arXiv:2302.00635v1 [cs.DC] 24 Jan 2023

分布式业务应用中的共享 SAT 求解器与 SAT 内存

Sergejs Kozlovičs

拉脱维亚大学数学与计算机科学研究所，Raina 大道 29 号，里加，拉脱维亚，LV-1459 sergejs.kozlovics@lumii.lv

摘要。 我们提出了一种软件架构，其中 SAT 求解器作为分布式业务应用的共享网络资源。多个 SAT 求解器可以并行运行于专用硬件（多处理器系统或配备特定图形处理器的系统）上，也可以运行于云端。为避免网络节点之间复杂的消息传递，我们提出了共享 SAT 内存这一新概念，多个不同的 SAT 求解器和实现业务逻辑的模块都可以对其进行读写访问。因此，我们的架构能够通过现有的高级编程语言，轻松生成、多样化并求解 SAT 实例，而无需考虑网络。我们通过将整数分解问题转换为 SAT 的用例，展示了该架构的应用。

关键词： SAT · 分布式应用 · 软件架构 · 整数分解

1 引言

布尔可满足性问题（SAT）有许多实际应用，例如电路设计、模型生成与验证、规划、软件包管理、程序分析以及其他约束满足问题[7,22,9,13]。由于 SAT 是 NP 完全问题，对于 NP 类中的任何其他问题，都存在多项式归约，因此 SAT 堪称 NP 类的“灵丹妙药”[1]。即使某些 NP 问题尚不确定是否为 NP 完全问题（例如整数分解或图同构），在找到特定的高效算法（如果存在）之前，将其归约为 SAT 也不失为一种合理的权宜之计。

虽然目前尚无已知的 SAT 多项式算法，我们也不知道这样的算法是否存在（P=NP 问题），但人们已提出众多技术来高效求解 SAT。这些技术包括启发式方法、冲突驱动子句学习、回跳、随机重启、消息传递和机器学习 [18,6,19]。因此，最先进的 SAT 求解器能够找到变量赋值

[1] 不过，要找到将问题归约为 SAT 的最直接低次多项式归约，可能颇具挑战。

适用于包含数万个变量和子句的 SAT 实例，这意味着许多实际的可满足性问题应用如今已可求解。

有些求解器针对单核 CPU 运行进行了优化（例如 Glusoce 和 GSAT），而另一些则会利用特定硬件，例如 HordeSAT 会使用多个 CPU 核心 [3,11,4]。在 IMCS[2], ，我们开发了自己的求解器 QuerySAT，它在内部使用图神经网络，并且需要图形处理器[3]才能获得更好的性能 [21]。

使用分支的求解器（例如 MapleSAT[4] 和 Lingeling[5]）可以与并行求解器配合使用，后者能够将单个 SAT 实例拆分为多个子任务并行执行。此外，还可以针对同一个 SAT 实例并行启动多个不同的求解器，希望其中一个能更快地找到解。我们还可以组合不同的求解器，从而构建混合求解器，例如，可以用第三方求解器替换 QuerySAT 中的查询机制。

由于 SAT 求解器的计算开销很大，而且通常需要专用硬件，我们考虑将 SAT 求解器作为共享资源提供，并将其部署在专用的本地硬件或云服务器上。本文将介绍以下任务：

1. 将 SAT 求解器作为共享资源集成到分布式业务软件中。2. 优化多个并行 SAT 求解器之间的通信（例如，用于多样化和学习子句交换）。

我们的想法是引入共享 SAT 内存，使所有参与模块（SAT 求解器和业务软件组件）都能够访问它。

下一节将介绍相关术语，并引入 Web 内核这一概念——它是分布式应用的操作系统内核类比。Web 内核将充当 SAT 求解器、业务逻辑单元和 SAT 内存之间的通信代理。随后，我们将介绍实现方面的重要细节。具体而言，我们将展示如何实现 SAT 内存，以及如何在多个共享硬件单元上管理 SAT 求解器。在第 5 节中，我们将介绍一个基于整数分解的使用场景。最后，我们将讨论相关工作，并概述未来的研究方向。

2 定义

分布式应用程序是指在多台自主计算机上运行，并通过网络相互通信的程序。通常，它由两个独立的程序组成：后端（服务器端软件）和前端

[2] 拉脱维亚大学数学与计算机科学研究所[3] 例如 Nvidia T4 16GB 图形处理器（用于我们的实验）[4] MapleSAT 有多种配置。它基于 MiniSAT [10]；参见 https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/maplesat.[5] https://github.com/arminbiere/lingeling

（客户端软件）。不过，也可能包含多个服务器节点和客户端节点。

Web 计算机是一种抽象，通过提供单个目标计算机 [15,16] 的假象，简化了分布式应用的开发。Web 计算机将网络以及多用户/多进程管理抽象出来。因此，分布式应用可以像传统桌面应用一样创建，即只需关注一个程序、一台目标计算机和一个用户。

