* (ver Figura 5.52). Nesse padrão, as alocações de longa duração são realizadas apenas no início do experimento, e no restante do teste a memória é alocada e desalocada imediatamente.

Diferentemente da análise do tempo de execução, apresentada na Seção 5.4.1, onde os alocadores que obtiveram a menor média da variável de resposta foram superiores em todos os níveis dos fatores, o alocador Hoard apresentou um desempenho muito superior em apenas um dos níveis do AL. Portanto, a análise comparativa do consumo de memória contemplou dois casos, um para cada nível do fator AL. Dessa forma foi possível compreender melhor a diferença no consumo de memória entre os alocadores nos diferentes cenários testados.

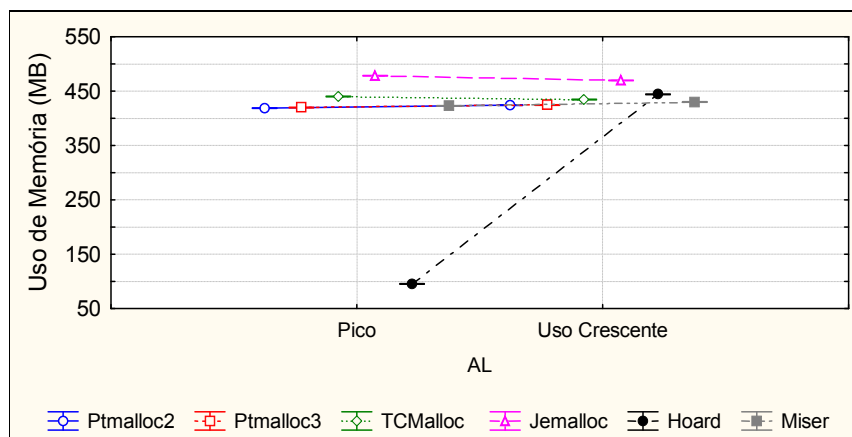


Figura 5.52. Consumo médio de memória dos alocadores por padrão de alocações de longa duração.

As Figura 5.53 e Figura 5.54 apresentam os resultados com relação aos fatores de carga de trabalho, respectivamente NA e TA. Em ambos os casos, no nível $AL=Pico$, o Hoard se sobressai em relação aos demais alocadores, especialmente quando o teste é realizado com um nível alto de NA e TA. Os alocadores, Ptmalloc2, Ptmalloc3 e Miser possuem um comportamento muito semelhante em todos os cenários avaliados nesta análise. O TCMalloc e o Jemalloc foram os alocadores que mais consumiram memória. Considerando apenas os testes onde o $AL=Uso\ Crescente$, é possível notar que os alocadores Ptmalloc2 e Ptmalloc3 obtiveram o menor consumo de memória em todos os cenários, seguidos pelo Miser e TCMalloc. Nestes cenários o Hoard obteve a quinta melhor posição em termos de consumo de memória, na frente apenas do Jemalloc.

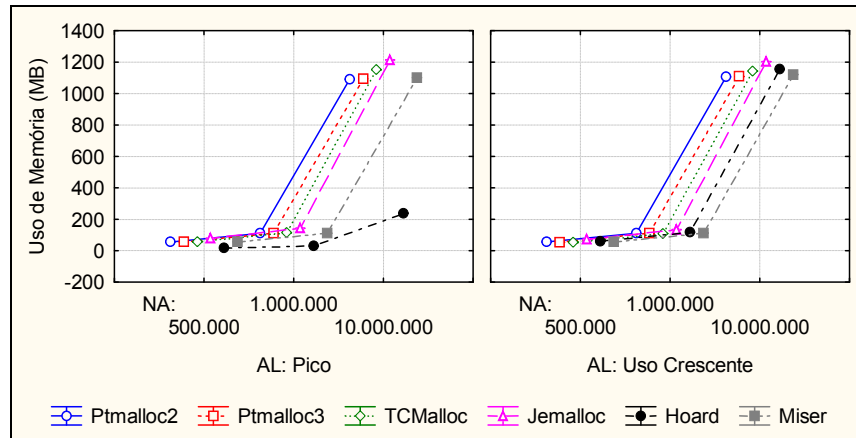


Figura 5.53. Consumo de memória dos alocadores por combinação dos fatores NA e AL.

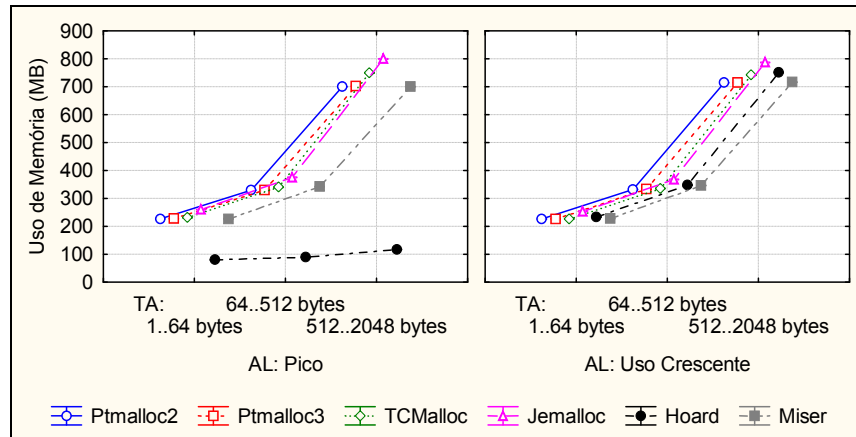


Figura 5.54. Consumo de memória dos alocadores por combinação dos fatores TA e AL.

De forma geral, o *número de processadores* não influenciou no consumo de memória dos alocadores, por essa razão nesta seção não será feita a análise do consumo de memória com respeito ao fator *número de processadores* (NP).

Por meio dos resultados observados neste estudo experimental, é possível estabelecer dois *rankings* distintos de alocadores com relação ao seu consumo de

memória. Embora o Hoard tenha obtido o menor consumo de memória geral (Figura 5.51), a sua economia de memória ocorreu apenas em metade dos cenários de testes avaliados, aqueles com o fator $AL=Pico$. Cenários onde o $AL=Uso Crescente$ apresentaram um ranking completamente diferente, o que deve ser levado em consideração. As Tabelas 5.5 e 5.6 apresentam os dois rankings, um para cada nível dos padrões das alocações de longa duração. Assim como o ranking estabelecido no tempo de execução, estes rankings apresentam também uma coluna com a comparação entre o espaço usado do alocador e o espaço usado do respectivo número #1 do ranking em questão.

Tabela 5.5. Ranking dos alocadores pelo consumo médio de memória nos testes feitos com o padrão $AL=Pico$.

Ranking	Alocador	Comparação do consumo de memória com o alocador mais econômico
#1	Hoard	-
#2	Ptmalloc2 / Ptmalloc3	+340%
#4	Miser	+344%
#5	TCMalloc	+363%
#6	Jemalloc	+403%

A diferença entre o consumo de memória dos alocadores Ptmalloc2 e Ptmalloc3 não foi significativa estatisticamente, ao nível de significância de 5%.

Tabela 5.6. Ranking dos alocadores pelo consumo médio de memória nos testes feitos com o padrão $AL=Uso Crescente$.

