arXiv:2602.06680v1 [cs.PL] 6 Feb 2026

同一引擎，多种档位：不同粒度的并行不动点迭代（扩展版）

Ali RasimKocal[1][0009−0004−9019−0852] ✉、MichaelSchwarz[2][0000−0002−9828−0308]、Simmo Saan[3][0000−0003−4553−1350] 和 Helmut Seidl[1][0000−0002−2135−1593]

[1] 慕尼黑工业大学，德国加兴[2] 新加坡国立大学，新加坡
[3] 塔尔图大学，爱沙尼亚
{ali.rasim.kocal,helmut.seidl}@tum.de
m.schwarz@nus.edu.sg, simmo.saan@ut.ee

摘要。不动点迭代是静态分析器的算法核心。并行化不动点引擎可以显著缩短分析时间。以往的方法通常会预先固定任务粒度，例如以程序线程或过程为单位——这会使引擎永久停留在单一档位。为此，我们提出以参数化任务粒度的方式并行化通用不动点引擎，也就是说，引擎可以在不同档位下运行。我们以自顶向下求解器 **TD** 为基础，扩展其对混合流敏感性的支持，并实现了两种相互竞争的并行化理念；二者都基于任务池，由任务池将任务调度给数量固定的工作线程。两种理念中的任务性质各不相同。在即时方法中，所有任务都访问同一个线程安全哈希表，以维护求解器状态；而在独立 方法中，每个任务都有自己的状态，并通过发布/订阅 数据结构与其他任务交换数据。我们为静态分析框架 Goblint 的不动点引擎实现了这两种理念，并报告了在大型真实程序上的实验结果。

关键词： 抽象解释 · 静态分析 · 不动点迭代 · 并行化。

1 引言

基于抽象解释的静态分析旨在无需运行程序即可推断程序不变式。为此，候选不变式由某个有界格（即域）中的抽象值表示，其中格运算交和并分别对应逻辑合取和析取。随后，对程序的分析会产生一个定义在该域上的方程组，其解对应于程序的归纳不变式。这类分析的设计包括定义并实现

抽象域及相应的抽象传递函数，这些函数对应于控制流图中的边，同时还要规定抽象值之间的关系，例如程序点（在上下文中）或全局变量的抽象值之间的关系。

理想情况下，此类分析的设计者应专注于这些问题，而不必同时关注如何计算方程组解的算法细节。将分析的设计与不动点引擎的实现分离开来至关重要，尤其是在不动点引擎较为复杂时。因此，诸如 [30, 41]这样的不动点引擎，理想情况下应以独立于应用的方式实现，以便重复使用。**TD** [30] 就是这样一种不动点引擎。它适用范围广，因为它不对未知量集合的有限性作任何假设，不需要预先了解方程系统中变量之间的依赖关系（我们称这些变量为未知量，以便将它们与程序变量区分开来），并且会动态检测应当应用扩张/收缩的位置。此外，它支持混合流敏感分析，其中一些未知量以流不敏感的方式分析，而另一些未知量的值则以流敏感的方式确定 [37]。

虽然 Astrée [20] 和 Pikos [15] 等分析器已实现并行化，但 Goblint [42], 等许多其他分析器仍是顺序程序，因此无法利用多个核心加速不动点计算。在此，我们旨在提供通用的并行化策略，不对待分析的程序施加限制，并且能够轻松适配各种分析，尤其是混合流敏感分析，同时支持多种粒度的任务。我们不只支持预先定义的一组粒度，而是通过根未知量来识别任务。对于每个根 x，都会创建一个独立的求解器实例，通过按需驱动的方式探索方程系统，从而在局部求解 x 。为实现分析的并行化，例如在程序线程层面进行并行化，我们将已创建程序线程的端点选作根。对于顺序代码分析的并行化，可选方案包括将过程的端点或特定*if-else* 语句的分支作为根。为尽量减少访问共享数据时的同步开销，理想情况下，不同的根应分别探索互不相交的未知量集合。但在实践中，这一点很难保证。我们提出了两种方法来处理不同求解器任务之间的交互：

– 即时方法将所有求解器数据按其对应的未知量分组，并存储在一个共享的线程安全数据结构中，从而保证不同任务对各个未知量的访问具备事务性。–另一方面，独立方法维护求解器数据的多个本地副本，并引入发布-订阅机制，在不同任务之间共享信息。