Web 计算机由以下主要部分组成：

– **数据存储器**（或 Web 内存），在网络节点（例如客户端和服务器）之间共享；由于它会持续自动且透明地同步，每个网络节点都可以访问 Web 内存，就像它是直接连接在该节点上一样；– **指令存储器**（或代码空间），用于存放可执行/可解释的代码；特定代码仅会传送给能够执行/解释该代码的节点（例如，Java 和 Python 代码安装在服务器端并在那里执行，而 JavaScript 代码既可以传送到客户端浏览器并在那里执行，也可以借助 node.js 等 JavaScript 引擎在服务器端传送并执行）；– **Web 处理器**，即能够从代码空间执行特定类型指令的软件模块；– **Web 输入/输出设备**，可以是直接连接到特定物理节点并通过特定应用程序编程接口提供的物理设备（例如打印机），也可以是在单个节点或集群中以软件形式运行的虚拟设备（例如数据库或文件系统）。

请注意，出于安全考虑，Web 计算机将指令存储器与数据存储器分开。因此，它采用的是哈佛架构（而非冯·诺依曼架构）。这样可以防止服务器端代码执行共享数据存储器中的代码，而客户端能够修改共享数据存储器（我们不应信任客户端） [2]。

尽管 Web 计算机仍需要多个物理网络节点才能运行，但 Web 应用程序并不直接访问这些节点，而是通过中间层访问，从而实现单台计算机的假象。由于该中间层的作用类似于操作系统，其参考实现称为$webAppOS$（可访问 http://webappos.org）。

Web 内核是 webAppOS 应用访问 Web 内存、调用指令存储器中的代码（代码将由某个 Web 处理器执行）以及访问 Web 输入/输出设备的抽象层（图 1）。Web 内核将网络通信抽象出来，并营造出一种假象：所有 Web 计算机组件都与发起调用的应用代码位于同一网络节点上。由于调用代码可能需要网络通信，因此称为 Web 调用。Web 调用会由 Web 内核排队并转发给相应的 Web 处理器。所有 Web 调用都是异步的（不过，可以使用现代编程语言的 async/await 功能作为语法糖，简化异步代码）。

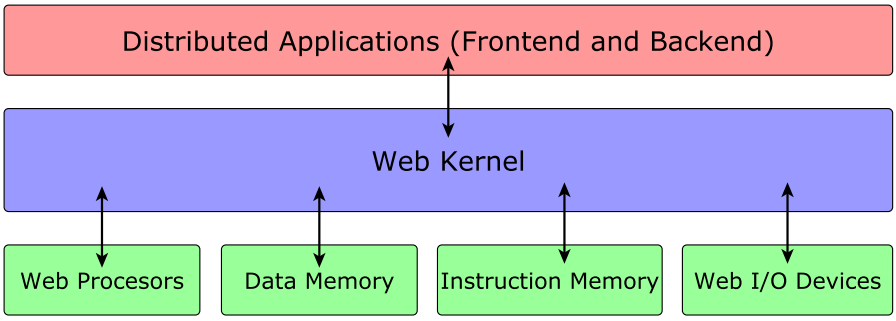

图 1. webAppOS Web 内核作为一种抽象，将网络因素抽离出来

事实上，Web 内存是一种类似图的结构，用于存储对象、对象的状态（属性）以及对象之间的链接。此外，Web 内存还存储元级信息（对象类、继承关系等）。因此，Web 内存类似于 Java 虚拟机使用的面向对象编程内存。Web 内存支持多重继承。

代码空间则将可执行代码存储为**Web 方法**，这些方法可以是函数，也可以是方法（具体取决于编程语言）。每个 Web 方法都由一个与实现无关的全限定名（例如 ClassName.methodName）标识。因此，Web 调用由全限定方法名、Web 内存槽、该槽中的 Web 对象引用（相当于 this 或 self）以及一个 JSON 对象参数指定。每个 Web 调用还必须返回一个 JSON 对象（如果不需要返回结果，至少返回一个空 JSON 对象 {}）。错误（包括网络错误）通过错误字段返回。

同一个类的不同方法可以用不同的编程语言编写（我们将这种方法称为**逐方法粒度**）。显然，所有节点中必须有能够执行给定 webAppOS 应用所用不同类型代码的 Web 处理器。

由于方法可以由不同的 Web 处理器执行（这些处理器可能位于不同的网络节点上），因此无法实现静态链接。将特定 Web 调用替换为相应方法实现的过程称为**Web 链接**。鉴于 Web 链接具有动态且分布式的特性，我们使用**鸭子类型** 机制来定义 Web 对象与接口之间的实现关系。[6]