Ranking	Alocador	Comparação do consumo de memória com o alocador mais econômico
#1	Ptmalloc2 / Ptmalloc3	-
#3	Miser	+1,41%
#4	TCMalloc	+2,35%
#5	Hoard	+4,71%
#6	Jemalloc	+10,6%

A diferença entre o consumo de memória dos alocadores Ptmalloc2 e Ptmalloc3 não foi significativa estatisticamente, ao nível de significância de 5%.

5.6 Considerações Finais

Neste capítulo foi apresentada uma análise experimental de seis alocadores de memória. Cada alocador foi avaliado em termos de tempo de execução e consumo de memória, por um total de 648 tratamentos, 30 replicações e com seis

diferentes fatores variando de dois a seis níveis. No total, o experimento executou 19440 testes por alocador.

De forma geral, dos fatores avaliados apenas o *padrão das alocações de longa duração* (AL) não apresentou uma influência significativa no tempo de execução dos alocadores. Todos os alocadores apresentaram um tempo de execução proporcional aos níveis dos fatores de carga de trabalho do experimento (NA e TA). Com relação a ganhos com o paralelismo, apenas três dos seis alocadores obtiveram um tempo menor de execução quando testados em ambientes multiprocessados e com múltiplas *threads* os quais foram: Jemalloc, TCMalloc e Hoard. O Jemalloc foi o alocador com o melhor desempenho em todos os cenários considerados, seguido pelo TCMalloc, Hoard, Ptmalloc3, Miser e Ptmalloc2. Neste experimento, os alocadores que obtiveram a menor média no tempo de execução foram também os melhores em aproveitar o paralelismo dos cenários com mais de uma *thread* e um processador. Vale ressaltar que o alocador Jemalloc foi, em média, 5,3 vezes mais rápido do que o alocador padrão da *glibc*, o qual obteve o pior desempenho geral no experimento realizado.

Com relação ao consumo de memória, o *número de processadores* foi o único fator que não apresentou uma influência significativa. O consumo de memória de todos os alocadores foi maior nos altos níveis de *número de alocações* e *tamanho das alocações* (NA e TA). O Hoard foi o alocador que obteve o menor consumo médio de memória neste estudo, contudo, esta economia se deu apenas nos cenários com *AL=Pico*. Nestes casos, o Hoard apresentou uma economia de memória substancial se comparado aos demais alocadores avaliados, chegando a utilizar 76,2% de memória a menos do que o Jemalloc. Em cenários onde o *padrão das alocações de longa duração* seguiu o *Uso Crescente* (*AL=Uso Crescente*), o resultado foi diferente. Nestes, os alocadores com o menor consumo de memória foram Ptmalloc2 e Ptmalloc3, seguidos pelo Miser, TCMalloc, Hoard e Jemalloc. A Tabela 5.7 sumariza esses resultados apresentando os rankings dos alocadores com relação ao tempo de execução e consumo de memória.

Tabela 5.7. Ranking dos alocadores pelo tempo de execução e consumo de memória.

Ranking	Tempo de Execução	Consumo de Memória (AL=Pico)	Consumo de Memória (AL=Uso Crescente)
#1	Jemalloc	Hoard	Ptmalloc2 / Ptmalloc3
#2	TCMalloc	Ptmalloc2 / Ptmalloc3	Miser
#3	Hoard	Miser	TCMalloc
#4	Ptmalloc3	TCMalloc	Hoard
#5	Miser	Jemalloc	Jemalloc
#6	Ptmalloc2		

Os resultados deste estudo mostram que os alocadores avaliados como os mais rápidos também foram os que utilizaram mais memória. Se comparado aos demais alocadores, o Jemalloc e o TCMalloc obtiveram um desempenho médio cerca de 4,2 vezes superior à média dos demais alocadores e utilizaram cerca de 8% a mais de memória. O alocador Hoard obteve o menor consumo médio de memória e adicionalmente obteve ganhos com o paralelismo ficando no segundo grupo dos alocadores mais rápidos. O Ptmalloc3, Miser e Ptmalloc2 apresentaram um comportamento semelhante em todos os cenários; eles não obtiveram ganhos de desempenho com o aumento do paralelismo, obtiveram o mais alto tempo de execução médio e foram o grupo dos alocadores com o menor consumo de memória nos casos onde a alocação de longo prazo é feita durante toda a execução do experimento (*AL=Uso Crescente*).

6. CONCLUSÕES

6.1 Principais Resultados

Em engenharia de sistemas computacionais, o gerenciamento de memória principal tem um impacto importante no desempenho e escalabilidade de sistemas de software (Ferreira et al., 2011a). Novos alocadores vêm sendo desenvolvidos visando melhor explorar perfis específicos de aplicações e ambientes de execução (Areias e Rocha, 2012) (Benosman et al., 2013) (Cho et al., 2014). Contudo, a transparência na usabilidade e a complexidade na avaliação experimental do alocador de memória para um dado perfil de uso da memória, apresentam-se como barreiras na escolha apropriada do alocador a ser usado no projeto de uma aplicação.

Visando contribuir para a solução deste problema, este estudo apresentou uma avaliação experimental de um conjunto de seis alocadores de memória amplamente usados atualmente. As cargas de trabalho usadas na avaliação experimental dos alocadores foram planejadas com base em um estudo de caracterização de uso da memória em diferentes tipos de aplicação. Os resultados obtidos neste trabalho enfatizam a importância da escolha do alocador no projeto das aplicações, face às diferenças significativas observadas experimentalmente nos experimentos com cada alocador em conjunto com a aplicação de teste.

6.2 Limitações da Pesquisa

Neste trabalho, algumas restrições limitaram a expansão da aplicabilidade da pesquisa, conforme descrito a seguir.

No Capítulo 4 foi apresentada a caracterização das alocações dinâmicas de memória de sete diferentes aplicações, em oito cenários de uso no total. O número de aplicações testadas foi limitado pelas condições de escolha da caracterização (Seção 4.3), especialmente o uso do alocador padrão e a automatização dos testes. Este número de aplicações pode ser considerado pequeno e tal estudo pode ser expandido para abrigar um número maior de categorias diferentes de aplicações.

Nem todos os resultados experimentais puderam ser justificados, com alto grau de confiança, pela análise das estruturas e estratégias implementadas pelos algoritmos dos alocadores, detalhados no Capítulo 3. Isso porque na análise dos algoritmos foram considerados principalmente aspectos de alto nível do alocador, como política de alocação e principais estruturas de dados. É possível que alguns mecanismos desconsiderados nesta análise, assim como técnicas e boas práticas de

otimização de algoritmos, tenham influenciado de forma mais relevante o resultado em nível experimental.

Embora se tenha testado um considerável conjunto de cenários na avaliação dos alocadores, os resultados obtidos no Capítulo 4 se aplicam apenas aos cenários que possuam fatores e níveis compatíveis com os usados nos tratamentos deste estudo. Não é possível afirmar se os resultados obtidos neste estudo seriam mantidos em condições diferentes, portanto, é exigida sempre a comparação dos cenários avaliados com os esperados para uso da aplicação, a fim de se considerar o uso dos resultados obtidos nesse trabalho.

Neste sentido, os resultados obtidos neste estudo podem ser usados como guia, especialmente quanto ao comportamento observado na maior parte dos 678 cenários avaliados, uma vez que eles contemplam seis alocadores de memória largamente usados e condições de uso (cargas de trabalho) típicas de diferentes categorias de aplicações de uso atualmente.

