动态存储分配中装箱算法出人意料的效率

一篇思想性摘要

Christos Lamprakos cplamprakos@microlab.ntua.gr 雅典国立技术大学，雅典，希腊；鲁汶天主教大学，鲁汶，比利时

Sotirios Xydis sxydis@microlab.ntua.gr 雅典国立技术大学，雅典，希腊

Francky Catthoor francky.catthoor@imec.be IMEC 鲁汶科学园，比利时；鲁汶天主教大学，鲁汶，比利时

Dimitrios Soudris dsoudris@microlab.ntua.gr 雅典国立技术大学 希腊雅典

arXiv:2305.01497v1 [cs.PL] 2 May 2023

摘要

二维矩形装箱（二维装箱问题）是动态存储分配的一种已知抽象。我们认为，这类抽象有助于实际应用。二维装箱问题算法优化其放置方案的完工时间，即已使用地址范围的大小。乍看之下，现代虚拟内存系统采用按需分页，使完工时间不再是相关的优化判据：分配器通常采用稀疏寻址，只需关注页边界内产生的碎片化。但在嵌入式领域，内存的部分区域会被静态预分配，因此完工时间仍是一个合理的度量指标。

近期研究表明，将分配器视为黑盒二维装箱问题求解器是有意义的。例如，存在一种基于二维装箱问题的碎片化度量，它通常与最大常驻集大小（RSS）呈单调相关。鉴于该领域对碎片化定义尚无定论，且物理内存节省价值巨大，我们因此希望将分配器生成的放置方案与通过二维装箱问题制定、以优化完工时间为目标的方案进行比较。当然，分配器必须在线运行，而二维装箱问题算法则处理完整的请求轨迹；但由于双方优化的标准都与减少内存浪费有关，研究二者之间的关系仍兼具学术与实践意义。

遗憾的是，目前没有可用于动态存储分配（DSA）的二维装箱问题算法实现。本文首次提出了对最先进水平算法的实现，但尚不完整。我们通过将其输出的完工时间与原作者给出的理论上界进行比较，验证其功能。与此同时，我们识别并记录了关键细节，以助力未来开展类似工作。

我们的实验涵盖 4 种现代内存分配器和 8 种真实应用工作负载。根据实证结果，我们得出了几项值得注意的观察：就完工时间而言，分配器在 93.75% 的情况下优于 Robson 最坏情况下界。在 87.5% 的情况下，GNU 的 malloc 实现展现出与 2DBP 最先进算法相当或更优的性能，尽管后者采用离线方式运行。

最令人惊讶的是，就碎片化而言，2DBP算法表现出色，所得解最多可提升 2.46 倍。未来的研究可以利用这些发现来推动面向内存的优化。

CCS 概念：• **软件及其工程** → **虚拟内存**；**主存**；分配／释放策略。

关键词：动态存储分配、内存碎片、装箱

ACM 参考格式：

Christos Lamprakos、Sotirios Xydis、Francky Catthoor 和 Dim- itrios Soudris。2023 年。真实场景中用于动态存储分配的装箱算法的意外效率：一篇思想摘要。载于《*2023* 年 *ACM SIGPLAN* 内存管理国际研讨会论文集》（*ISMM '23*），*2023*年*6*月*18*日，美国佛罗里达州奥兰多。ACM，纽约，纽约州，美国，13页。https://doi.org/10.1145/3591195.3595279

允许免费制作本作品全部或部分内容的数字或纸质副本，供个人或课堂使用，但不得为营利或商业利益制作或分发副本，且副本首页必须附有本声明及完整引文。本作品第三方组件的版权必须得到尊重。其他用途请联系所有者/作者。*ISMM '23*，*2023*年*6*月*18*日，美国佛罗里达州奥兰多© 2023 版权所有：所有者/作者。ACM ISBN 979-8-4007-0179-5/23/06。https://doi.org/10.1145/3591195.3595279

1 引言

尽管动态内存分配（动态存储分配）在现代计算中无处不在，但从基本原理出发来看，它一直未得到妥善处理。碎片化这一根本敌人至今仍缺乏明确的定量定义 [4, 13]。尽管几十年来我们一直了解工作负载行为、分配器策略与碎片化之间的相互作用 [18], ，但我们仍缺乏一种系统方法，根据程序的动态内存特征对其进行分类。这些缺口会以物理内存开销的形式给系统带来隐性成本。

在一项姊妹研究中， [9],我们声称，将工作负载与分配器的交互[1] 构建为二维矩形装箱（2DBP）实例，可以为未来的分析与设计方法奠定有依据的基础。我们通过在表示空间中定义一种碎片化度量，并展示对于许多工作负载，该度量与最大常驻集大小（RSS）呈单调递增相关，来支持这一主张。本文从该研究结束之处继续探讨：如果 *2DBP* 是一种有效的基础，那么真实的分配器放置方案与 *2DBP* 算法生成的解相比如何？

乍看之下，这种比较似乎有悖直觉；分配器在线运行，而 2DBP 算法则以完整的请求序列作为输入。但正是这种差异让我们的研究具有价值，因为它为分配器的实际局限提供了实证视角。通过将分配器与离线预言机进行比较，我们可以衡量它们与最优行为之间的差距。预期这一差距会因应用程序而异，因此我们能够找出可从自定义放置策略中获益良多的工作负载。反过来，寻找自定义策略的过程也可能受到 2DBP 领域研究的启发。

二维装箱问题有多个子类别，但只有其中一种适用于动态存储分配 [3]。矩形通常可以沿二维平面的两个维度滑动 [7]；而在我们的情形中，矩形在一个轴上的位置必须固定，分别表示分配和释放时间。这种变体在理论文献中通常称为动态存储分配 [2]。据我们所知，目前还没有可用的二维装箱动态存储分配算法实现。因此，我们着手实现 Buchsbaum 等人在2003 [2]发表的最先进水平的算法。本文记录了我们在此过程中获得的所有见解。我们的贡献如下：

- 用于模拟动态存储分配的最先进二维装箱问题算法的部分实现[2]
- 关于 BA 本身及我们自己的实现存在缺陷的一系列说明[3]
- 根据通用策略的 Robson 最坏情况下界，对 4 种现代内存分配器在 8 种负载下的完工时间进行评估 [16]
- 比较分配器生成的放置方案与 BA 相应解在完工时间和页内碎片化方面的表现

本文其余部分安排如下：第 2 节阐述了促使我们开展这项工作的原因。第3节介绍必要的背景知识。第 4 节介绍我们的 BA 实现。第 5 节汇总收集到的实证证据，并进行初步讨论。相关工作列于第6节。第 7节总结本文。

2 动机与理由

如前所述，本文是姊妹研究的直接成果。那项研究提出，二维装箱问题可能是表示工作负载与分配器之间交互的一种有用工具。尽管不可能全面梳理该研究的所有内容，我们仍尝试总结其中的一些关键观点，并将其与本文的工作联系起来。

2.1 对结构化表示的需求

我们的研究基于这样一个猜想：若要建立严谨的内存管理理论，就必须回归基本原理 [9]。所谓基本原理，是指碎片化是任何分配器面临的主要难题，而它取决于工作负载行为与分配器策略之间的交互 [18]。

为了定义碎片化，该领域采用常驻集大小（RSS）的函数 [14], ，这可能受到了 John-stone 和 Wilson 在 1998 [6] 中提出的四种替代性表述的影响。尽管原作者鼓励人们探讨各个选项的效用，却没有人对此进行评估。通用策略是否足以应对现代工作负载尚不明确，因为支持两种观点的研究都存在 [1, 12]。

我们姊妹研究的研究目标是找到一种结构化表示，用于捕捉工作负载与分配器之间的交互，并提出一种系统化方法，使其能够在实践中应用于真实的工作负载。我们首先观察到动态存储分配研究存在两个截然不同的分支：旨在真实环境中运行的实践性研究，以及探索分配器策略的极限和其他方面的理论研究。

我们尤其感兴趣的是，DSA 与二维矩形装箱的一种变体之间的相似性

[1]在本文中，“分配器”始终指管理 Linux 虚拟内存的通用非移动分配器。下文将该算法简称为 BA，即“Buchsbaum 等人发表于 2003 [2] 的算法”。