每个 webAppOS 应用都可供多个用户使用，并且每个用户都可以与同一应用建立多个并发会话。每个这样的应用实例称为一个**Web 进程**；它拥有专属的 Web 内存槽，由**webPid** 标识（本质上是通用唯一识别码）。

[6] 如果接口 I 是一组 Web 方法 m_1、m_2、……，且对象 o 包含在 Web 内存类 C_1、C_2、……中，那么 o 实现 I，当且仅当 $\forall m_i \exists C_j$：ClassName(C_j).methodName(m_i) $\in$ 代码空间。

Web 输入/输出设备通过代码库实现，这些代码库定义了设备特定类，并提供设备特定方法的实现。对于真实硬件，这些方法通过本机驱动程序或操作系统调用访问设备；对于虚拟设备，则提供软件实现。由于所有 Web 进程共享 Web 输入/输出设备，因此设有一个名为**根 Web 内存**的专属 Web 内存槽，用于存储 Web 输入/输出设备（及其状态）。因此，可以通过 Web 内核调用设备特定方法，方式与调用普通 Web 调用相同。不过在内部，设备特定的 Web 调用以软件中断的形式实现——它们在 Web 处理器中优先于普通 Web 调用。这是有意为之，因为我们可能需要中断先前发送给设备的命令，或尽快释放设备供其他用户使用。[7]

Web 内核已作为 webAppOS 的一部分提供。在我们的架构中，它将充当 SAT 求解器、业务逻辑单元和 SAT 内存之间的通信中介，这些组件可以位于不同的网络节点上。

3 集成 SAT 求解器

每个 SAT 求解器都集成为一个虚拟 Web I/O 设备（即实现为一个软件单元）。在根 Web 内存中，不同类型的 SAT 求解器表示为 SatSolver 超类的子类，而每个具体的求解器实例则表示为相应子类的对象。通常，求解器实例的数量对应于空闲 CPU 核心的数量（即未被 Web 处理器占用的核心）；不过，求解器类型和所需硬件（例如图形处理器）也会影响每个节点上的求解器数量。无论如何，只需遍历根 Web 内存，即可轻松检测所有节点上可用的求解器。

3.1 求解器级并发

SAT 实例可以是独立 SAT 问题、更大问题的子任务，也可以是其多样化变体。由于求解 SAT 是一项计算密集型任务，我们有意将每个求解器限制为一次只处理一个 SAT 实例。不过，由于不同用户可能同时调用求解器，因此必须以线程安全的方式实现求解器，并在必要时返回错误属性。

每个求解器都必须提供以下 Web 方法（=Web I/O 设备中断）：

– solve({satMemoryUrl, timeout, diversification})— 使用给定的多样化准则，在指定的 SAT 内存中存储的 SAT 实例上启动求解器（下一节将讨论 SAT 内存，随后的小节将讨论多样化）。

[7] 这意味着 Web 输入/输出设备必须以线程安全的方式实现。

求解器从 SAT 内存中获取子句，自行进行多样化，求解 SAT 实例，并返回值为 SAT、不可满足或未知的 result 属性。当既无法得出满足变量赋值，也无法得出矛盾时（例如，使用基于启发式方法或随机化求解器时），会返回后一个值。求解器被中断时，也会返回未知值。若结果为 SAT，则会在 model 属性中返回变量赋值（一个 JSON 数组）。

假设求解器已经在求解另一个 SAT 实例。在这种情况下，求解方法必须在错误属性中返回忙碌标志，或者最多等待 timeout 秒，直到求解器可用。

– 暂停({}) — 请求求解器尽可能快地中断搜索，并允许之后恢复搜索（例如，必须保留调用栈）；– 恢复({}) — 恢复此前中断的搜索；– 取消({})— 永久中断当前正在处理的 SAT 实例的搜索。因此，当前正在运行的求解 Web 调用必须返回未知。

我们假设，发生任何错误时（例如，要求未暂停的求解器恢复，或超出内存/CPU 限制），都会返回错误属性。此外，由于上述 Web 调用属于 Web I/O 设备中断，因此调用方 Web 进程的 webPid 始终会被隐式传递。这样，求解器便可以验证暂停 Web 调用是否源自调用过 solve 的同一 Web 进程，例如，这是一项有用的安全预防措施。

3.2 多样化

以下多样化设置可传递给求解 Web 调用。我们从 HordeSAT 基于组合的求解器中借鉴了这些设置，并将其转换为 JSON 语法，以符合 Web 内核 [4] 的要求。每种求解器类型都可以采用不同的方式实现多样化。