6.3 Contribuições para a Literatura

A Tabela 6.1 apresenta os trabalhos publicados com resultados parciais obtidos no decorrer desta pesquisa.

Tabela 6.1. Publicações Científicas.

Conferência	Qualis	Formato	Citação
SBESC 2013: Simpósio Brasileiro de Engenharia de Sistemas Computacionais.	B4	Artigo Completo	(Costa et al., 2013)
ACM SAC 2014: The 29th ACM/SIGAPP Symposium On Applied Computing	A1	Pôster	(Costa et al., 2014)
SBESC 2014: Simpósio Brasileiro de Engenharia de Sistemas Computacionais	B4	Artigo Completo	(Costa e Matias, 2014)

Além disso, têm-se a previsão para a produção de um artigo para um periódico, o qual será submetido em 2015 e conterà todo o trabalho apresentado nessa dissertação, com ênfase nos resultados obtidos na comparação dos alocadores.

6.4 Dificuldades Encontradas

As principais dificuldades encontradas durante essa pesquisa estão descritas a seguir:

Escolher um conjunto relevante de aplicações a serem caracterizadas e automatizar cada cenário de teste, uma vez que ao se trabalhar com um conjunto de aplicações heterogêneas, têm-se uma abordagem distinta para cada aplicação.

Analisar um conjunto muito grande de dados na avaliação experimental dos alocadores, pois ao se analisar cada tratamento de forma detalhada acaba-se por estabelecer relações muito complexas. Um grande desafio foi o estabelecimento de um equilíbrio entre o nível de detalhamento da análise com a generalização dos resultados obtidos.

6.5 Trabalhos Futuros

O estudo experimental desse trabalho pode ser expandido para a obtenção de novos resultados. A etapa da caracterização das alocações de memória dinâmica pode ser ampliada com novas categorias de aplicações e cenários de uso, a fim de ampliar os resultados de caracterização do consumo de memória.

Recomenda-se também uma expansão do experimento de comparação dos alocadores, a fim de se contemplar novos algoritmos de alocação, os quais vêm sendo desenvolvidos recentemente. Com isso, novas estratégias de gerenciamento de memória podem ser avaliadas, ampliando as opções de escolha de alocador pelo usuário, além de oferecer para a comunidade de pesquisa desta área um conjunto maior de estudos comparativos.

Finalmente, é importante destacar que este estudo faz parte de um projeto maior, o qual prevê o projeto e desenvolvimento de um novo alocador de memória do tipo UMA. Com o conhecimento obtido neste estudo, em especial sobre a eficiência das estratégias de alocação dos diferentes alocadores avaliados, aliado à identificação dos padrões de consumo de memória das aplicações consideradas, espera-se incorporar ao projeto do novo alocador, estratégias adequadas aos diferentes cenários estudados.

REFERÊNCIAS BIBLIOGRÁFICAS

- Alexa. (2014). *The top 500 sites on the web*. Acesso em 20 de Outubro de 2014, disponível em Alexa: <http://www.alexa.com/topsites>
- Apache Software Foundation. (2014a). *Apache HTTP server benchmarking tool*. Acesso em 20 de Outubro de 2014, disponível em Apache HTTP Server: <http://httpd.apache.org/docs/2.2/programs/ab.html>
- Apache Software Foundation. (2014b). *Apache HTTP Server Project*. Acesso em 20 de Outubro de 2014, disponível em <http://httpd.apache.org/>
- Areias, M., e Rocha, R. (2012). An Efficient and Scalable Memory Allocator for Multithreaded Tabled Evaluation of Logic Programs. In *Proceedings of 18th International Conference on Parallel and Distributed Systems*, (pp. 636-643). Singapore.
- Barea, A. H. (2011). *Analysis and evaluation of high performance web servers*. Barcelona: M.S. Thesis, EETAC, UPC.
- Benosman, R., Barkaoui, K., Albrieux, Y. (2013). A new dynamic IPC-memory allocator based on a paging approach. In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, (pp. 382-389). Helsinki.
- Berger, E. D., McKinley, K. S., Blumofe, R. D., Wilson, P. R. (2000). Hoard, a scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth international conference on Architectural support for programming languages and operating systems*, (pp. 117-128). New York.
- Chang, J. M., Hasan, Y., Lee, W. H. (2000). A high-performance memory allocator for memory intensive applications. In *Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, (pp. 6-12). Beijing.
- Cho, Y., Lee, D., Jun, H., Eom, Y. (2014). Lock-free memory allocator without garbage collection on multicore embedded devices. In *Proceedings of the IEEE International Conference on Consumer Electronics (ICCE)*, (pp. 428-429). Las Vegas.
- Collins, G. O. (1961). Experience in automatic storage allocation. *Communications of the ACM*, (pp. 436-440). Volume 4. New York.

- Costa, D. E. e Matias, R. (2014). Um Estudo Experimental para Caracterização de Alocações Dinâmicas de Memória. In *Proceedings of the IV Simpósio Brasileiro de Engenharia de Sistemas Computacionais (to appear)*. Manaus.
- Costa, D. E., Fernandes, M., Matias, R., Araújo, L. B. (2013). Análise teórico-experimental de algoritmos de alocação de memória. In *Proceedings of the 3th Brazilian Symposium on Computing Systems Engineering*. Niterói.
- Costa, D. E., Fernandes, M., Matias, R., Araújo, L. B. (2014). Experimental and Theoretical Analyses of Memory Allocation Algorithms. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. (pp. 1545-1546). Gyeongju.
- Developers, V. (2014). Acesso em 23 de Setembro de 2014, disponível em Valgrind: <http://valgrind.org/>
- Dickey, T. E. (2013). Acesso em 20 de Outubro de 2014, disponível em Lynx: <http://lynx.browser.org/>
- Elah, N. (2014). Acesso em 23 de Setembro de 2014, disponível em Code::Blocks: <http://www.codeblocks.org/>
- Evans, J. (2006). A Scalable Concurrent malloc(3) Implementation for FreeBSD. In *Proceedings of the The BSD Conference*.
- Ferreira, T. B. (2012). *Análise do Desempenho de Algoritmos de Alocação Dinâmica de Memória*. Trabalho de Conclusão de Curso. Universidade Federal de Uberlândia, Uberlândia.
- Ferreira, T. B., Matias, R., Macedo, A., Araújo, L. B. (2011a). A Quantitative Comparison of Memory Allocators for Multicore and Multithread Applications. *Journal of Internet Technology*, Vol. 13, (pp. 512-532).
- Ferreira, T.B., Matias, R., Macedo, A., Araujo, L. B. (2011b). An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications. In *Proceedings of the 2th International Conference on Parallel and Distributed Computing Applications and Technologies* (pp. 92-98). Gwangju.
- Ferreira, T.B., Matias, R., Macedo, A., Araujo, L. B. (2011c). Comparação de Alocadores de Memória em Aplicações Multithread/Multicore. In *Proceedings of the VIII Workshop de Sistemas Operacionais*. Florianópolis.
- Foundation, O. (2014a). *MySQLSlap: Emulation Client*. Acesso em 20 de Outubro de 2014, disponível em MySQL: <http://dev.mysql.com/doc/refman/5.1/en/mysqlslap.html>
- Foundation, O. (2014b). *Sakila Sample Database*. Acesso em 20 de Outubro de 2014, disponível em MySQL: <http://dev.mysql.com/doc/sakila/en/>
- Garey, M. R., Graham, R. L., Ullman, J. D. (1972). Worst case analysis of memory allocation algorithms. In *Proceedings of the Fourth Annual ACM Symposium on the Theory of Computing*. (pp. 143-150). New York.