[3]实际上，我们无法确定观察到的缺陷是由什么原因造成的。也有可能是我们的源代码中存在尚未发现的[2]漏洞，因为从纯理论构想转向实际实现预计会带来困难。我们感谢 BA 的原作者。没有他们的贡献，就不会有这篇论文。我们向两位作者表达了谢意，并发送了全文征求反馈。我们收到了三份回复，其中共同的看法是，距离算法提出已过去太久，无法再提出任何实质性的意见。两位主要作者之一（截至本文撰写时，另一位尚未回复）强调，将数学思想应用于实践时，预期需要作出调整是。

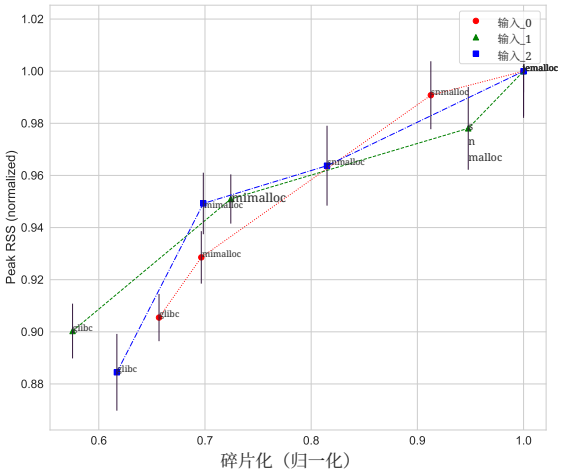

图 1. 三个 Linux `xmllint` 工作负载的峰值常驻集大小与其基于二维装箱问题的碎片化程度的散点图，数据来自四种现代内存分配器（`glibc`、`jemalloc` [4], `mimalloc` [10] 和`snmalloc` [11]）。黑色误差线表示常驻集大小测量值的标准差。碎片化程度的计算是确定性的。

（2DBP） [2, 3]。在此之前，2DBP 一直被视为 NP 难优化问题 [5], ，近似算法则会生成完工时间最短的放置方案。我们的目的并非创造一种新颖的 2DBP 算法；在按需分页的情境下，优化虚拟内存的完工时间毫无意义。但如果我们将分配器本身视为优化标准未知的 2DBP“算法”，又会怎样？由此产生的结构应当能够充分复用我们所关注的工作负载与分配器的交互。

为此，我们设计了一种可移植的基于跟踪的仿真方法，将工作负载与分配器之间的交互表示为二维装箱问题实例。为了判断生成的表示是否包含任何实际价值，我们研究了它们与最大常驻集大小（常驻集大小）之间的关系。我们定义了二维装箱问题空间中的碎片化程度，并对与 4 种现代内存分配器关联的 28 个工作负载进行了测量。对于 46.4% 的受研究工作负载而言，根据斯皮尔曼相关系数（$\rho > 0.65$），基于二维装箱问题的碎片化程度与最大常驻集大小呈单调关系。二维装箱问题中的碎片化程度越低，真实场景中的内存占用最多可减少 30% 。图1给出了一个示例。

基于跟踪的模拟数据进行计算所得的结果与经验性常驻集大小测量相关，这一事实让我们确信二维装箱问题的威力。这种相关性并不均匀，这意味着与通常的看法相反，碎片化并非 总是导致常驻集大小波动的罪魁祸首。

2.2 使用二维装箱问题的逻辑结论

如果将工作负载-分配器对表示为二维装箱问题实例是合理的，那么计算其近似最优解并探索这些解之间的关系，或许能带来有用的见解。例如，到目前为止，除了真实分配器在现实输入上的放置通常比 Robson 的最坏情况下界好得多之外 [1, 6, 16]，我们对真实分配器放置的界限还没有任何具体认识。然而，Robson 的界限和离线二维装箱算法都假定内存采用单一的连续内存映射——因此将碎片化概念化为放置方案的完工时间偏离最优值。

一方面，采用按需分页的 Linux 虚拟内存使完工时间变得无关紧要；众所周知，分配器通常采用稀疏寻址来应对碎片化（这促使我们在姊妹研究中定义了一个页面局部指标）。另一方面，Linux 系统并非唯一使用动态内存分配的系统。在嵌入式领域，预先静态分配并管理连续的物理空间往往是常态。沿着这一思路推演至其逻辑结论，我们得到这篇思想性摘要的立论依据：

- 碎片化是一个依赖上下文的概念
- 2二维装箱问题是对碎片化成因的一种与上下文无关的表述，也就是程序行为与放置策略之间的相互作用
- 我们不知道分配器在碎片化方面能够达到的极限
- 我们知道2DBP算法针对某一种特定碎片化定义所能达到的极限
- 在 2DBP 的通用基础上比较 2DBP算法与分配器，可能会带来有益的成果——一个简单的例子是能够区分哪些程序可以从自定义放置策略中进一步获益，哪些不能。

3 背景

本节介绍我们所采用的 2DBP表述（第 3.1 节），以及我们所实现的算法（第 3.2 节）。

3.1 将程序与分配器的交互视为装箱实例

假设某个应用程序正在运行，并且在动态内存方面由某个指定的分配器为其提供服务。在运行过程中，该应用程序会生成一个请求序列Krequests $R = \{R_0, R_1, \ldots\ldots, R_{K-1}\}$。为简便起见，假设对于每个请求 R_i，$i \in \{0, \ldots\ldots, K-1\}$，以下条件均成立：

$$R_i = \begin{cases} M(n) & \text{(allocate n bytes)} \\ F(j),\ j < i & \text{(free memory allocated for request } R_j\text{)} \end{cases} \tag{1}$$

分配器可以视为一个函数 $A()$，作用于类似 R这样的请求序列。处理完最后一个请求后，分配器的输出是一组 $N = K/2$ 已放置的内存块，以下称为"作业"：

$$A(R) = J_{PA} = \{J_{PA,0}, ..., J_{PA,N-1}\} \tag{2}$$

$$J_{PA,i} = (t_{sA,i}, t_{eA,i}, h_{A,i}, p_{A,i}) \tag{3}$$

下标 PA 表示"由分配器 A 放置"，i 是唯一的作业标识符（$i \in \{0、.....、 N-1\}$），$t_{sA,\ i}$ 是作业 i 被分配的时间点，$t_{eA,\ i}$ 是相应的释放时间，$h_{A,\ i}$ 是 A 分配的内存块大小（根据大小类和块元数据策略，不同分配器会为相同大小的请求生成不同大小的内存块），而 $p_{A,\ i}$ 是作业放置处的虚拟地址。时间以已分配字节数计量，并按照以下规则向前推进：

$$t(R_i) = \begin{cases} 0 & \text{initially} \\ t(R_{i-1}) & \text{iff } R_i = F(j):\ j < i \\ t(R_{i-1}) + h_{A,i} & \text{iff } R_i = M(n) \end{cases} \tag{4}$$

我们还假设以下陈述成立：

- 所服务的应用程序是单线程且确定性的。每次使用相同输入执行时，它都会生成完全相同的请求序列
- 分配器将所有作业放入同一内存映射[4]
- 不存在内存泄漏，不存在重复释放，更一般地说，请求序列是 格式良好的。

作业序列 J_{PA} 可以视为分配器 A 为由相应的未放置 作业序列 J 定义的二维装箱问题（二维装箱）所设计的解：

$$J = \{J_0, ..., J_{N-1}\} \tag{5}$$

$$J_i = (t_{s,i}, t_{e,i}, h_i) \tag{6}$$

然而，分配器 A 并非像二维装箱问题中通常那样优化最终放置的完工时间，而是根据其策略所隐含的某种未知判据，将每项作业放置在 J 中。

现在我们再记下一些稍后会用到的定义。首先，如果一个作业在开区间 $(t_{s,i}, t_{e,i})$ 内存活，就认为它是存活的。因此，我们定义存活函数 $a(J_i, t)$:

$$a(J_i, t) = \begin{cases} 1 & t_{s,i} < t < t_{e,i} \\ 0 & \text{elsewhere} \end{cases} \tag{7}$$