– 排名 — 此特定求解器在所有正在处理当前 SAT 问题的求解器中的索引；举例来说，排名可用于初始化随机数生成器种子；– 数量 — 正在处理当前 SAT 问题的求解器数量；– 相位 — 一个可选的 JSON 对象，包含名为 xi 的属性，例如 x5, $1 \leq i \leq$#(SAT 变量)；对于变量 x_i，phases[xi] 表示搜索期间优先尝试的布尔值（即变量的“相位”）。

请注意，虽然根 Web 内存中的每个求解器都被视为单个实例，但它内部可以依赖多个并行求解器（例如，多个 GPU 子程序）。这类“内部求解器”对 Web 内核不可见，也无法按照上述方式进行多样化。

3.3 加入并行求解器

求解器可以将任务拆分为多个子任务，并通过 parallelize（Web 内核提供的内置 Web 调用）将它们转发给多个并行求解器。parallelize Web 调用接受一个子 Web 调用列表 L，并在可能时并行调用这些子 Web 调用。从技术上讲，parallelize 会在 Web 内存中存储一个计数器，并为来自 L 的每个 Web 调用追加 join method（同样是内置方法）的调用。join method 会递增计数器，并存储子 Web 调用的返回值。当计数器达到 L 的长度时，parallelize 会返回子调用返回值的列表，供父求解器处理。

4 实现 SAT 内存

与 SAT 求解器一样，每个 SAT 内存实例也表示为一个虚拟 Web I/O 设备。采用这种设计是为了避免引入新的内存类型，以免使 Web 内核架构变得复杂。与 Web 内存不同，SAT 内存并非用于所有 Web 进程。此外，根据用于表示子句的范式（例如合取范式或代数范式），SAT 内存可能有不同的类型[8]。此外，某些 Web 方法（例如范式之间的转换器）可能需要同时使用多种类型的 SAT 内存。

本节重点介绍子句以合取范式（CNF）给出的 SAT 内存，因为合取范式用于 DIMACS 文件格式——这是多数 SAT 求解器支持的事实标准。此外，它也用于 SAT 竞赛[9]。

包含合取范式子句的 SAT 内存实例，表示为根 Web 内存中 SatCnf Web 类的实例。与预先配置的 SatSolver 实例不同，SatCnf 对象可以动态创建和删除。这种方法类似于将外部数据存储（例如 USB 驱动器）连接到 PC 并将其卸下的过程。

CNF SAT 内存由自由变量 x_1、x_2、.....、x_n 以及以下形式的 CNF 子句组成：(l_1、l_2、.....、l_k)，其中每个 l_i、$1 \leq l_i \leq n$都是表示文字的整数：

$$x_{l_i} \text{ if } l_i > 0$$
$$\neg x_{l_i} \text{ if } l_i < 0$$

自由变量的数量可以在运行时增加。例如，在将布尔公式转换为 CNF 时，可以引入额外变量以避免公式呈指数级增长（从而得到可满足性等价公式，而非等价公式）。此外，通过引入辅助变量，可以更容易地表示特定的公式片段。

通过指定布尔变量的初始数量来创建 SatCnf 对象。

[8] 本文仅讨论 CNF。[9]
https://satcompetition.github.io

SatCnf Web 类以线程安全的方式实现以下 Web 方法：

– addVariable({}) - 添加一个新的自由变量，并返回其索引（正整数）；– addClause({clause}) - 将新的合取范式子句作为整数数组添加（l_1、l_2、...），这些整数表示文字，其中 $1 \leq |l_i| \leq$ #（自由变量）。为避免重复子句，我们会在添加子句前对文字排序。不过，也可以使用布隆过滤器。– clauses({}) - 返回当前存储在 SAT 内存中的子句数组，例如：`[[1,-2,3],[4,-5]]`；– fork({detach}) - 返回一个新的 SatCnf 实例，该实例保留当前 SAT 内存中的子句，并且只能向分叉出的 SAT 内存添加新子句。[10] 如果 detach=为 true，则对原始 SAT 的后续更改不会反映到分叉中。否则，分叉将与原始实例共享自由变量，并接收子句更新。分叉有助于定义可传递给 solve 的 SAT 子任务。例如，可以将特定变量赋值作为单文字子句添加。也可以将学习到的子句（在基于 CDCL 的求解器中）添加到分叉出的内存中。

请注意，我们没有用于读写 SAT 内存中变量赋值的方法。尽管这看起来有悖直觉，但这样可以防止不同求解器执行不同变量赋值时发生冲突（例如，解不唯一或只有部分解的情况）。

虽然上述方法便于构造初始 SAT 实例，但通过 Web 调用这些方法会产生一定的序列化/反序列化开销；当多个求解器访问 SAT 内存时，这种开销并不可取。此外，这也会对基于组合的求解器产生负面影响，因为它们需要在运行时交换学习到的子句。

为了尽可能降低 Web 内核开销，我们引入了对 SAT 内存的直接访问，这与传统硬件计算系统中的 DMA[11] 功能类似。我们通过 WebSocket[12]实现了这一功能。