本文其余部分安排如下：第2 节通过示例说明如何使用副作用方程组来形式化分析多线程 C语言程序。第3节介绍改进版**TLTD** （顶层**TD**）**TD** ，为我们的并行化奠定基础。第4 节随后介绍我们并行化**TLTD**的两种方法，并通过实验

进行评估，结果见第5节。讨论第6节中关于并行求解器的相关工作后，我们在第7节总结研究结果。

2 背景

线程模块化分析的基本设定（例如 [28, 29], 中所考虑的情形）旨在为每个程序点推断单一不变式。为此，针对每个程序点 v，引入一个未知量，其取值来自候选不变式的某个有界格 $\mathbb{D}$。为每个程序点 v 提供一个右端函数 f_v，用以说明在考虑程序控制流的前提下，如何由其他程序点的不变式得到 v 的不变式。此类未知量称为局部变量。在更精细的设定中，可为每个程序点计算多个不变式，例如依据调用上下文或线程id（或二者）加以区分。在这种情况下，可为每个程序点引入多个局部变量，例如未知量 $\langle v, c\rangle$，对应于上下文 c中的程序点v。为了借助共享数据结构抽象线程间交互，还额外引入一组专门的全局变量。这些未知量的值表示共享数据的不变式，且这些不变式应在整个程序执行过程中始终成立。因此，它们的值被流不敏感地累积。

```
1 int g; 2 void* foo(void*a) { 3 g = (int)a; 4 return
NULL; 5 } 6 int main() { 7 pthread_t t1;int a; 8 g = 0; 9
pthread_create(&t1, NULL, foo, (void *)42); 10 a =g; 11
a =a+1; 12 return a; 13 }
```

图 1：一个多线程 C 程序

考虑图 1中的多线程 C 程序，它将作为贯穿全文的示例。假设我们要以线程模块化的方式执行 区间分析 [8]。针对图 1 中的程序，区间分析试图推断 a 和 g的值的紧确上下界。由于变量g 是共享的，我们为 g引入一个全局未知量，为方便起见，使用相同的名称。作为 g的候选不变式所构成的有界格，我们使用由自然序定义的区间 $\mathbb{I}$ 集合（扩展了 $\emptyset$）。

在我们的示例中，局部变量集合由程序点集合构成。出于技术原因，我们将被调用过程的起点 视为全局变量，因为

这些起点的值并非通过右端项获得，而是通过分散在程序各处的调用点所提供的贡献值获得。一般而言，区间分析会为每个局部程序变量维护一个独立的区间。不过，在我们的极简示例中，函数 a main main和 foo中都只有一个局部变量。因此，在这种情况下，局部变量的候选不变式可以取自与全局变量相同的域 $\mathbb{I}$。下文假设局部变量按照图 1中的程序文本行号命名。

3 求解器 TLTD

令 $\mathcal{X}$ 表示所有未知量的集合。分析运行包括反复计算右端项，直到找到未知量的一个赋值 $\sigma : \mathcal{X} \to \mathbb{D}$，使每个右端项都得到处理。具体而言，这意味着重新计算每个右端项时，只会产生已被 σ涵盖的结果。

本节介绍一个版本**TLTD** 的自顶向下求解器**TD** [30]„ 它采用经过修改的数据结构组织方式，并增加了一个顶层工作集。此修改本身也有益处，同时还是我们实现并行化的基础。其实现见图2、3和4，其中与**TD** 不同的部分均已突出显示，但数据结构修改相关部分除外。

TD 会在分析过程中遇到未知量时立即递归求解。相比之下，**TLTD** 引入了一个主循环（图2），该循环针对工作集 x 中的每个未知量调用函数iterate，以确定 x 的值。最初，`work` 假定包含感兴趣的未知量，通常是`main` ($\langle 13 \rangle$ 的端点（在我们的示例中），它会触发对所有可能产生影响的未知量进行迭代。集合`work` 可以动态地在分析过程中接收更多根未知量，从而使我们能够并行求解未知量。

```
46  当 非（集合.is_为空！ 工作集） 执行（
47    迭代（集合.choose！ 工作集）
48  ） 完成
```