某一特定时刻的负载 对应于在 t时仍处于活动状态的作业的高度总和 t：

[4]我们所说的"内存映射"，是指由分配器管理的连续虚拟地址空间，它通过 `sbrk`或 `mmap` 获取。

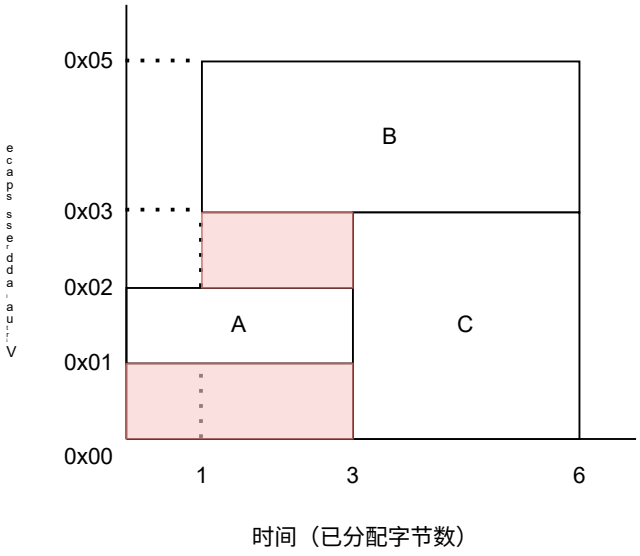

图 2. 一个简单的二维装箱问题示例。考虑以下请求序列：(i) A = malloc(1)，(ii) B = malloc(2)，(iii) free(A)，(iv) C = malloc(3)，(v) free(B)，以及 (vi) free(C)。此图将上述请求与一个假想分配器的响应结合起来，将块 A 放置在虚拟地址 0x01，将块 B 放置在 0x03，将块 C 放置在 0x00。水平轴表示已分配字节数随时间的变化。每次分配请求后，时间向前推进；每次释放请求后，时间保持不变。姊妹研究提出的关于碎片化的定义由两个阴影矩形表示。它们代表分配器未使用的区段，因此分配器保留了较高的地址以处理所有请求。

$$l(t) = \sum_{i=0}^{N-1} a(J_i, t) h_i \tag{8}$$

在整个生命周期内，一个作业贡献单个负载$|J_i| = (t_{e,i} - t_{s,i})h_i$。一系列作业 J由其总负载 $L_T = \sum_{i=0}^{N-1} |J_i|$来表征，也由其最大负载 $L = l(t_{tL}):\ l(t_i) \le L,\ \forall i \in \{0, ..., N-1\}$来表征。

二维装箱问题的一个贴切类比是俄罗斯方块游戏：我们希望尽量缩小已放置方块之间的间隙。我们提出的碎片化指标是间隙与总负载之比。回到公式 2 中的分配器放置情况，我们遍历其中的内容，并记录属于同一虚拟页的各个作业之间的所有间隙。请注意，在采用按需分页的虚拟内存情境下，将页与页之间的间隙视为碎片化来源是没有意义的。

因此，放置 J_{PA} 除了作业之外，还意味着存在第二个矩形序列 G，对应于本段所定义的作业间隙。假设这些间隙的数量为 N_G，则可将碎片化量化为：

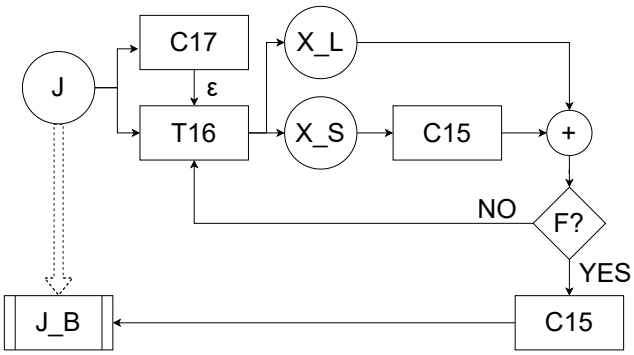

图3。 BA 理论上。 J， X_L 和 X_S 表示中间盒子集合。 + 算子将多个盒子集合合并为一个集合。 F 条件将在正文中解释。三个矩形是 BA 的基本支柱：推论 17（第3.2.2节）、定理16（第3.2.3节）和推论15（第3.2.4节）。 J_B 是算法的最终产物，即一组 高度相同的盒子。双点虚线箭头表示初始的不同尺寸作业序列经过 BA 应用后，如何转变为一组尺寸相同的盒子。

$$F = \frac{\sum_{j=0}^{N_G} |G_j|}{L_T} \tag{9}$$

3.2 算法

BA 对形如以下公式所定义的输入进行处理：5 和6。与公式2 和3 中的抽象分配器一样，它会生成一系列已放置的作业。在任意时刻，每个地址至多由一个作业占用。

所优化的判据是输出放置方案的完工时间M，也就是某个放置方案所使用的最大地址。BA 能生成近似最优解。具体而言，BA 保证 $M \le [1 + O((h_{max}/L)^{1/7})]L$。同一篇论文还介绍了一种更强的变体，它能为每个 $\epsilon > 0$ 生成满足 $M \le (2 + \epsilon)L$ 的放置方案。[5]

由于一些原因（稍后将在第 3.2.7 节说明），我们目前只能实现 BA 的较弱版本。接下来先详细介绍这个较弱版本。其概览见图 3。关于各个组件，即原论文中推导出的定理和推论，我们沿用 STOC 会议论文集中确立的命名和编号

[5]如果两种算法之间的强弱关系并不明显，可以考虑以下情况：(i) 对于 $h_{max} = L$ $M \le [1 +$, $O(1)]L \Rightarrow \exists\, c_1 > 0$ $M \le 1 + c_1 L$ 弱算法保证 ( ) : ( )。要胜过强版本，应满足 $1 + c_1 < 2 + \epsilon \Rightarrow c_1 < 1 + \epsilon$。可以说，这类实例或许会出现，但由于我们对 c_1的范围一无所知，就必须按照最坏情况进行推理，并认为 c_1 超过 $1 + \epsilon$的概率要高得多。(ii) 对于 $h_{max} = L/128$，我们同样可以得到条件 $c_2 < 2(1 + \epsilon)$。请记住，强算法适用于所有 $\epsilon > 0$，因此我们不妨认为它足够小，以至于要求 $c_2 < 2$。同样，我们没有证据表明 c_2的范围如此狭窄。

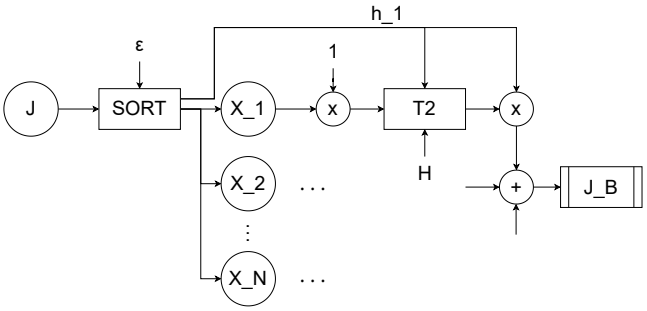

图4。推论15，展开说明。这里出现了一个新组件，即定理2。稍后会详细解释。这里还展示了一个新算子 x。它用于更改作业的高度。例如，它将 X_1中的作业从初始高度 h_1更改为单位高度作业。

版本 [2]。如果可能，强烈建议将其副本与这篇学术摘要并排阅读。

我们已尽可能让本文自成一体，力求描绘并阐述我们从原始出版物中实现的所有内容。尽管如此，BA 仍然相当复杂，这是可以理解的。除了查阅原始出版物，我们还建议将本节内容读两遍：先按默认顺序读一遍，再倒序读一遍。

3.2.1 主要思想。 由于序列 J 中存在多种尺寸，二维装箱问题是 NP 困难；如果所有矩形的高度都相同，则可以通过区间图着色（IGC）推导出该问题的最优解。BA 利用了这一事实。

BA 将所有数据视为 盒子，其定义由与公式 6相同的三元组给出。 J 中的作业是空盒子。算法的多个部分会创建包含现有盒子的新盒子。最终目标是将 中的所有作业装入 J，使其成为一系列高度相同的盒子。然后可以应用 IGC 来找到最优放置方案。图3 展示了如何将 J 转换为由 J_B 组成的一组同尺寸盒子。从此处开始，每当使用“作业”一词时，它既可能指实际作业，也可能指盒子。对于 BA 来说，一切都是盒子。