- Ghemawat, S., e Menage, P. (2014). *TCMalloc : Thread-Caching Malloc*. Acesso em 20 de Outubro de 2014, disponível em <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>
- Gloger, W. (2006). *Wolfram Gloger's malloc homepage*. Acesso em 20 de Outubro de 2014, disponível em <http://www.malloc.de/en/>
- GNU. (2012). *GNU Octave*. Acesso em 20 de Outubro de 2014, disponível em GNU: <http://www.gnu.org/software/octave>
- Grunwald, D., Zorn, B., Henderson, R. (1993). Improving the Cache Locality of Memory Allocation. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation* (pp. 177-186). New York: ACM.
- Häggander, D., Lidén, P., Lundberg, L. (2001). A Method for Automatic Optimization of Dynamic Memory Management in C++. In *Proceedings of the International Conference on Parallel Processing*, (pp. 489-498). Valencia.
- Kamp, P. H. (1998). Malloc(3) revisited. In *USENIX 1998 Annual Technical Conference: Invited Talks*, (pp. 193-198). Berkeley.
- Krishnan, M. R. (Fevereiro de 1999). Heap: Pleasures and pains. *Microsoft Developer Newsletter*.
- Larson, P., e Krishnan, M. (1998). Memory allocation for long-running server applications. In *Proceedings of the ISSM*. (pp. 176-185). Vancouver.
- Lea, D. (1996). *A Memory Allocator*. Acesso em 20 de Outubro de 2014, disponível em <http://g.oswego.edu/dl/html/malloc.html>
- Linux. (2010). *ptrace(2) - Linux man page*. Acesso em 23 de Setembro de 2014, disponível em Linux.de: <http://linux.die.net/man/2/ptrace>
- Linux. (2013). *munmap(2) Linux man page*. Acesso em 20 de Outubro de 2014, disponível em die.net: <http://linux.die.net/man/2/munmap>
- Linux. (2014). *taskset(1) - Linux man page*. Acesso em 20 de Outubro de 2014, disponível em <http://linux.die.net/man/1/taskset>
- M. Masmano, I. R. (2006). A comparison of memory allocators for real-time applications. In *Proceedings of the 4th Int'l workshop on Java tech. for real-time and embedded systems* (pp. 68-76). ACM Int'l Conf. Proc. Series.
- Matias, R., Ferreira, T. B., Macedo, A. (2011). An experimental study on user-level memory allocators in middleware applications. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics SMC*. (pp. 2431-2436). Anchorage.
- Matias, R., Macedo, A., Ferreira, T. B. (2011). Análise Experimental Comparativa de Algoritmos de Alocação de Memória de Código Aberto. *Fórum Internacional de Software Livre*. Porto Alegre.

- Michael, M. M. (2004). Scalable Lock-Free Dynamic Memory Allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, (pp. 35-46). New York.
- Montgomery, D. C. (2000). *Design and Analysis of Experiments* (3^a ed.). John Wiley.
- Mozilla Foundation. (2014). *Firefox*. Acesso em 20 de Outubro de 2014, disponível em Mozilla community website: <http://br.mozdev.org/>
- Oracle Corporation. (2014). *MySQL*. Acesso em 20 de Outubro de 2014, disponível em <http://www.mysql.com/>
- Ortega, A. L. (2014). Acesso em 20 de Outubro de 2014, disponível em Cherokee: <http://cherokee-project.com>
- Owens, M. (2013). Acesso em 20 de Outubro de 2014, disponível em Inkscape: <http://www.inkscape.org/pt>
- PostgreSQL Development Group. (2014). *PostgreSQL*. Acesso em 20 de Outubro de 2014, disponível em <http://www.postgresql.org/>
- Randell, B. (1969). A note on storage fragmentation and program segmentation. In *Proceedings of the Communications of the ACM*, (pp. 365-372). Volume 12. New York.
- Robson, J. M. (Julho de 1971). An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, (pp. 416-423). Volume 18. New York.
- Robson, J. M. (Julho de 1974). Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, (pp. 491-499). Volume 21. New York.
- Robson, J. M. (Agosto de 1977). Worst case fragmentation of first fit and best fit storage allocation strategies. *Computer Journal*, (pp. 242-244).
- Smith, G. D. (1986). Solving Linear Problems: Exact Methods. In: G. D. Smith, *Numerical Solution of Partial Different Equations* (3^a ed., pp. 119-122). Oxford: OUP.
- SystemTap. (2012). *Overview*. Acesso em 20 de Outubro de 2014, disponível em SystemTap: <https://sourceware.org/systemtap/>
- Tannenbaum, B. (2010). *Miser – A Dynamically Loadable Memory Allocator for Multi-Threaded Applications*. Acesso em 20 de Outubro de 2014, disponível em Intel Developer Zone: <https://software.intel.com/en-us/articles/miser-a-dynamically-loadable-memory-allocator-for-multi-threaded-applications>
- The NetBSD Foundation. (2014). Acesso em 20 de Outubro de 2014, disponível em The NetBSD Project: <http://www.netbsd.org/>
- Vahalia, U. (1996). *UNIX internals: the new frontiers*. NJ, USA: Prentice Hall Press.
- VideoLAN. (2014). Acesso em 23 de Setembro de 2014, disponível em VLC Media Player: <http://www.videolan.org/vlc>

Wazlawick, R. S. (2009). *Metodologia de Pesquisa para Ciência da Computação* (1 ed.). Campus Elsevier.

Wilson, P. R., Johnstone, M. S., Neely, M., Bole, D. (1995). Dynamic Storage Allocation: A Survey and Critical Review. *In Proceedings of the IWMM '95 Proceedings of the International Workshop on Memory Management* (pp. 1-116). London: Springer-Verlag.

APÊNDICE A: TOPOLOGIA DO PROCESSADOR UTILIZADO NO EXPERIMENTO DE COMPARAÇÃO DOS ALOCADORES

Tabela A.1. Primeira parte da topologia completa do processador utilizado no experimento de comparação dos alocadores.

Machine (63GB)