图 2：**TLTD** 的顶层循环

TLTD **算法** 的核心是迭代 函数（图 3）。当已调用它来处理未知量 x 时，迭代 会反复计算x （第 26行）提供的右侧，直到达到不动点。调用`eq` x 提供的右侧是一个高阶函数，用于形式化如何确定 x 的抽象值。它由三个函数参数 get、set 和需求参数化，它们代表不动点引擎的接口。第一个函数参数 get 用于访问其他未知量的值。

示例*1*。 考虑赋值 `a = a + 1;` 图 1 中的第 11 行。接着，该行之后未知量 $\langle 12 \rangle$ 的右侧如下所示：

$$\mathsf{eq}\ \langle 12 \rangle = \mathbf{fun}\ \mathsf{get}\ \mathsf{set}\ \mathsf{demand}\ \to (\mathsf{get}\ \langle 11 \rangle) +^\sharp [1,1] \tag{1}$$

```
1 let rec iterate x =2 let rec query y =3 let dy =查找
data y in 4 if (not (dy.called) && not (is_全局变量 y)) then (
5 iterate y 6 ); 7 dy.influences <- (Set.add x dy.influences);
8 dy.value in 9 let side g contribution =10 let dg =查找
data g in 11 dg.stable <- true; 12 let new_value = 域连接
dg.value contribution in 13 if not (域小于等于关系 new_
value dg.value) then ( 14 dg.value <- new_value; 15 使失稳
g; 16 ) in 17 提升 y =18 let dy =查找 data y in 19
dy.toplevel <- true; 20 工作集 :=Set.add y !work in 21 22 let
dx =查找 data x in 23 dx.called <- true; 24 while not
dx.stable do ( 25 dx.stable <- true; 26 let new_value = eq
x query side promote in 27 if not (域小于等于关系 dx.value
new
30 )
x !w
```

器。

其中， $+^\sharp$ 是域 $\mathbb{I}$ 上编程语言运算符“$+$”的抽象版本。□

程序全局变量的值以流不敏感地 进行分析，并在 $\mathcal{X}$ 中表示为全局变量 未知量（全局变量）。全局变量本身没有右端项，但会接收来自局部变量右端项的贡献值。因此，第二个参数集合提供了一种产生此类贡献值（已调用副作用）的方式。

示例 *2*。 考虑赋值 `g = (int) a;` 在第 3 行。

$$\begin{aligned}\mathsf{eq}\ \langle 4\rangle = \mathbf{fun}\ \mathsf{get}\ \mathsf{set}\ \mathsf{demand}\ \rightarrow\ & \mathbf{let}\ d = \mathsf{get}\ \langle 3\rangle\ \mathbf{in}\\ & \mathsf{set}\ g\ d;\\ & d\end{aligned} \tag{2}$$

这里，首先通过 get 获取 a 的值 d，它恰好也是要传播到未知量 $\langle 4\rangle$ 的值。此外，还通过调用 set $g\ d$ 向全局变量 g 提供了一个贡献值。□

第三个参数需求由 **TLTD** 引入。它可以作用于未知量，这些未知量的值必须计算出来，例如为了确定其副作用，但不会直接使用。以下示例说明了一种可能的用法。

示例 *3*。考虑在第9行创建新线程后，程序点 10 的右侧。创建的线程将并发执行函数foo ，输入为 42。因此，对应的方程会将区间 $[42{,}42]$ 作为 42 的抽象，贡献到foo的起点。求值foo 的结果并非确定下一个程序点 10 的值所必需。尽管如此，仍必须分析foo 的函数体，因为它可能通过set影响全局未知量。因此，需求 会针对 $\langle 5\rangle$这个foo的端点被调用。据此，我们定义

$$\begin{aligned}\mathsf{eq}\ \langle 10\rangle = \mathbf{fun}\ \mathsf{get}\ \mathsf{set}\ \mathsf{demand}\ \rightarrow\ & \mathbf{let}\ d = \mathsf{get}\ \langle 9\rangle\ \mathbf{in}\\ & \mathsf{set}\ \langle 3\rangle\ [42, 42];\\ & \mathsf{demand}\ \langle 5\rangle;\\ & d\end{aligned} \tag{3}$$