3.2.2 推论 17. 这是算法的入口点。[6]它唯一的作用是计算误差参数 ϵ，并将其作为输入提供给定理16。 ϵ 定义为（h_{max}/L）$^{1/7}$。直观地说，它的值越大，整体装箱的效果就越不理想。

3.2.3 定理 16. 此阶段处理任意作业集合，并要求将误差参数 $\epsilon \in (0, 1]$ as 作为额外

[6]将推论和定理视为执行阶段，可能会让人觉得奇怪。但原论文中的每个证明都是通过构造给出的：先提出一个论断，再根据先前的推论和定理描述要对输入执行的操作，最后根据所涉及操作的性质验证该论断。因此，完整地跟随 BA，就相当于跟随每个组成部分如何应用于其余部分。

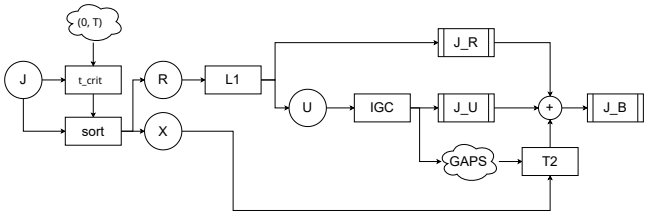

图5. 定理2，展开形式。文中也出现了引理 1 和区间图着色（IGC）。

输入。然后计算元素 $r = h_{max}/h_{min}$、 $\mu = \epsilon/\log^2 r$和 $H = \lceil \mu^5 h_{max}/\log^2 r \rceil$。接着将作业划分为两个互不相交的子集：

- X_S：高度不超过 μH 的作业
- X_L：其余作业

应用推论15处理 X_S，误差参数为 $\epsilon = \mu$，高度参数为 H。由此得到一组高度为H的盒子。将这些盒子与 X_L、r和 μ 中的作业合并，$\log^2 r \geq 1/\epsilon$或$\log^2 r < 1/\epsilon$会被重新计算，随后出现以下两种情况之一：要么定理16
??

3.2.4 推论15。此阶段接受：(i) 一组作业 J，(ii) 一个误差参数 $\epsilon > 0$ 以及 (iii) 一个高度参数 H。作业 J的高度必须至多为 ϵH；由于图3中 X_S的定义，默认满足此条件。

图4展示了这一阶段：根据不等式$(1+\epsilon)^{i-1} < h \leq (1+\epsilon)^i$，将作业分类放入桶中。同一桶中的作业会向上取整，使其高度为 $h_i = \lfloor (1+\epsilon)^i \rfloor$。因此，图中$X_1$ 的所有作业高度均为 h_1，依此类推。随后将每个桶缩放至单位高度，并调用定理2，高度参数为$\lfloor H/h_i \rfloor$。再将定理2的输出放大为高度为H的盒子，并将所有盒子子集（即对每个 X_i并行应用上述流程所创建的子集）合并为一个集合J_B，其中包含高度为 H的盒子。

3.2.5 定理2。 这是 BA 中最复杂的部分，是一个递归过程。图 5 中展示了该过程。定理 2 的输入为：(i) 一组单位高度作业；(ii) 一组 边界区间 ，在图中以云表示。初始时，边界区间等于 J 中作业的整个时间范围，再加上一个至少有一个作业处于存活状态的随机时刻。边界区间对应于 J 中用于对作业排序的时间段；这次排序依据的不是作业高度，而是其存活性：

- 如果我们将所有边界区间的端点称为关键时刻， R 包含至少在一个关键时刻处于存活状态的作业
- 集合包含在任何关键时刻都不处于活动状态的作业。

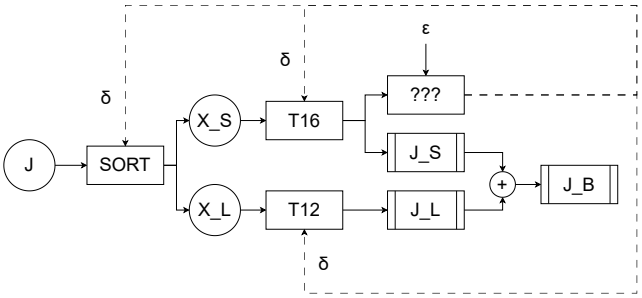

图 6。 定理 19。连续箭头表示时间正常推进。虚线箭头表示随时间传递的信息。我们未能成功实现这一点。

随后，将引理 1 应用于 R 中的作业。由此生成一组装箱作业 J_R 和一组未解决作业 U。U 中的作业通过 IGC 进行装箱，并基于该装箱过程，除 J_U 外，还会生成一组新的边界区间。这些区间将作为输入，进一步递归调用定理 2，重点处理 X 中的作业。在最后一次调用定理 2 时，预计 X 为空。返回的装箱作业将与较浅递归层的结果合并。

3.2.6 引理 1。 这是 BA 的基石，而且足够简单直接，无需配图。输入为：一组在特定时刻处于存活状态的单位高度作业 t，以及一个高度参数 H。输出为一组盒子 J_B 和一组尚未解决、即尚未装箱的作业 U。

我们省略装箱推导，因为它主要服务于原始出版物中的数学论证，并由此得出整体算法与完工时间相关的保证。进一步阐述其内部机制不属于本文的讨论范围。

3.2.7 定理19。 如第 3.2 节所述，BA论文中有一种比推论 17 更强的算法。该算法如图 6所示。其输入是任意一组作业和一个误差参数 $\epsilon > 0$。

该定理带来的主要难点在于，它使用了未来的信息。算法首先根据某个尚未计算出的“微小正数 δ”进行基于高度的排序操作。随后，它将定理16应用于第一个子集，得到一个（$1 + c\delta$）-近似。然后它通过下式计算 δ：

$$\delta(c+1) = \epsilon \tag{10}$$

这一信息必须穿越回过去，以供排序操作和定理16本身使用！我们在图 6 中用“???”标示这一难点。

我们认为有一种变通方法属于某种形式的推测执行：选择 δ 随机值，如果它在可接受的误差范围内满足公式 10，就继续执行剩余步骤；否则重试。我们没有进一步探索这一想法。

4 实现

现在来谈谈我们的实现。本文并非旨在探讨软件工程；我们不会花篇幅介绍所用的具体工具，例如编程语言、排序算法、作业表示等。我们将着重讨论遇到的高层次障碍，以及为克服这些障碍而设计的高层次机制。经验更丰富的开发者或许会提出更高效的机制，但他们也会遇到我们接下来要讨论的同样复杂问题。

表 1. 导致我们调整 BA 的一个示例。如果定义 X_L和 X_S的高度阈值小于 J中的最小高度，定理16 就会陷入无限循环。

h_{min}	h_{max}	(ϵ_t, h_t)	(ϵ_p, h_p)
8	524288	(0.76, 0.033)	(6.19, 8.27)
8	1048576	(0.97, 0.037)	(6.55, 10.01)
16	524288	(0.75, 0.037)	(6.12, 18.91)
8	75497472	(0.98, 0.020)	(6.59, 9.62)

表 1展示了一个示例。前两列显示了一些代表性工作负载的特征。第三列包含 (i) 推论 17 计算得出的理论 ϵ，以及 (ii) 定理16 用于将其作业拆分到 X_S和 X_L中的相应阈值。鉴于定理16 是递归的，如果在某个时刻 X_S为空，算法就会陷入无限循环。如果我们依赖推论 17 提供的理论 ϵ，几乎所有输入都会出现这种情况。表1的第四列显示了使定理16 得以收敛的相应数值。在这种情况下，所使用的实际 ϵ 数值远远超出了$(0, 1]$ BA 理论上要求的范围。

此时，我们可以选择放弃，也可以设计一种容错方案，以便继续推进。由于各个阶段之间存在很强的依赖关系，这一点现在应该已经显而易见，因此无法孤立地处理每个组件。我们选择了第二种方案。我们的设计基于两个关键观察：