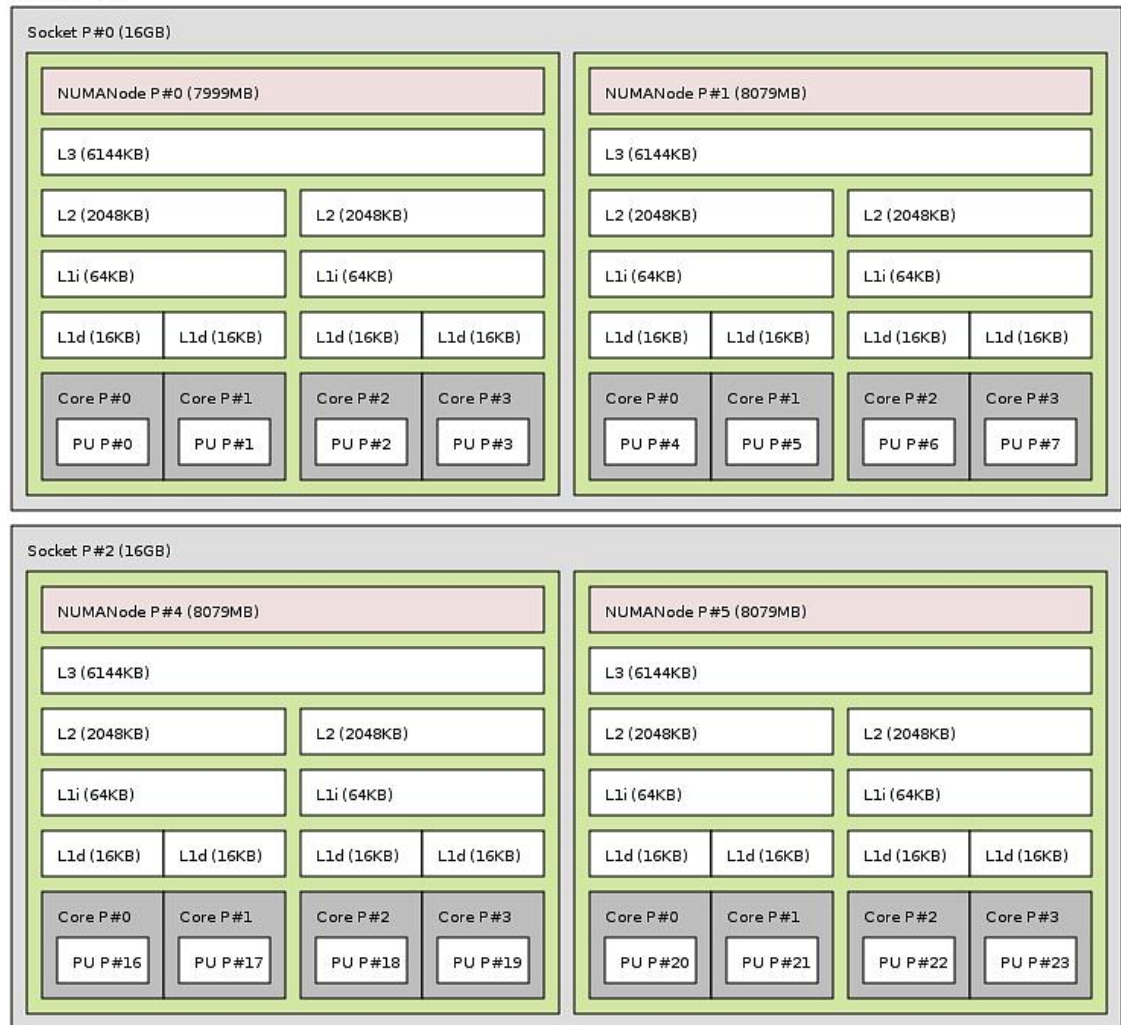
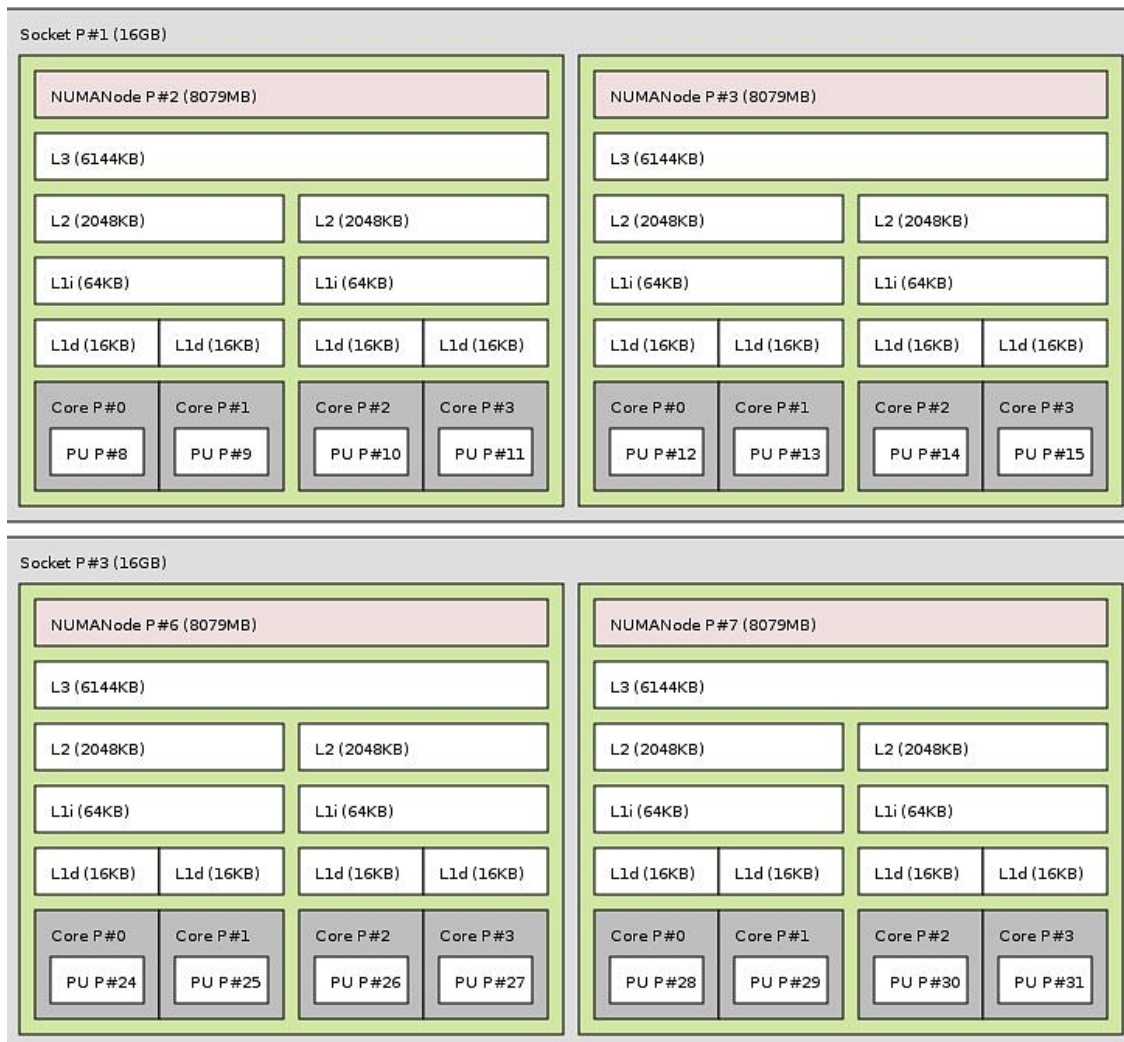


Tabela A.2. Segunda parte da topologia completa do processador utilizado no experimento de comparação dos alocadores.