□

总而言之，三个参数函数 get、set 和 demand 构成了通用的接口，使 **TLTD** 能够以下列方式与右侧的求值交互：

– 函数式参数 get 接收局部求解器函数 query。调用该函数并传入某个未知量 y 时，最终会返回数据中存储的 y 的值。不过，在此之前，如果尚未开始对 y 进行迭代，就会调用 iterate 计算 y 的最新值。为检查这一点，数据中 y 的记录会维护一个标志 which ； 函数 iterate 在进入 $while$ 循环对 y 执行不动点迭代之前，会将其设为 true。对 y 的迭代结束后，会将 x 插入数据中 y 的 set 属性 influences 集合，以记录 x 对 y 的依赖关系，并返回 y 的最新值。

– 函数式参数 set 接收局部函数 side，它的参数是一个全局变量 g 和对 g 的贡献值。首先，将数据中 g 的属性 stable 设为 true，以便将 g 纳入未知量结果集。如果新的贡献值已被 g 的旧值包含，求解器就继续执行。否则，将数据中 g 的值更新为新旧值的并，并调用 destabilize，通知所有直接或间接依赖于 g 的其他未知量，因为它们当前的值可能已经过时。– 最后，求解器函数 promote 会提供给函数式参数需求。调用 promote 并传入未知量 y 时，它会将 y 添加到顶层工作集，并在数据中记录 y 已提升至顶层。请注意，这里不会记录对 y 的影响关系。

对哈希映射 数据 和一个未知量 x调用时，辅助函数 查找——为简洁起见，此处省略——会在 x 中查找 数据的记录；如果记录不存在，则创建一条记录，并返回该记录的引用。该记录由图 3中访问的字段组成。这与 **TD** 中介绍的做法不同， [30], 后者针对每个字段分别使用哈希映射来组织数据。

具体而言，函数 iterate 对 x 执行的迭代过程如下：首先，将 data 中 x 对应的 called 标志设为 true。重复对 x 进行迭代，直到 data 中为 x 存储的 x 的值不再变化，即达到稳定状态。为标记这一状态， x 对应的数据记录中包含一个 stable 条目；在 while 循环开始前，该条目会被乐观地设为 true。每当对 x 右侧的求值产生的值未被旧值包含时，就将记录数据 x 中的 value 条目更新为 x 的旧值与新值的并。此外，对于所有直接或间接受 x影响的其他未知量，通过调用辅助函数 destabilize，将其 stable 条目设为 false。图 4 所示的函数 destabilize 还会检查失稳的未知量 y 是否为顶层。如果是，则将其重新加入顶层工作集 work，以便重新迭代；这样即使不再查询 y ，也能确保考虑到所有影响关系。最后，假设未知量 x 未失稳——无论是在最近一次对其右侧求值期间，还是在最近一次因修改 x的 value 条目而调用 destabilize 时都未失稳。此时，迭代即告终止。如果 x 已标记为顶层，则也会从 work 中移除 x 。最后，将 data 中名为 called 的条目重置为 false。

例 *4.* 在我们的例子中，全局变量 g 通过 main 中程序点 $\langle 9\rangle$ 的右侧获得贡献值 $[0,0]$ 。计算 $\langle 11\rangle$ 的右侧时会读取该值，并在 $\langle 12\rangle$ 的右侧将其递增，从而为 main 的端点 $\langle 13\rangle$ 提供第一个值。然而，在程序点 $\langle 10\rangle$ 的右侧，会向函数 foo 的起点 $\langle 3\rangle$ 贡献值 $[42,42]$ ；与此同时，在同一个右侧中，foo 的端点 $\langle 5\rangle$ 被转换为一个顶层未知量并插入工作集。随后对 $\langle 5\rangle$ 的迭代又向 g 贡献了 $[42,42]$ 。将这一新贡献值与 g 的旧值合并后，求解器得到 g 的区间 $[0,42]$ 。随后调用使失稳函数，会将 main 的端点 $\langle 13\rangle$ 重新放回工作集 work。在第二轮中，

```
34 并使失稳 x =35 让 dx = 查找数据 x，位
于 36 让 influences = dx.influences，位于
37 dx.influences <- 集合.empty；38 集合.
iter（函数 y ->39 让 dy = 查找数据 y，位于
40 dy.stable <-false；41 如果
dy.toplevelthen（42 work :=集合.add y !
wor
```