- 组件可分为安全和不安全部分。例如，将不同的盒子集合合并为一个集合是安全的。调用另一个组件则是不安全的
- 大多数情况下，不安全部分发生故障是因为 ϵ 推论 17 计算出的值

接下来的部分将说明如何充分利用这些观察结果。

4.1 通过故障传播控制不安全部分

BA 具有级联结构。在调整参数他组件，在每个阶段重新装箱。如果我们将每个定理和推论视为一个单独的函数，堆栈跟踪从推论 17 开始，并贯穿流水线的其余部分不断增长。

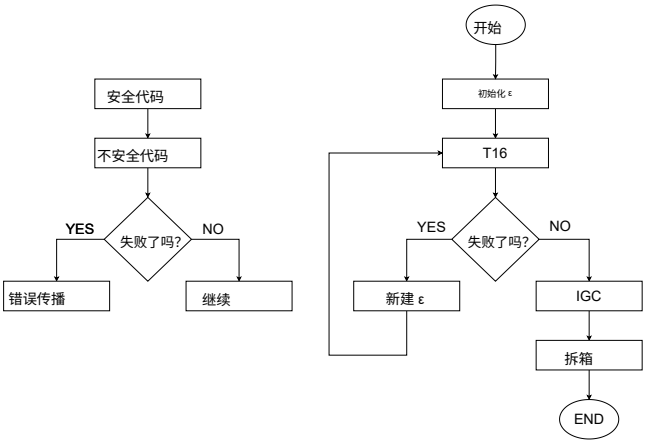

图 8. 我们的容错机制将每个 BA 组件划分为安全部分和不安全部分。每个不安全部分都映射到一个特定的判据，用于判断该部分是否发生故障。发生故障时，错误信号会沿流水线反向传播，最终到达 BA 的入口点，即推论 17。此机制经过调整，以便在检测到故障时更改 ϵ 的值（右侧）。

经过反复试验，我们推断出 ϵ 最常是导致某个组件未能按预期工作的原因。因此，我们的想法是在检测到故障时，将信号传播至调用栈的根部，也就是 BA 的入口点——推论 17。如图8右侧所示，在这种情况下会计算出新的 ϵ，并重试整个流程。因此，我们投入了大量精力来识别可能导致故障的组件代码段（图8中的"不安全代码"）。表1中已给出一个示例。

识别过程完成后，实现容错机制就很简单了。我们只需在执行不安全代码后立即检查特定情形条件。如果条件的结果是"错误"，我们就向调用方传播错误信号。所有对组件函数的调用本身都被视为不安全点：因此，被调用方返回后，调用方会检查自身的错误条件；如果发现错误，就会继续向上层传递，直到到达推论 17，由它调整 ϵ。

4.1.1 生成新的 ϵ 值。 一些简单的启发式方法负责监督重新计算推论 17 的误差参数的过程：

- 在第一次迭代中，遵循第 3.2.2
- 如果最后一个值未装入任何作业，则将 ϵ 增加 10%。
- 如果最后一个值装入的作业 比前一个更多，就将它设为下限。未来再尝试更小的值没有意义

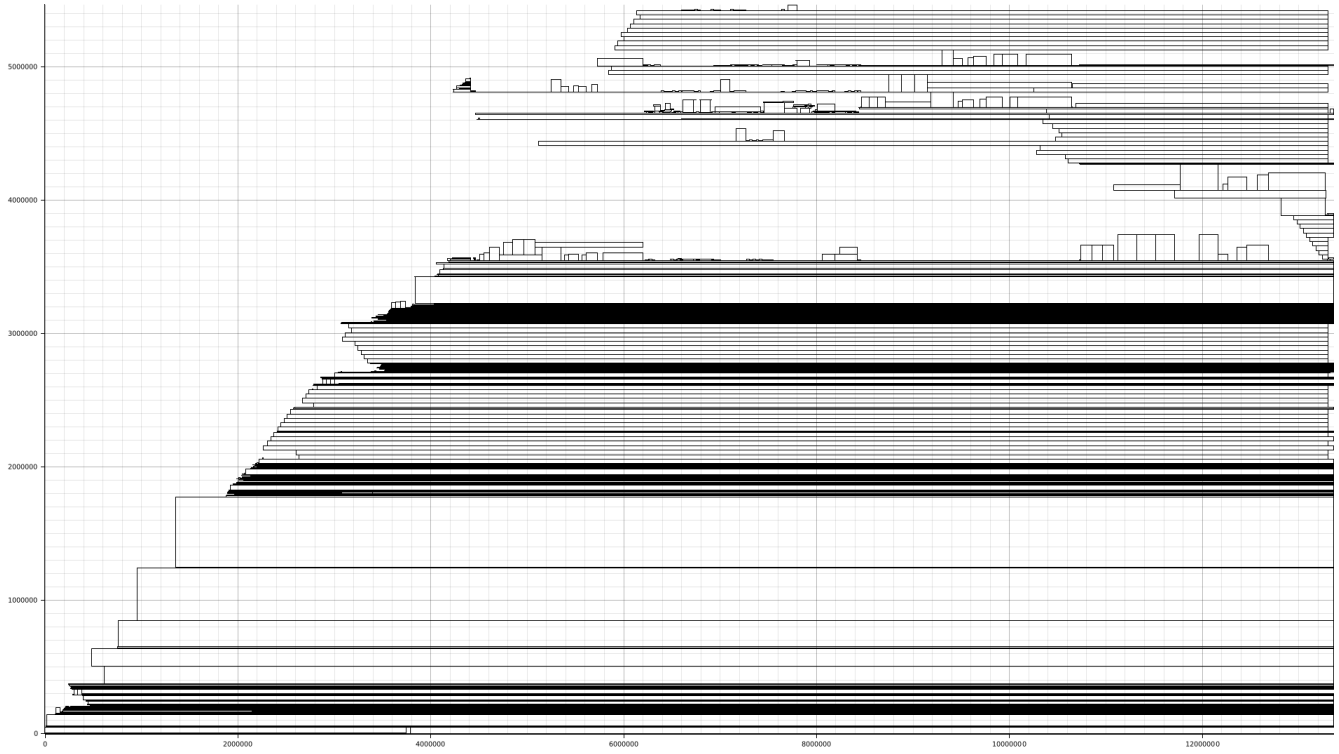

图 7。我们的 BA 实现为 bork基准测试生成的一个示例放置 n.

- 如果增加后，ϵ 超过下限的两倍，就从下限重新开始。尝试过大的值也没有意义

4.1.2 辅助信号。除了错误信号之外，我们还使用了另外两个事实证明必不可少的信号。第一个信号已经提到过：我们会跟踪原始输入集中的多少个作业在失败前成功装入了箱中。第二个信号将在下文讨论。

4.2 通过关键时刻注入绕过定理2的边界情况

正如定理16取决于 X_S 是否包含任何作业才能收敛一样，定理2取决于 R 是否非空（见图 5）。

定理2的排序操作采用存活性判据：在至少一个关键时刻处于存活状态的作业会被放入R。我们注意到，在某些边缘情况下，输入中没有任何作业符合这一条件。理论上，这种情况不应该发生：BA论文从未暗示需要考虑 R 是否为空。但在实践中，我们必须考虑这一点。我们认为，这是由于初始化组件边界区间时使用了"随机时刻"（见第3.2.5节）。由于时间以已分配字节数衡量，取值范围足够大，因此可能出现此类边缘情况。

因此，在检测到空的 R时，除了错误信号外，还会返回一个不同的时刻，此时存在问题的输入确实包含活动作业。这是唯一一次错误没有一路向上传播回入口点的情况，因为此时重新尝试定理2是合理的：将返回的时刻注入边界区间向量后再次执行。如果第二次尝试仍然失败，我们允许实现自行处理，在遇到上述边界情况时使用适当的关键时刻，而不返回。[7]

4.3 处理装箱作业以获取最终放置

收敛后，BA会生成一组高度相同的盒子。这些盒子中包含盒子，盒子中又包含盒子，如此层层嵌套，直至每个盒子的核心部分，其中放置着一部分初始作业。此时，所有内容都已装入盒子，但尚未进行放置。如图8 右半部分所示，接下来还有两项任务：