SAT 内存实例的 WebSocket 可通过 directUrl 属性中提供的地址访问。所有 WebSocket 连接均由 SatCnf 类的内部集线器管理。通过一个 WebSocket 添加新变量或子句时，集线器会将其转发到其他 WebSocket。这样，使用同一 SAT 内存的所有其他节点就能高效地交换新变量和子句。

简而言之，WebSocket 中使用的协议采用与 SatCnf 方法对应的二进制消息。不过，addVariable 和 addClauses 既可以发送，也可以

[10] 从技术上讲，为了避免复制子句，与原始 SAT 内存的公共部分会被提取出来，并且只存储一次。[11] 直接内存访问[12] HTTP 协议的扩展，用于通过网络实现高效的双向通信

接收。接收到这些消息时，客户端应将其视为来自其他求解器的变量和子句同步消息。

除了 addVariable 消息外，我们还引入了以下消息（由客户端发送）：

– 锁定变量（因此，最后使用的变量索引保持不变）；– 添加多个自由变量；– 解锁变量。

当需要添加多个自由变量并使用其索引构造子句时，这些方法可以避免多次往返通信。变量只能在较短时间内保持锁定，因为其他 SATMemory 用户发送的 addVariable 调用将一直等待，直到变量解锁。

5 使用示例

本节以整数分解问题及其向 SAT 问题的转换作为使用案例。为简单起见，我们考虑由恰好两个长度相同的整数因子 (> 1) l 构成的乘积。如果两个因子都是素数，则我们的转换会生成一个具有唯一解的 SAT 实例。

输入：乘积的长度为 l and $2l$ 位。
输出：SAT 内存中的一个 SAT 实例。

SAT 内存填满后，将调用根 Web 内存中第一个可用的基于组合的 SAT 求解器的 solve web 方法，以获得 SAT 赋值。该赋值的前 $2l$ 个布尔值对应于所讨论的两个因子的各个位。

5.1 Karatsuba 算法

为了渐近地最大限度减少生成的变量和子句数量，我们采用 Karatsuba 乘法算法，该算法基于递归的“分治”方法 [14]。

如果 $u = (u_{2n-1} \dots u_1 u_0)$ 和 $v = (v_{2n-1} \dots v_1 v_0)$ 是 $2n$ 位整数，它们可以写成

$$u = 2^n U_1 + U_0 \text{ and } v = 2^n V_1 + V_0$$

（最高有效位在左侧）。然后

$$uv = (2^{2n} + 2^n)U_1 V_1 + 2^2 (U_1 - U_0)(V_0 - V_1) + (2^n + 1)U_0 V_0. \qquad (1)$$

Karatsuba 算法会递归应用于式（1）中三次 n 位数乘法中的每一次。由于该算法要求每一层的位数均为偶数，因此我们从将 u 和 v 表示为 2^k 位数开始（在前面补上缺失的前导零）。

5.2 初始化 Web 内存

在 Web 内存中，一个 m 位整数将表示为由 m 个文字组成的有序列表：对于正文字 x_i，第 i 位的值对应于 x_i 的值（假为 0，真为 1）；对于负文字 $\neg x_i$，第 i 位对应于 x_i 的否定。因此，同一个变量可以重复用作正文字或负文字。

由于我们已知每个因子中的位数（$l = 2^k$，原因如上所述），因此我们使用 $2l = 2^{k+1}$ 个自由变量初始化一个新的 SAT 内存实例。这样，因子 u 和 v 就表示为文字列表 x_{l-1}、$\ldots x_2$、x_1 和 x_{2l-1}、$\ldots x_{l+1}$、x_l。

5.3 生成 SAT 公式

SAT 公式生成过程会引入额外的整数（必要时添加额外的自由变量）。这些额外整数是通过对现有整数应用变换函数得到的。

变换函数构造为布尔表达式，用于关联源整数和目标整数的文字。布尔表达式由布尔基本运算与、或和非构成，也可以使用辅助运算布尔异或、（包含 3 个元素的）多数函数、蕴含和等价；这些运算都可以通过布尔基本运算轻松实现。