图4：辅助函数使失稳

main的主体中，求解器将重新计算 $\langle 11 \rangle$ 和 $\langle 12 \rangle$ 的值，分别得到 $[0, 42]$和 $[1, 43],$ ，它们也就是main端点的最终值。□

与**TD** 相比，**TLTD** 的改动为并行化奠定了基础：

– 将TD所维护的多个数据结构合并为一个，可减少对大型数据结构进行可能代价高昂的访问次数；这些数据结构之后可能会由多个任务共享。– 引入顶层工作集，提供了一种机制，可用延迟或（如第4节所述）并行执行取代对未知量的递归下降。

与原始**TD** 相比，第一项修改不会改变求解器的语义（ [35, 36]给出了不含副作用的原始**TD** 的形式正确性证明）。第二项修改确保每个延迟未知量 x 一旦加入顶层工作列表，就会在其失稳时重新加入该工作列表。这样， x 便保证会持续迭代，直至稳定化。

Goblint 中实现的 **TLTD** 版本还支持加宽 和 收窄 [8, 9]。这一扩展与其并行化无关，因此此处省略。

4 并行化 TLTD

基于任务池的并行化是一种常见的方法，其中有数量较多的任务被调度到数量通常较少的处理器上。例如，这种方法已用于模块化分析 [40]。当分析由足够大的模块组成时，例如线程或函-数，这种方法非常适用。然而，对于求解第1节所述的方程组，未知量以及相应的任务将不再对应于大型组件，而是对应于

单个程序点。当大型程序中存在数量如此之多的微小组件时，为每个未知量创建一个求解器线程并不现实。我们的想法是仅使用**TLTD** 的顶层工作集作为并行化的任务池。任务由根未知量动态确定，这些根未知量会触发对方程系统更大部分的递归探索，随后由任务池将任务映射到固定的一组工作线程上。这样无需预先了解方程系统即可实现并行化。然而，求解器任务的划分以非确定性方式进行，在多次求解器运行中，右端项的求值顺序可能不同。在存在加宽和收窄的情况下，这可能导致精度不同的解。在我们的运行示例中，顶层工作集以main的端点初始化，随后foo的端点也会进入顶层工作集。对这两个端点分别在不同任务中调用iterate，将触发一个任务探索main函数体中的未知量，并进一步探索foo函数体中的未知量。我们提出了两种不同的方法，用于组织多个任务之间的交互。

– 即时方法将所有求解器数据保存在一个共享的线程安全数据结构中，从而保证不同工作线程对单个未知量的访问具有事务性。–独立方法则维护求解器数据的多个本地副本，并引入发布-订阅机制，以便在任务之间共享信息。

即时方法确保在任何时刻，对于任意给定的未知量，最多只有一个工作线程正在进行迭代。这样便可避免无效计算，但代价是每次访问数据都需要同步。独立方法并不试图排除重复工作，而是将同步需求限制在对发布-订阅机制的访问上。具体而言，这些方法的工作方式如下。

4.1 即时方法

该即时方法依赖于数据结构 `data`的线程安全实现。为确保对未知量 x 进行操作的工作线程只向 `data`写入一致的结果，即基于 `data`中最新值的结果，我们采用事务性模型（例如，参见 [11]）。因此，对于 x上的任何操作（例如 `iterate`），都会在操作开始时读取数据。

如果数据在结果保存到共享数据之前发生变化，我们就使用新读取的数据重复整个操作。在实现中，这是通过将数据中的记录替换为原子记录来实现的，我们可以对这些原子记录执行比较并交换 操作（CAS）。仅当记录的最后已知值（由第二个参数提供）未发生变化时，CAS 操作才会设置记录的新值。请注意，这种方法能够实现无锁 实现，这得益于**TLTD** 为与某个未知量相关的所有数据保留单一引用这一事实。我们使用一个哈希映射，其桶采用*Michael-Scott* 队列 [19]实现。在我们的应用中，某个未知量的数据原子包装器一旦初始化，