APÊNDICE B: TABELAS DO TESTE DE TUKEY

Tabela B.1. Testes de Tukey da avaliação do alocador Ptmalloc2 com relação ao tempo de execução por fatores de carga de trabalho (NAxTA).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)										
Fatores		Variável	Grupos Homogêneos							
NA	TA	Tempo de Execução (ms)	1	2	3	4	5	6	7	8
500000	1	132,409		****						
500000	3	179,808			****					
1000000	1	258,417				****				
1000000	3	347,573	****							
500000	5	353,062	****							
1000000	5	684,584					****			
10000000	1	2349,088						****		
10000000	3	3268,020							****	
10000000	5	6666,179								****

Tabela B.2. Testes de Tukey da avaliação do alocador Ptmalloc2 com relação ao tempo de execução por número de threads (NT).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)							
Fator	Variável	Grupos Homogêneos					
NT	Tempo de Execução (ms)	1	2	3	4	5	6
1	991,006	****					
2	1208,346		****				
4	1574,103			****			
8	1812,326				****		
16	1901,578					****	
32	2005,400						****

Tabela B.3. Testes de Tukey da avaliação do alocador Ptmalloc2 com relação ao tempo de execução por número de processadores (NP).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)							
Fator	Variável	Grupos Homogêneos					
NP	Tempo de Execução (ms)	1	2	3	4	5	6
1	1090,804	****					
2	1414,823		****				
4	1664,365			****			
8	1706,672				****		
16	1776,894					****	
32	1839,202						****

Tabela B.4. Testes de Tukey da avaliação do alocador Ptmalloc2 com relação ao Consumo de Memória por fatores de carga de trabalho (NAxTA).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)											
Fatores		Variável	Grupos Homogêneos								
NA	TA	Consumo de Memória (MB)	1	2	3	4	5	6	7	8	9
500000	1	30,166	****								
500000	3	43,687		****							
1000000	1	59,623			****						
1000000	3	86,734				****					
500000	5	92,623					****				
1000000	5	184,561						****			
10000000	1	589,805							****		
10000000	3	862,045								****	
10000000	5	1844,348									****

Tabela B.5. Testes de Tukey da avaliação do alocador Ptmalloc2 com relação ao Consumo de Memória por *padrão de alocações de longa duração* (AL).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)			
Fator	Variável	Grupos Homogêneos	
AL	Consumo de Memória (MB)	1	2
1	418,7639	****	
2	424,2561		****

Tabela B.6. Testes de Tukey da avaliação do alocador Ptmalloc2 com relação ao Consumo de Memória por *número de threads* (NT).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)			
Fator	Variável	Grupos Homogêneos	
NT	Consumo de Memória (MB)	1	2
16	419,8550	****	
8	420,1072	****	
4	421,0151	****	
2	421,5407	****	
32	421,5662	****	
1	424,9761		****

Tabela B.7. Testes de Tukey da avaliação do alocador Ptmalloc3 com relação ao tempo de execução por fatores de carga de trabalho (NAxTA).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)								
Fatores		Variável	Grupos Homogêneos					
NA	TA	Tempo de Execução (ms)	1	2	3	4	5	6
500000	1	142,717	****					
500000	3	154,643	****					
500000	5	204,458	****	****				
1000000	1	263,612	****	****				
1000000	3	299,527		****	****			
1000000	5	409,752			****			
10000000	1	2384,270				****		
10000000	3	2977,106					****	
10000000	5	6647,052						****

Tabela B.8. Testes de Tukey da avaliação do alocador Ptmalloc3 com relação ao tempo de execução por número de threads (NT).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)						
Fator	Variável	Grupos Homogêneos				
NT	Tempo de Execução (ms)	1	2	3	4	5
1	684,170			****		
32	1416,513	****				
16	1451,462	****	****			
8	1520,712		****			
4	1639,011				****	
2	2276,890					****

Tabela B.9. Testes de Tukey da avaliação do alocador Ptmalloc3 com relação ao tempo de execução por número de processadores (NP).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)				
Fator	Variável	Grupos Homogêneos		
NP	Tempo de Execução (ms)	1	2	3
1	827,532			****
4	1572,955	****		
8	1607,318	****	****	
32	1648,801	****	****	
16	1660,053	****	****	
2	1672,100		****	

Tabela B.10. Testes de Tukey da avaliação do alocador Ptmalloc3 com relação ao Consumo de Memória por fatores de carga de trabalho (NAxTA).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)											
Fatores		Variável	Grupos Homogêneos								
NA	TA	Consumo de Memória (MB)	1	2	3	4	5	6	7	8	9
500000	1	29,474	****								
500000	3	42,994		****							
1000000	1	59,363			****						
1000000	3	86,639				****					
500000	5	91,948					****				
1000000	5	184,830						****			
10000000	1	591,891							****		
10000000	3	864,704								****	
10000000	5	1847,823									****

Tabela B.11. Testes de Tukey da avaliação do alocador Ptmalloc3 com relação ao Consumo de Memória por *padrão de alocações de longa duração* (AL).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)			
Fator	Variável	Grupos Homogêneos	
AL	Consumo de Memória (MB)	1	2
1	419,8612	****	
2	424,5088		****

Tabela B.12. Testes de Tukey da avaliação do alocador Ptmalloc3 com relação ao Consumo de Memória por *número de threads* (NT).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)			
Fator	Variável	Grupos Homogêneos	
NT	Consumo de Memória (MB)	1	2
16	421,0800	****	
2	421,2900	****	
8	422,0204	****	****
4	422,2811	****	****
1	422,9411	****	****
32	423,4975		****

Tabela B.13. Testes de Tukey da avaliação do alocador Hoard com relação ao tempo de execução por fatores de carga de trabalho (NAxTA).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)									
Fatores		Variável	Grupos Homogêneos						
NA	TA	Tempo de Execução (ms)	1	2	3	4	5	6	7
500000	1	83,228	****						
500000	3	129,177	****	****					
1000000	1	148,850	****	****					
500000	5	188,719		****	****				
1000000	3	245,773			****				
1000000	5	341,009				****			
10000000	1	1287,914					****		
10000000	3	2311,109						****	
10000000	5	2943,277							****

Tabela B.14. Testes de Tukey da avaliação do alocador Hoard com relação ao tempo de execução por número de threads (NT).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)						
Fator	Variável	Grupos Homogêneos				
NT	Tempo de Execução (ms)	1	2	3	4	5
4	491,878		****			
8	582,328	****				
2	631,612	****				
16	961,190			****		
1	1025,937				****	
32	1426,426					****

Tabela B.15. Testes de Tukey da avaliação do alocador Hoard com relação ao tempo de execução por número de processadores (NP).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)					
Fator	Variável	Grupos Homogêneos			
NP	Tempo de Execução (ms)	1	2	3	4
16	576,700	****			
32	580,515	****			
8	611,279	****			
4	778,163		****		
1	1117,755			****	
2	1454,959				****

Tabela B.16. Testes de Tukey da avaliação do alocador Hoard com relação ao Consumo de Memória por fatores de carga de trabalho (NAxTA).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)										
Fatores		Variável	Grupos Homogêneos							
NA	TA	Consumo de Memória (MB)	1	2	3	4	5	6	7	8
500000	1	23,761		****						
500000	3	32,495			****					
1000000	1	44,280				****				
500000	5	60,408	****							
1000000	3	60,693	****							
1000000	5	116,610					****			
10000000	1	402,011						****		
10000000	3	562,004							****	
10000000	5	1125,066								****

Tabela B.17. Testes de Tukey da avaliação do alocador Hoard com relação ao Consumo de Memória por padrão de alocações de longa duração (AL).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)			
Fator	Variável	Grupos Homogêneos	
AL	Consumo de Memória (MB)	1	2
1	95,2726	****	
2	444,1334		****

Tabela B.18. Testes de Tukey da avaliação do alocador Hoard com relação ao Consumo de Memória por número de threads (NT).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)					
Fator	Variável	Grupos Homogêneos			
NT	Consumo de Memória (MB)	1	2	3	4
4	264,1326	****			
8	265,3884	****			
2	265,4363	****			
1	269,5285		****		
16	271,9684			****	
32	281,7636				****