以下是用作构建模块的变换函数列表。L_1、L_2、.....是由文字组成的列表，用于表示相应的源整数或目标整数，分别记为 I_1、I_2、.....：

- L_1.negation()$\rightarrow L_2$：将 I_1 转换为以 $|L_1|$位二进制补码表示法表示的 $I_2 = -I_1$。具体来说，我们对 I_1 的所有位取反（即对 L_1的所有文字取反），然后加 1；通过半加器公式（来自数字逻辑），将 I_1 的第 i位与第 i位进位位相加。- L_1.sum_with(L_2) $\rightarrow L_3$（假设 $|L_1| = |L_2|$）：将 I_1 和 I_2 转换为以 $|L_1|$位二进制补码表示法表示的 $I_3 = I_1 + I_2$。忽略 I_3 的溢出位（如果有）。具体来说，我们使用 1 个半加器公式和 $L_1 - 1$ 个加法器公式。- L_1.product_with(L_2) $\rightarrow L_3$（假设 $|L_1| = |L_2| = 2^k$、$|L_3| = 2 \cdot 2^k$）：根据方程（1），将两个以 2^k位二进制补码表示法表示的 2^k位整数 I_1 和 I_2 转换为以 2^{k+1}位二进制补码表示法表示的 2^{k+1}位整数 $I_3 = I_1 \cdot I_2$。具体细节如下：
  - 通过求和_和取反来表示两个数之差；
  - 根据方程（1）确定内部 Karatsuba 乘积 $(U_1 - U_0)(V_0 - V_1)$ 的符号；
  - 将符号位从 2^k 位二进制补码表示法扩展至 2^{k+1} 位二进制补码表示法。

通过应用上述变换函数，我们可以构造一个将初始 $2l$ 自由变量与 $2l$ 文字绑定的（非合取范式）SAT 公式

积（显然，最顶层的变换会将积_应用于两个初始列表 x_{l-1}、 $\ldots x_2$、 x_1 和 x_{2l-1}、 $\ldots x_{l+1}$、 x_l）。我们的变换函数实现会引入额外的变量，并为重复出现的公式重复使用文字。

构造公式后，我们使用以下模式将其转换为合取范式:

– 使用与、或和非重新表述辅助布尔函数（例如异或和多数函数）；– 应用德摩根定律，以消除提取到括号外的否定；– 为内部合取引入新变量和等价关系。例如，给定公式 ($x_1 \vee$(x_2&x_3&x_4))，我们用一个新变量 x_5 替换 (x_2&x_3&x_4)，并添加等价关系 $x_5 \equiv x_2$&x_3&x_4；– 为右侧包含两个以上文字的等价关系（来自前一种模式）引入新变量，例如，用 x_6 表示 x_3&x_4（相应的等价关系为 $x_6 \iff x_3$&x_4）；–借助真值表，根据含三个文字的等价关系（例如上文的 $x_5 \iff x_2$&x_6 和 $x_6 \iff x_3$&x_4 ）构造合取范式子句。

随后，我们通过附加与乘积位对应的文字合取来扩展合取范式。如果乘积位为 1，则添加对应的单文字子句；如果该位为 0，则将该文字取反。

为了排除其中一个因子为 1 的情况，我们添加两个子句（$x_{l-1} \vee \ldots x_3 \vee x_2$）和（$x_{2l-1} \vee \ldots x_{l+2} \vee x_{l+1}$），这表示每个因子至少有一位（不包括最右侧的位 x_1 或 x_l）为 1。

我们通过添加两个单文字子句 $\neg x_{l-1}$ 和$\neg x_{2l-1}$来排除负因子，这表示每个因子的最高有效位均为 0（在二进制补码表示法中，非负数的该位必须为 0）。

为了在乘积由两个素数组成时指定唯一解，我们采用条件 $u - v \geq 0$。该条件对应的子句可用与主 SAT 公式类似的方法得到。

上述变换函数（应用于以文字列表表示的整数）可以轻松映射到任何基于面向对象编程的语言。对 Web 内存和 SAT 求解器（通过相应 Web 输入/输出设备的 Web 方法）的访问也基于面向对象编程。因此，可以完全以面向对象的方式生成 SAT 实例，将其放入 Web 内存，然后传递给求解器，而无需考虑网络。此外，我们的构件还可作为构建更大型变换函数的基础。

当前版本的 SAT 生成器（仍在开发中）可在 IMCS 的 GitHub 页面 https://github.com/LUMII-Syslab/sat-generator 上获取。

6 相关工作

先进启发式方法、搜索空间剪枝以及对 SAT 子句的智能预处理，极大提升了 SAT

求解器在单个中央处理器上运行时的性能。这类求解器主要基于 DPLL[13] 变体 CDCL[14] 和 VSIDS[15]。在并行化方面也开展了大量研究，主要有三种方法：1）以不同设置运行多个顺序 SAT 求解器；2）将搜索空间划分为（互不相交的）子任务，并在这些子任务上运行多个求解器实例；3）采用基于组合的方法，让不同类型的求解器并行工作，并能够对其进行多样化处理以及在它们之间交换学习到的子句 [12,4]。

在我们的方法中，每个 SAT 求解器都被视为串行求解器（即使其内部使用了并行化）。不过，也可以使用多个求解器（类型相同或不同），而 Web 内核（webAppOS 的一部分）可将它们集成起来，以支持不同类型的并行化。