- 通过 IGC 推导出最优外层箱体放置方案
- 递归拆箱已放置的盒子，直到找到原始作业。在每个拆箱阶段，根据父级的放置位置放置子项，即所包含的盒子。

原始出版物中一个容易被忽略的重要细节是，拆箱最后一个作业后形成的放置非常稀疏，这可能是递归

[7]是的，这看起来确实有多么投机取巧就有多么投机取巧。我们期待并赞赏未来通过比反复试错更具原则性的方法付出努力，从而得出最终答案。

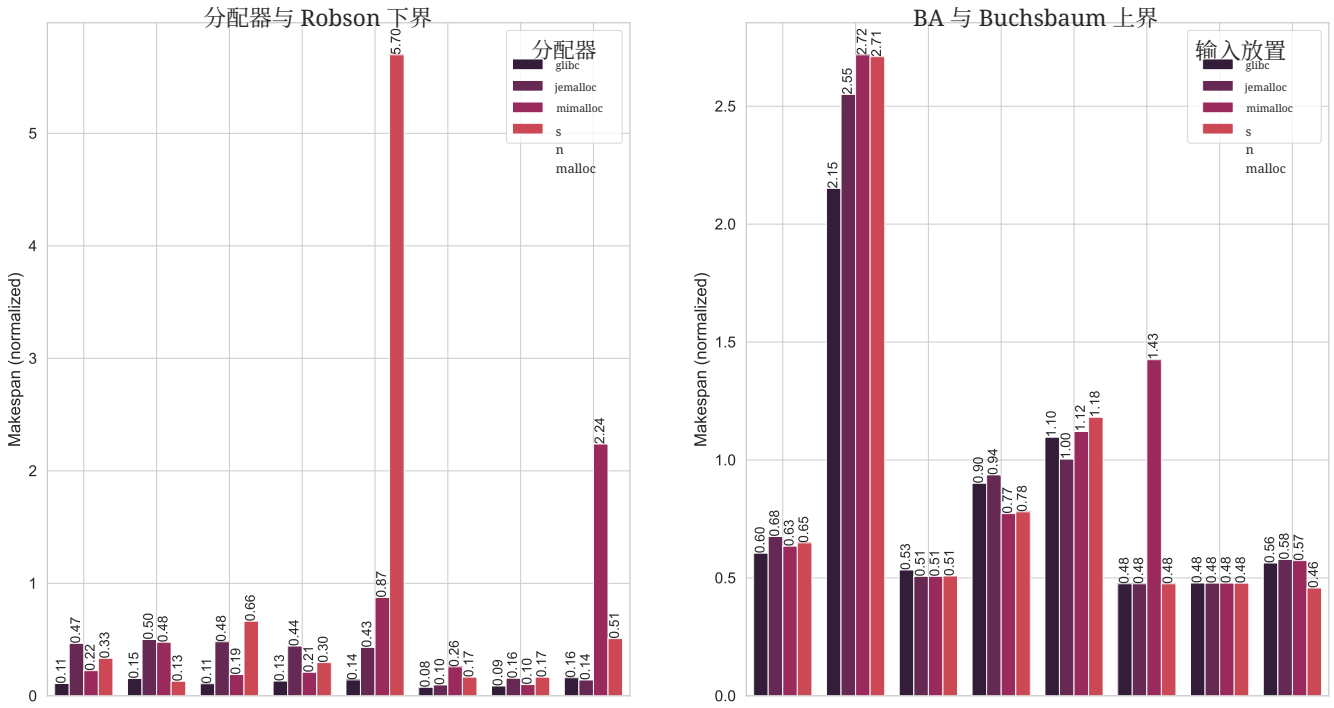

图 9. 左：各分配器的完工时间相对于 Robson 最坏情况下界（$\frac{1}{2}L \cdot log_2 h_{max}$）的归一化结果。右：BA 的完工时间相对于其对应上界 $[1+2\cdot(h_{max}/L)^{1/7}]L$ 的归一化结果。请注意，我们将原上界中的大 O 表示法替换成了 2 倍因子。

缩放由推论15施加。要使该放置更紧凑，可以采用一个非常简单的算法：(i) 按地址递增顺序排列作业，(ii) 将第一个作业放置在地址零处，(iii) 保留一个包含所有遍历过的作业的向量。遍历其余作业，每次检查遍历向量，找出与当前作业时间重合的作业。如果存在这样的作业，就将当前作业紧挨着最高的重合作业放置；否则将其放置在地址零处。持续遍历，直到完成。

4.4 关于性能和可扩展性的说明

BA 的递归特性往往会导致一台配备 32 GiB 动态随机存取存储器的服务器，在仅包含 2 万个作业的工作负载下也出现内存不足错误。这个数量至少比真实工作负载的规模小两个数量级。不过请注意，作业数量并不是预测 OOM 杀手的决定性因素：正如后文（表 2 和 3）将展示的，我们成功完成了针对包含 8 万和 20 万个作业的工作负载的 BA 全程运行。因此，我们认为，更细粒度的特征（例如作业的大小分布，以及它们在时间上的放置方式，即由程序行为控制的放置）更为相关。我们尚未找到区分可处理与不可处理情形的判据。

5 结果与讨论

我们在一台 x86_64运行 Ubuntu 20.04 LTS 的商用服务器上进行了实验。我们从一组采用多种最先进的分配器的应用程序中收集放置数据，随后将这些数据输入 BA。所有基准测试均来自 openbenchmark- ing.org 上的单线程测试集合。[8] 我们使用 Phoronix TestSuite[9] 来安装并运行这些应用程序。测量结果的完整记录见表 2和 3。

所有展示的结果均来自基于跟踪的模拟。在此情境下，Linux 内核侧的虚拟到物理映射决策并不重要：其基本假设是所有映射均为连续映射。所有评估均针对虚拟地址空间中的放置进行。此类决策对主存的影响程度已在本文的姊妹研究中确定（详见第2节）。

5.1 界限比较

我们有两个理由将放置方案与理论界限进行比较：对于分配器而言，这是一个很好的机会，可以看看它们相对于 Robson 最坏情况下界的表现

[8] https://openbenchmarking.org/suite/pts/single-threaded [9] https://www.phoronix-test-suite.com/

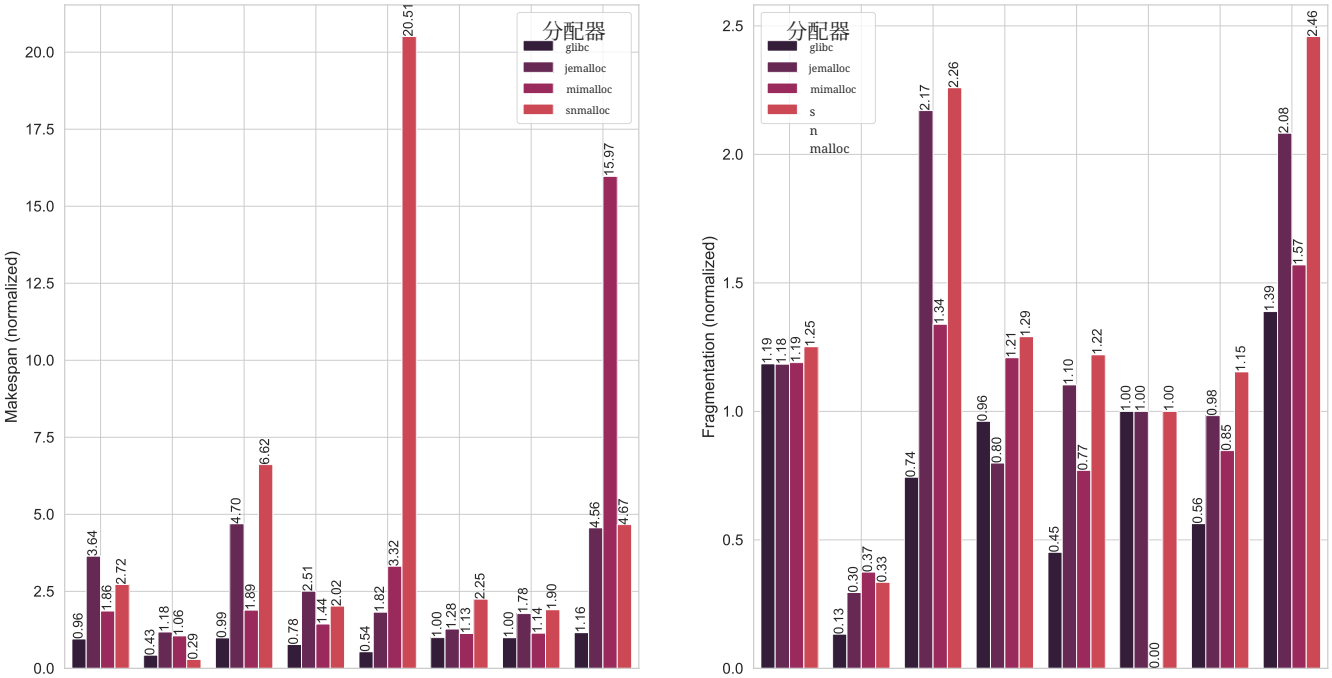

图 10. 左：分配器的完工时间相对于相应 BA 放置的完工时间进行归一化。右：基于 2DBP 的页局部碎片的对应数据。特别是在 dcraw 和 mimalloc 上，分配器的碎片化得分为完美的 0，而 BA 的表现更差——因此图中为零值。