Tabela B.19. Testes de Tukey da avaliação do alocador Miser com relação ao tempo de execução por fatores de carga de trabalho (NAxTA).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)											
Fatores		Variável	Grupos Homogêneos								
NA	TA	Tempo de Execução (ms)	1	2	3	4	5	6	7	8	9
500000	1	111,281	****								
500000	3	162,453		****							
1000000	1	218,656			****						
1000000	3	315,091				****					
500000	5	351,776					****				
1000000	5	686,269						****			
10000000	1	2055,486							****		
10000000	3	3047,135								****	
10000000	5	6839,881									****

Tabela B.20. Testes de Tukey da avaliação do alocador Miser com relação ao tempo de execução por número de threads (NT).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)							
Fator	Variável	Grupos Homogêneos					
NT	Tempo de Execução (ms)	1	2	3	4	5	6
1	999,620	****					
2	1206,291		****				
4	1519,725			****			
8	1735,791				****		
16	1834,087					****	
32	1896,505						****

Tabela B.21. Testes de Tukey da avaliação do alocador Miser com relação ao tempo de execução por número de processadores (NP).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)							
Fator	Variável	Grupo Homogêneo					
NP	Tempo de Execução (ms)	1	2	3	4	5	6
1	1046,554	****					
2	1357,966		****				
4	1636,141			****			
8	1659,517				****		
16	1728,790					****	
32	1763,051						****

Tabela B.22. Testes de Tukey da avaliação do alocador Miser com relação ao Consumo de Memória por fatores de carga de trabalho (NAxTA).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)											
Fatores		Variável	Grupos Homogêneos								
NA	TA	Consumo de Memória (MB)	1	2	3	4	5	6	7	8	9
500000	1	30,348	****								
500000	3	45,651		****							
1000000	1	59,933			****						
1000000	3	90,364				****					
500000	5	93,115					****				
1000000	5	185,623						****			
10000000	1	591,021							****		
10000000	3	896,431								****	
10000000	5	1846,160									****

Tabela B.23. Testes de Tukey da avaliação do alocador Miser com relação ao Consumo de Memória por padrão de alocações de longa duração (AL).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)			
Fator	Variável	Grupos Homogêneos	
AL	Consumo de Memória (MB)	1	2
1	422,9541	****	
2	430,0784		****

Tabela B.24. Testes de Tukey da avaliação do alocador Miser com relação ao Consumo de Memória por número de threads (NT).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)				
Fator	Variável	Grupos Homogêneos		
NT	Consumo de Memória (MB)	1	2	3
4	424,6658	****		
8	424,8881	****		
16	425,6581	****	****	
2	427,0188		****	
32	427,2294		****	
1	429,6371			****

Tabela B.25. Testes de Tukey da avaliação do alocador Jemalloc com relação ao tempo de execução por fatores de carga de trabalho (NAXTA).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)										
Fatores		Variável	Grupos Homogêneos							
NA	TA	Tempo de Execução (ms)	1	2	3	4	5	6	7	8
500000	1	41,9281	****							
500000	3	43,1245	****							
500000	5	57,5300		****						
1000000	1	64,3188			****					
1000000	3	72,6351				****				
1000000	5	100,5610					****			
10000000	1	489,7032						****		
10000000	3	563,5643							****	
10000000	5	825,9785								****

Tabela B.26. Testes de Tukey da avaliação do alocador Jemalloc com relação ao tempo de execução por número de threads (NT).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)								
Fator	Variável	Grupos Homogêneos						
NT	Tempo de Execução (ms)	1	2	3	4	5	6	
8	205,2560	****						
16	208,0718		****					
4	212,0445			****				
32	217,3655				****			
2	258,2751					****		
1	405,2159							****

Tabela B.27. Testes de Tukey da avaliação do alocador Jemalloc com relação ao tempo de execução por número de processadores (NP).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)								
Fator	Variável	Grupos Homogêneos						
NP	Tempo de Execução (ms)	1	2	3	4	5	6	
32	179,3181	****						
16	186,8342		****					
8	202,0815			****				
4	237,3448				****			
2	283,3969					****		
1	417,2534							****

Tabela B.28. Testes de Tukey da avaliação do alocador Jemalloc com relação ao Consumo de Memória por fatores de carga de trabalho (NAxTA).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)											
Fatores		Consumo de Memória (MB)	Grupos Homogêneos								
NA	TA		1	2	3	4	5	6	7	8	9
500000	1	48,226	****								
500000	3	61,296		****							
1000000	1	80,565			****						
1000000	3	112,163				****					
500000	5	116,278					****				
1000000	5	222,256						****			
10000000	1	642,336							****		
10000000	3	941,488								****	
10000000	5	2041,732									****

Tabela B.29. Testes de Tukey da avaliação do alocador Jemalloc com relação ao Consumo de Memória por padrão de alocações de longa duração (AL).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)			
Fator	Variável	Grupos Homogêneos	
AL	Consumo de Memória (MB)	1	2
2	469,7065	****	
1	478,3692		****

Tabela B.30. Testes de Tukey da avaliação do alocador Jemalloc com relação ao Consumo de Memória por número de threads (NT).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)						
Fator	Variável	Grupos Homogêneos				
NT	Consumo de Memória (MB)	1	2	3	4	5
2	460,8525	****				
4	462,5241	****	****			
1	464,4776		****			
8	470,1565			****		
16	482,2090				****	
32	504,0074					****

Tabela B.31. Testes de Tukey da avaliação do alocador TCMalloc com relação ao tempo de execução por fatores de carga de trabalho (NAxTA).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)											
Fatores		Variável	Grupos Homogêneos								
NA	TA	Tempo de Execução (ms)	1	2	3	4	5	6	7	8	9
500000	1	36,662	****								
500000	3	45,559		****							
1000000	1	64,901			****						
500000	5	73,366				****					
1000000	3	81,776					****				
1000000	5	139,704						****			
10000000	1	561,823							****		
10000000	3	724,519								****	
10000000	5	1286,103									****

Tabela B.32. Testes de Tukey da avaliação do alocador TCMalloc com relação ao tempo de execução por número de threads (NT).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)							
Fator	Variável	Grupos Homogêneos					
NT	Tempo de Execução (ms)	1	2	3	4	5	6
8	260,0286	****					
16	277,5489		****				
4	279,7043			****			
32	287,9185				****		
2	373,4132					****	
1	530,9948						****

Tabela B.33. Testes de Tukey da avaliação do alocador TCMalloc com relação ao tempo de execução por número de processadores (NP).

Teste de Tukey – Grupos Homogêneos (alpha = ,05)							
Fator	Variável	Grupos Homogêneos					
NP	Tempo de Execução (ms)	1	2	3	4	5	6
8	260,3803	****					
16	267,6787		****				
32	280,1651			****			
4	297,0716				****		
2	369,3410					****	
1	534,9716						****

Tabela B.34. Testes de Tukey da avaliação do alocador TCMalloc com relação ao Consumo de Memória por fatores de carga de trabalho (NAxTA).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)											
Fatores		Variável	Grupos Homogêneos								
NA	TA	Consumo de Memória (MB)	1	2	3	4	5	6	7	8	9
500000	1	26,855	****								
500000	3	40,710		****							
1000000	1	57,488			****						
1000000	3	85,655				****					
500000	5	92,675					****				
1000000	5	191,562						****			
10000000	1	601,982							****		
10000000	3	886,654								****	
10000000	5	1952,843									****

Tabela B.35. Testes de Tukey da avaliação do alocador TCMalloc com relação ao Consumo de Memória por padrão de alocações de longa duração (AL).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)			
Fator	Variável	Grupos Homogêneos	
AL	Consumo de Memória (MB)	1	2
2	434,5859	****	
1	440,1750		****

Tabela B.36. Testes de Tukey da avaliação do alocador TCMalloc com relação ao Consumo de Memória por número de threads (NT).