SAT 内存是我们提出的一种创新方案，旨在提供专为求解 SAT 问题而设计的分布式内存。在 SAT 内存的实现中，我们采用集中式方法（而非分布式方法）来处理一致性问题 [17]。不过，我们的双向 WebSocket 允许 Web 内存客户端交换子句，这是迈向混合方法的一步。此外，这也使我们能够避免使用回调。直接访问 SAT 内存的方式，类似于在 Linux 和 BSD 系统中访问共享字符设备（位于 /dev 目录中）。

我们面向求解器和 SAT 内存的 API，与 HordeSAT 使用的 Portfolio Solver Interface 以及 MiniSAT 的外部接口 [4]类似。不过，我们将与 SAT 求解相关的方法和用于访问 Web 内存中子句的方法分开。此外，我们的两个 API（分别面向求解器和 SAT 内存）都遵循纯粹的面向对象编程原则 [23]。另外，HordeSAT 使用同一个 setSolverInterrupt 调用来暂停搜索和取消搜索（MiniSAT 接口不具备此功能）。相比之下，我们使用不同的方法，因为在前一种情况下必须保留调用栈，而在后一种情况下则可以释放求解器使用的所有资源。

整数分解问题是生成困难 SAT 实例的方法之一。ToughtSAT[16] 是一套 SAT 生成器，其中包括一个用于生成整数分解问题 SAT 实例的生成器。它曾用于 SAT 竞赛 2019 [5]。遗憾的是，尽管该生成器生成的 SAT 实例简洁，但在处理某些因子时会报错，并且生成的 SAT 实例可能存在多个解。

一种更通用的 SAT 实例生成方法是使用 SAT 编译器。Picat-SAT、FznTiny 和 MiniZinc-SAT 都是很好的代表 [24,13,20]。尽管一些 SAT 编译器会对整数使用二进制对数编码，但已知的通用 SAT 编译器中，没有一种使用 Karatsuba 算法来表示整数乘积。

[13] 戴维斯–普特南–洛格曼–洛夫兰算法[14] 冲突驱动子句学习[15] 变量状态独立衰减和[16] 2022年，该项目可在 https://github.com/joebebel/toughsat.git 获取；较早的参考资料指向 https://toughsat.appspot.com/，该地址现已无法访问。

我们转换为合取范式时，采用了证明 SAT 可归约为 3-CNF-SAT 时所使用的构造合取范式子句的模式 [1]。

7 结论

我们提出了一种软件架构，通过 webAppOS Web 内核在分布式环境中共享 SAT 求解器。在我们的方法中，可以将任何类型的 SAT 求解器用作共享求解器；唯一的要求是将其表示为 SatSolver 子类，并实现第 3.1 节中的应用程序编程接口（Web 方法）（通常只需编写少量粘合代码即可）。我们的设计既适用于单个共享 SAT 求解器，也适用于以不同设置启动的多个 SAT 求解器，或用于并行处理多个子任务的多个 SAT 求解器。

由于 SAT 求解器被表示为独立的 webAppOS I/O 设备，因此既可以本地部署（例如，安装在特定硬件上），也可以基于云端。这为以服务形式提供的商业 SAT 求解器带来了机遇。

在我们的架构中，一个新颖的组件是共享 SAT 内存，它既可以作为 webAppOS 虚拟 I/O 设备访问，也可以通过 WebSocket 直接访问。虽然我们考虑了用于存储合取范式子句的 SAT 内存，但该架构也可以推广为支持析取范式（DNF）和代数范式（ANF）中的子句。通过支持多种类型的 SAT 内存，我们的架构允许开发者创建可同时使用多个 SAT 内存的 Web 方法（例如，用于数据转换）。

我们还设想进一步扩展 SAT 内存。例如，基于置信传播和调查传播的 SAT 求解器需要存储与 SAT 变量和子句相关的实数权重（称为“警告值”） [8]。

致谢 本研究得到拉脱维亚科学委员会资助，项目编号 2021/1-0479“使用深度神经网络进行组合优化”。

这篇预印本未经同行评审，也未进行任何投稿后的改进或修正。本文的正式版本发表于 CCIS 第 1598 卷《数字商务与智能系统：第十五届波罗的海国际会议，Baltic DB&IS 2022，拉脱维亚里加，2022年7月4日至6日，会议论文集》，可在线访问：https://doi.org/10.1007/978-3-031-09850-5_14。

参考文献

1. Aho, A.V.、Hopcroft, J.E.、Ullman, J.D.：《计算机算法的设计与分析》。计算机科学与信息处理 Addison-Wesley 系列，Addison-Wesley 出版公司，马萨诸塞州雷丁（1974） 2. Andrews, M.、Whittaker, J.A.：《如何攻破 Web 软件：Web 应用程序和 Web 服务的功能与安全测试》。Addison-Wesley Professional（2006） 3. Audemard, G.、Simon, L.：《关于 Glucose SAT 求解器》。《国际人工智能工具杂志》**27**（01），1840001（2018年2月）