，该下界定义为 $\frac{1}{2}L \cdot log_2 h_{max}$，对应于"最优策略" [16,17]。对于 BA 而言，遵循作者给出的理论上界，即 $M \leq [1 + O((h_{max}/L)^{1/7})]L$，是衡量我们实现功能正确性的可靠指标——考虑到我们为实现收敛而采用了大量临时修补措施，这一点尤为重要。两种比较结果均见图 9。

尽管采用了稀疏寻址，分配器在 32 种情况中的 30 种情况下仍优于 Robson 最坏情况界。需要注意的是，我们通过从使用过的最大地址中减去最小地址来计算完工时间，并且始终使用虚拟地址进行操作。因此，即使位于"端点"内存块之间、可能从未映射到物理内存的区域，也会被计入其中——这意味着稀疏寻址确实会影响所报告的完工时间。这一结果与先前在 RSS 领域得出类似结论的研究一致 [1, 6]。

关于 BA，我们得到了一个更为紧致的上界。在 75% 的时间里，我们的实现确实符合 $M \leq [1 + 2 \cdot (h_{max}/L)^{1/7})]L$。

尤其值得注意的是glibc在所有工作负载下相对于 Robson 下界都展现出了出色的性能。通过检查图7所示的可视化放置文件，我们发现glibc 几乎从不使用稀疏地址空间来寻址每个生成的内存映射，而更"现代"的实现（例如 mimalloc）则并非如此。我们认为，其极短的完工时间正是源于这一观察。

5.2 分配器与 BA

图 10 展示了分配器相对于 BA 的性能实证证据。这里有不少值得关注之处。鉴于本文是一篇思想性摘要，我们不打算全面解释所展示的数据。不过，我们会尽最大努力给出初步解读。

1. 在 75% 的情况下，BA 在完工时间方面优于分配器。这是意料之中的，因为 BA 具有离线优化完工时间的特性。

   一个不太明显的细节是，从趋势上看，图10's 左半部分大体上与图9's 左半部分一致。换言之，分配器相对于 BA 的完工时间，与分配器相对于 Robson 下界的完工时间成比例。这或许表明，BA 的完工时间比 Robson 的数值更适合作为下界。但本文暂不探讨这个问题
2. 在 8 个工作负载中，有 7 个工作负载里 glibc 的完工时间与 BA 相当或更短。我们已经提到过，观察到 glibc 不愿采用稀疏放置，但它竟然能成为 BA 如此强劲的竞争对手，仍令人惊讶。

表2。为得出所讨论的实验结果，我们进行了完整的测量。这里唯一的额外信息是每个工作负载中的作业数量。当前数据在表 3中继续。

工作负载 (#作业)	分配器	L	h_{max}	Robson 界	BA 界	设置	完工时间	碎片化
bork (12837)	glibc	4919904	528376	46766652	8869297	idealloc	5365096	0.149
						real	5130728	0.176
	jemalloc	4923384	524288	46772148	8870789	idealloc	5993192	0.147
						real	21826256	0.174
	mimalloc	4956152	524288	47083444	8926065	idealloc	5664904	0.150
						real	10550912	0.178
	snmalloc	5267888	524288	50044936	9450893	idealloc	6132336	0.151
						real	16695280	0.189
bullet (79950)	glibc	35388688	26742776	436566283	72928235	idealloc	156912368	0.256
						real	67285248	0.034
	jemalloc	38789376	29360128	481130908	79945817	idealloc	203864400	0.047
						real	240987280	0.014
	mimalloc	38101248	27262976	470558789	78204253	idealloc	212559984	0.043
						real	224395264	0.016
	snmalloc	45991616	33554432	574895200	94534066	idealloc	256191312	0.042
						real	74727936	0.014
espeak（984）	glibc	1549032	675832	14999513	3068195	idealloc	1638008	0.048
						real	1622136	0.036
	jemalloc	1799744	786432	17623959	3565175	idealloc	1806344	0.039
						real	8480992	0.084
	mimalloc	2455104	1048576	24551040	4855554	idealloc	2459656	0.024
						real	4652672	0.032
	snmalloc	2520656	1048576	25206560	4975939	idealloc	2528784	0.024
						real	16728144	0.054
java-scimark2（11261）	glibc	5358936	528376	50939916	9608547	idealloc	8659120	0.174
						real	6711904	0.167
	jemalloc	5428672	524288	51572384	9720874	idealloc	9108328	0.222
						real	22833600	0.174
	mimalloc	5526976	524288	52506272	9885714	idealloc	7639624	0.149
						real	11009664	0.180
	snmalloc	5953440	524288	56557680	10598910	idealloc	8272960	0.147
						real	16728064	0.190

再次，图9承载了关于这一现象的一些信息：在 BA 相对于其自身上界表现最差的工作负载中，BA 遭受的挫败程度最大（`bullet`，`java-scimark2`，`compress-gzip`）

3. 最令人惊讶的发现是，在超过 65% 的情况下，BA 在页内碎片方面优于或不逊于分配器。

BA 的表现有多好，似乎再次与其完工时间相对于上界的情况相关（图9，右半部分）。`bullet` 和 `dcraw-mimalloc`是最明显的例子，但仔细观察会发现，这一现象在所有工作负载中都普遍存在

我们希望以上观点足以启发未来的研究。如果 2DBP 碎片确实会产生影响常驻集大小，然后可以设想基于 BA 计算出的“最优”放置方案，对各个工作负载进行基于性能剖析的优化。在适用的领域，也可以尝试针对完工时间开展类似工作。

6 相关工作

Wilson 等人撰写了关于动态存储分配（DSA）及碎片化核心作用的开创性论述 [18]。Johnstone 和 Wilson 首次研究了基于常驻集大小（RSS）的碎片化定义 [6]。Berger 等人表明，现代内存分配器在基于 RSS 的碎片化方面表现尚可 [1]。Maas 等人提出了一种新颖的碎片化定义，将内存立即重用的可能性纳入考量 [13]。Powers 等人和 Maas 等人提出了相当非传统的碎片化处理方法 [12, 14]。