Teste de Tukey - Grupos Homogêneos (alpha = ,05)					
Fator	Variável	Grupos Homogêneos			
NT	Consumo de Memória (MB)	1	2	3	4
16	433,8730	****			
32	434,4809	****			
8	435,8182	****	****		
4	437,6346		****	****	
2	439,1289			****	
1	443,3472				****

APÊNDICE C: ANÁLISE DE VARIÂNCIA DOS FATORES NO TEMPO DE EXECUÇÃO DOS ALOCADORES

Tabela C.1. Análise de variância (ANOVA) dos fatores no tempo de execução do Ptmalloc2.

	SS (Sum of Squares)	DF (Degrees of Freedom)	MS (Mean Square)	F (F-test)	p-value
NP	1,282022E+21	5	2,564044E+20	12552	0,00
NT	2,667820E+21	5	5,335640E+20	26119	0,00
NA	6,149012E+22	2	3,074506E+22	1505050	0,00
TA	9,847259E+21	2	4,923630E+21	241024	0,00
AL	4,500131E+16	1	4,500131E+16	2	0,13

Tabela C.2. Análise de variância (ANOVA) dos fatores no tempo de execução do Ptmalloc3.

	SS	DF	MS	F	p-value
NP	1,770363E+09	5	3,540726E+08	3462,0	0,00
NT	4,206150E+09	5	8,412300E+08	8225,2	0,00
NA	6,105769E+10	2	3,052885E+10	298498,1	0,00
TA	8,415864E+09	2	4,207932E+09	41143,4	0,00
AL	4,504517E+08	1	4,504517E+08	4404,3	0,11

Tabela C.3. Análise de variância (ANOVA) dos fatores no tempo de execução do Hoard.

	SS	DF	MS	F	p-value
NP	2,096502E+09	5	4,193004E+08	9204,6	0,00
NT	2,018891E+09	5	4,037782E+08	8863,8	0,00
NA	1,717040E+10	2	8,585202E+09	188464,0	0,00
TA	1,390380E+09	2	6,951901E+08	15260,9	0,00
AL	9,999058E+08	1	9,999058E+08	21950,1	0,09

Tabela C.4. Análise de variância (ANOVA) dos fatores no tempo de execução do Miser.

	SS	DF	MS	F	p-value
NP	1,247925E+09	5	2,495850E+08	14417	0,00
NT	2,123230E+09	5	4,246460E+08	24528	0,00
NA	5,841587E+10	2	2,920794E+10	1687114	0,00
TA	1,209990E+10	2	6,049949E+09	349458	0,00
AL	1,213199E+05	1	1,213199E+05	7	0,08

Tabela C.5. Análise de variância (ANOVA) dos fatores no tempo de execução do Jemalloc.

	SS	DF	MS	F	p-value
NP	1,313003E+08	5	2,626006E+07	74218	0,00
NT	9,855958E+07	5	1,971192E+07	55711	0,00
NA	1,372870E+09	2	6,864352E+08	1940040	0,00
TA	6,010987E+07	2	3,005493E+07	84943	0,00
AL	9,893587E+04	1	9,893587E+04	280	0,09

Tabela C.6. Análise de variância (ANOVA) dos fatores no tempo de execução do TCMalloc.

	SS	DF	MS	F	p-value
NP	1,805121E+08	5	3,610242E+07	71646	0,00
NT	1,752358E+08	5	3,504717E+07	69552	0,00
NA	2,660255E+09	2	1,330127E+09	2639668	0,00
TA	2,767399E+08	2	1,383700E+08	274598	0,00
AL	1,240472E+05	1	1,240472E+05	246	0,09

APÊNDICE D: ANÁLISE DE VARIÂNCIA DOS FATORES NO CONSUMO DE MEMÓRIA DOS ALOCADORES

Tabela D.1. Análise de variância (ANOVA) dos fatores no consumo de memória do Ptmalloc2.

	SS (Sum of Squares)	DF (Degrees of Freedom)	MS (Mean Square)	F (F-test)	p-value
NP	6,204550E+09	5	1,240910E+09	2	0,17
NT	5,498163E+10	5	1,099633E+10	14	0,00
NA	4,467618E+15	2	2,233809E+15	2764824	0,00
TA	8,284477E+14	2	4,142238E+14	512692	0,00
AL	1,465985E+11	1	1,465985E+11	181	0,00

Tabela D.2. Análise de variância (ANOVA) dos fatores no consumo de memória do Ptmalloc3.

	SS	DF	MS	F	p-value
NP	8,505189E+09	5	1,701038E+09	2	0,06
NT	1,410250E+10	5	2,820501E+09	4	0,00
NA	4,495085E+15	2	2,247542E+15	2849507	0,00
TA	8,305465E+14	2	4,152733E+14	526497	0,00
AL	1,049749E+11	1	1,049749E+11	133	0,00

Tabela D.3. Análise de variância (ANOVA) dos fatores no consumo de memória do Hoard.

ANOVA - Consumo de Memória - Hoard					
	SS	DF	MS	F	p-value
NP	3,801589E+10	5	7,603179E+09	16	0,00
NT	7,078466E+11	5	1,415693E+11	305	0,00
NA	1,773356E+15	2	8,866779E+14	1911662	0,00
TA	2,748053E+14	2	1,374026E+14	296238	0,00
AL	5,914809E+14	1	5,914809E+14	1275222	0,00

Tabela D.4. Análise de variância (ANOVA) dos fatores no consumo de memória do Miser.

	SS	DF	MS	F	p-value
NP	5,130275E+09	5	1,026055E+09	1	0,27
NT	5,609121E+10	5	1,121824E+10	14	0,00
NA	4,566727E+15	2	2,283363E+15	2805771	0,00
TA	8,161737E+14	2	4,080868E+14	501453	0,00
AL	2,466686E+11	1	2,466686E+11	303	0,00

Tabela D.5. Análise de variância (ANOVA) dos fatores no consumo de memória do Jemalloc.

	SS	DF	MS	F	p-value
NP	4,761296E+09	5	9,522592E+08	1	0,33
NT	4,464161E+12	5	8,928323E+11	1078	0,00
NA	5,256454E+15	2	2,628227E+15	3171946	0,00
TA	1,034057E+15	2	5,170286E+14	623990	0,00
AL	3,647099E+11	1	3,647099E+11	440	0,00

Tabela D.6. Análise de variância (ANOVA) dos fatores no consumo de memória do TCMalloc.

	SS	DF	MS	F	p-value
NP	2,745048E+10	5	5,490096E+09	7	0,06
NT	2,004706E+11	5	4,009412E+10	48	0,00
NA	4,907764E+15	2	2,453882E+15	2940401	0,00
TA	9,623752E+14	2	4,811876E+14	576590	0,00
AL	1,518176E+11	1	1,518176E+11	182	0,00