4. Balyo, T.、Sanders, P.、Sinz, C.：《HordeSat：大规模并行组合 SAT 求解器》（2015 10 1007 978 3 319 24318 4_12 年5月）。https://doi.org/5。/----。 Bebel, J.：《利用 Karatsuba 和 Espresso 对因式分解生成更难的 SAT 实例》。载于：SAT Race 2019 论文集。赫尔辛基大学（2019） 6. Biere, A.、Heule, M.、Maaren, H.v.（编）：《可满足性手册》。人工智能与应用前沿第336卷，IOS Press，阿姆斯特丹；华盛顿特区，第二版。（2021 1250382566 7），OCLC：on。Boulanger, J.L.（编）：《应用于复杂系统的形式化方法：B 方法的实现》。计算机工程系列，ISTE；Wiley，伦敦：新泽西州霍博肯（2014） 8. Braunstein, A.、Mezard, M.、Zecchina, R.：《调查传播：一种用于可满足性的算法》。Random Struct. Algorithms **27**，201–226（2005年9月） 9. Eggersglüss, S.、Eggersglüß, S.、Drechsler, R.：《高质量测试模式生成与布尔可满足性》。Springer，纽约（2012） 10. Eén, N.、Sörensson, N.：《一种可扩展的 SAT 求解器》。载于：Giunchiglia, E.、Tacchella, A.（编）《可满足性测试的理论与应用》。第502–518页。Springer 柏林海德堡，柏林、海德堡（2004） 11. Folino, G.、Pizzuti, C.、Spezzano, G.：《结合遗传算法与局部搜索的 SAT 并行混合方法》。IEEE Transactions on Evolutionary Computation **5**（4），323–334（2001年8月） 12. Hamadi, Y.、Jabbour, S.、Sais, L.：《ManySAT：一种并行 SAT 求解器》。JSAT **6**，245–262（2009年6月） 13. Huang, J.：《约束模型的通用布尔化》。载于：第14届约束编程原理与实践国际会议论文集。第144–158页。CP '08，Springer-Verlag，柏林、海德堡（2008） 14. Knuth, D.E.：《计算机程序设计艺术》第2卷：半数值算法，第2卷。Addison-Wesley，马萨诸塞州雷丁，第3版（1997） 15. Kozlovičs, S.：《Web 计算机及其操作系统：创建 Web 应用程序的新方法》。载于：第15届 Web 信息系统与技术国际会议论文集（2019） 16. Kozlovičs, S.：《webAppOS：为 Web 应用程序开发人员营造单一计算机的错觉》。载于：Bozzon, A.、Domínguez Mayo, F.J.、Filipe, J.（编）《WEBIST 2019：Web 信息系统与技术》。第1–21页。Springer International Publishing（2020） 17. Li, K.、Hudak, P.：《共享虚拟内存系统中的内存一致性》。ACM 计算机系统汇刊 **7**（4），321–359（1989年11月） 18. Malik, S.、Zhang, L.：《布尔可满足性：从理论难解性到实际成功》。ACM 通讯 **52**（8），76–82（2009年8月） 19. Moskewicz, M.、Madigan, C.、Zhao, Y.、Zhang, L.、Malik, S.：《Chaff：设计高效的 SAT 求解器》。载于：第38届设计自动化会议论文集（IEEE 编号01CH37232）。第530–535页（2001） 20. Nethercote, N.、Stuckey, P.J.、Becket, R.、Brand, S.、Duck, G.J.、Tack, G.：《MiniZinc：迈向标准化的约束编程建模语言》。载于：Bessiere, C.（编）第13届约束编程原理与实践国际会议论文集。第529–543页。Springer-Verlag London Ltd.，德国（2007） 21. Ozolins, E.、Freivalds, K.、Draguns, A.、Gaile, E.、Zakovskis, R.、Kozlovics, S.：《目标感知神经 SAT 求解器》。载于：2022 年国际神经网络联合会议（IJCNN）。第 1–8 页。IEEE，意大利帕多瓦（2022年7月）。https://doi.org/10.1109/IJCNN55064.2022.9892733, https://ieeexplore.ieee.org/document/9892733/

22. Rintanen, J.：作为可满足性的规划：启发式方法。人工智能 **193**，45–86（2012年12月）23. West, D.：对象思维。Microsoft professional，微软，华盛顿州雷德蒙德（2004年）24. Zhou, n.f., Kjellerstrand, H.：Picat-SAT 编译器。第 9585 卷，第 48–62 页（2016年1月）