表3。（接表 2）

工作负载（#作业）	分配器	L	h_{max}	Robson 上界	BA 界限	设置	完工时间	碎片化
压缩-gzip (22963)	glibc	318304	32832	2387728	572349	idealloc	627600	0.465
						real	339776	0.210
	jemalloc	383400	40960	2937213	690939	idealloc	693608	0.408
						real	1265472	0.450
	mimalloc	383400	40960	2937213	690939	idealloc	774520	0.397
						real	2567808	0.306
	snmalloc	383568	40960	2938500	691222	idealloc	816464	0.416
						real	16744448	0.507
dcraw (63)	glibc	81561656	81526776	1071751586	171607522	idealloc	81588600	0.000
						real	81564744	0.000
	jemalloc	83922104	83886080	1104495793	176573935	idealloc	83931320	0.000
						real	107294064	0.000
	mimalloc	83922104	83886080	1104495793	176573935	idealloc	251694312	0.333
						real	285212288	0.000
	snmalloc	134253760	134217728	1812425760	282476244	idealloc	134262976	0.000
						real	301798384	0.000
系统-libxml2 (200512)	glibc	153984	72720	1243425	306717	idealloc	172784	0.267
						real	200512	0.371
	jemalloc	169264	81920	1381357	337742	idealloc	195376	0.265
						real	891584	0.551
	mimalloc	169264	81920	1381357	337742	idealloc	193648	0.268
						real	3092480	0.420
	snmalloc	218464	131072	1856944	442691	idealloc	202521	0.199
						real	945232	0.490
tjbench （34618）	glibc	7607792	7307256	86732245	15959249	idealloc	7637200	0.007
						real	7612448	0.004
	jemalloc	7674960	7340032	87522768	16094960	idealloc	7693960	0.006
						real	13691920	0.006
	mimalloc	7707728	7340032	87896444	16158532	idealloc	7726728	0.006
						real	8830976	0.005
	snmalloc	8756320	8388608	100697680	18365011	idealloc	8775312	0.005
						real	16711728	0.006

在理论方面，Robson 计算出了适用于任意策略的一般最坏情况碎片化界 [15, 16]，以及针对最佳适配和首次适配策略的更紧界 [17]。Garey 和 Johnson 指出，最优放置是 NP 困难的 [5]。Chrobak 和 Ślusarek 将动态存储分配（DSA）表述为一个二维装箱问题（2DBP）实例 [3]。鉴于我们聚焦于二维装箱问题（2DBP），因此不讨论图着色等其他表述方式 [8]。

7 结论

本文聚焦于探讨动态内存分配的理论文献，旨在借助该领域的最先进成果，为真实系统带来优势。本文建立在一项研究的基础上，该研究证明，二维矩形装箱能够有效表征工作负载与分配器的交互。我们将这项工作进一步拓展：（i）实现目前已知最适合对动态内存分配建模的装箱算法；（ii）从完工时间和碎片化两方面，将该算法的产出与四种现代内存分配器进行比较。我们希望这项展示能激发人们对理论与实践相结合的内存管理方式的进一步兴趣。

致谢

若非 Buchsbaum 等人于 2003年 发表的原始 BA论文，本研究便无从开展。 [2] 因此，我们感谢 Adam Buchsbaum、Howard Karloff、Claire Mathieu、Nick Reingold 和 Mikkel Thorup 的贡献。此外，我们也感谢 Paul Wilson、Mark Johnstone、Michael Neely 和 David Boles，他们启发我们从第一性原理出发研究动态存储分配。 [18]

我们最初提交版本的所有改进，都得益于 ISMM 审稿人和 Shepherd 提供的反馈。我们尤其感谢 Shepherd 指引我们完成最后阶段的工作，也感谢 Erez Petrank 教授迅速解答我们的疑问。感谢 ISMM 组织委员会和程序委员会为我们提供了理想的平台来展示这项工作。

本研究获得希腊研究与创新基金会（HFRI）第三轮博士奖学金项目资助（奖学金编号：61/512200），并获得欧盟“地平线2020”研究与创新计划资助（资助协议编号：101021274）。

参考文献

[1] Emery D. Berger、Benjamin G. Zorn 和 Kathryn S. McKinley。2002 年。《重新审视定制内存分配》。SIGPLAN Not. 37, 11（11 月 2002 1 12 10 1145 583854 582421），–。[2] https://doi.org/ 。/。Adam L. Buchsbaum、Howard Karloff、Claire Kenyon、Nick Reingold 和 Mikkel Thorup。2003年。《动态存储分配中的 OPT 与 LOAD》。载于第三十五届 ACM 计算理论年度研讨会论文集（美国加利福尼亚州圣迭戈）（STOC '03）。计算机协会，纽约，NY，美国，556–564。https://doi.org/ 10 1145 780542 780624 。/。Marek Chrobak 和 Maciej Ślusarek。[3] 1988。《关于一个与动态存储分配相关的装箱问题》。RAIRO-Theoretical Informatics and Applications 22, 4（1988），487–499。[4] Jason Evans。2006。《适用于 FreeBSD 的可扩展并发 malloc (3) 实现》。载于 BSDCan 会议论文集，加拿大渥太华。[5] Michael Garey 和 David S. Johnson。1979。《计算机与难解性：NP 完全性理论指南》。W. H. Freeman and Company。[6] Mark S. Johnstone 和 Paul R. Wilson。1998。《内存碎片化问题：解决了吗？》。载于第一届国际内存管理研讨会论文集（加拿大不列颠哥伦比亚省温哥华）（ISMM '98）。计算机协会，纽约，NY，美国，26 36 10 1145 286860 286864 –。[7] https://doi.org/ 。/。2010 Jukka Jylänki。。《装箱的一千种方法——二维矩形装箱的实用方法》。取自 http://clb.demon.fi/files/RectangleBinPack.pdf（2010）。[8] H.A. Kierstead。1991。《一种用于动态存储分配的多项式时间近似算法》。Discrete Mathematics 88, 2（1991），231–237。https://doi.org/10.1016/0012-365X(91)90011-P

[9] Christos P. Lamprakos、Sotirios Xydis、Francky Catthoor 和 Dimitrios Soudris。2023。《将分配器视为装箱求解器，揭开碎片化之谜》。arXiv: 2304 10862 [ ] . cs.PL Daan Leijen、[10] Benjamin Zorn 和 Leonardo de Moura。2019。《Mimalloc：自由列表分片的实践》。载于《编程语言与系统》，Anthony Widjaja Lin（编）。Springer International Publishing，Cham，244–265。[11] Paul Liétar、Theodore Butler、Sylvan Clebsch、Sophia Drossopoulou、Juliana Franco、Matthew J. Parkinson、Alex Shamis、Christoph M. Wintersteiger 和 David Chisnall。2019。《Snmalloc：一种消息传递分配器》。载于 2019 年 ACM SIGPLAN 国际内存管理研讨会论文集（美国亚利桑那州凤凰城）（ISMM 2019）。计算机协会，纽约，NY，美国，122–135。https://doi.org/ 10 1145 3315573 3329980 。/。[12] Martin Maas、David G. Andersen、Michael Isard、Mohammad Mahdi Javanmard、Kathryn S. McKinley 和 Colin Raffel。2020。《面向 C++ 服务器工作负载的基于学习的内存分配》。计算机协会，纽约，NY，美国，541–556。https://doi.org/ 10.1145/3373376.3378525[13] Martin Maas、Chris Kennelly、Khanh Nguyen、Darryl Gove、Kathryn S. McKinley 和 Paul Turner。2021。《面向仓储规模计算机中非移动内存分配器的自适应大页释放》。载于 2021 年 ACM SIGPLAN 国际内存管理研讨会论文集（线上，加拿大）（ISMM 2021）。计算机协会，纽约，NY，美国，28–38。https://doi.org/ 10.1145/3459898.3463905[14] Bobby Powers、David Tench、Emery D. Berger 和 Andrew McGregor。2019。《Mesh：面向 C/C++ 应用程序的紧凑型内存管理》。载于第 40 届 ACM SIGPLAN 编程语言设计与实现会议论文集（美国亚利桑那州凤凰城）（PLDI 2019）。计算机协会，纽约，NY，美国，333 346 10 1145 3314221 3314582 –。https://doi.org/ 。/。[15] 1971 J. M. Robson。。《动态存储分配所需存储容量的估算》。J. ACM 18, 3（1971 年 7 月），416–423。https://doi.org/ 10 1145 321650 321658 。/。J. M. Robson。。《关于动态存储分配的若干函数的界》。[16] 1974 J. ACM 21, 3（1974 年 7 月），491–499。https://doi.org/ 10.1145/321832.321846[17] John M Robson。1977。《首次适配与最佳适配存储分配策略的最坏情况碎片化》。Comput. J. 20, 3（1977），242–244。[18] Paul R. Wilson、Mark S. Johnstone、Michael Neely 和 David Boles。1995。《动态存储分配：综述与批判性评述》。载于《内存管理》，Henry G. Baler（编）。Springer Berlin Heidelberg，柏林，海德堡，1–116。

收稿日期：2023年3月3日；录用日期：2023年4月24日