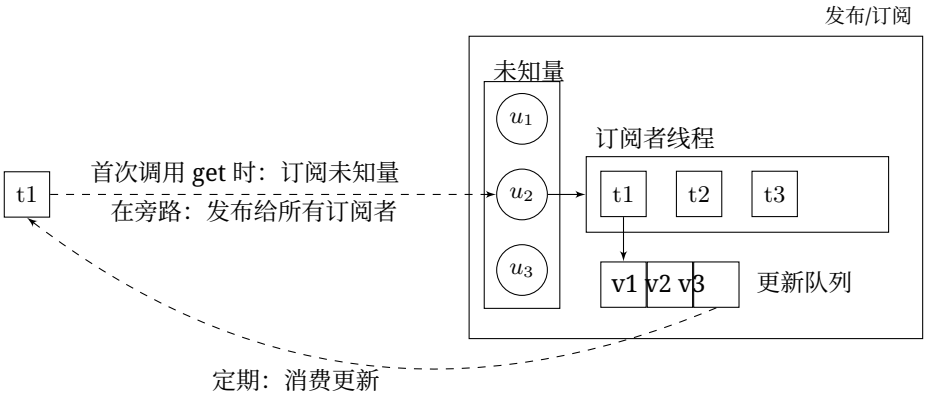


图5：用于全局未知量更新的发布/订阅系统

就不会再发生变化——只有它所引用的值可能会更新。此外，它永远不会从数据中移除，从而简化了哈希映射的实现并提高了其效率。

为了获得良好的性能，因 CAS 失败而导致的重复操作必须极为罕见。请注意，一旦任务 t 已在未知量 x上调用 iterate 函数，在 t 完成迭代并将 called 设为 false 之前，其他任务都不会在 x 上调用 iterate。因此， t 对 x 的 called 和 value 条目拥有独占写入权限，而其他求解器线程只能读取这些条目。其他求解器线程频繁更新、可能导致大量 CAS 失败的唯一字段是 influences。因此，influences 被实现为线程安全集合，并从事务模型中排除。这一设计决策将因 CAS 失败而导致的重复操作比例降低至平均不到 0.01%。不过，为了确保正确性，必须保证每个依赖于 x 的未知量 y，在 x 收到新值时都会失稳。为此，我们总是在保存新值之后 使 x 失稳，并在 y的迭代期间读取 x 的值之前 扩展 x 的 influences。当 x 在扩展 influences 与读取新值之间收到新值时，这一策略可能导致 y 过早失稳，并进行不必要的重新计算，最终得到相同的值。不过，这种情况很少发生，对性能的影响也微乎其微。

当所有已遇到并标记为顶层 的未知量都完成迭代后，求解器便会终止。

4.2 独立方法

理想情况下，对根未知量的良好选择会使递归下降进入方程系统中大体互不相同的部分。值得注意的是，当根未知量对应于程序线程的端点时，情况正是如此。因此，我们的第二种方法假定，即便会遇到，也只有少数局部未知量会被

多个任务遇到，且重新计算它们的成本低于同步开销。因此，它让每个任务操作于一份独立副本 哈希映射的数据。同步仅对全局变量的值是必要的，因为它们没有右端项，但可能接收来自多个局部变量的贡献值。我们使用发布/订阅系统 [22] 用以在各任务之间共享这些贡献值，如图5所示。在首次读取全局变量 g 时，工作线程订阅针对 g的贡献值，而对 g 的每一项贡献值则被发布到其订阅者的线程安全更新队列。这可以通过扩展`query` 与`side`函数轻松实现。此外，所有针对全局变量的贡献值都会被保留，以便新的订阅者也能获得最新的值。每个任务会定期（例如，在对每个右侧求值之后）消费其已订阅的全局变量的全部新更新。其做法是，在其本地副本的数据中更新这些值，并在必要时使受影响的未知量失稳。此外，若任务已订阅的某个全局未知量在任务终止后收到更新，从而可能使其先前结果失效，则这些任务会被重新激活。这保证与全局变量相关的所有更新都会被每个求解器任务纳入考虑。随后，当所有任务均已终止且所有更新队列都已完全处理时，求解器即终止。除对全局变量的处理之外，用于确保正确性的终止准则与**TLTD**中相同。

5 实验结果

我们实现了两种 **TLTD** 并行化理念，并在 OCaml 5.3中使用 Domains 实现真正的并发 [34]。改进后的顺序 求解器 **TLTD** 可通过运行第 2 节中的版本（扩展了加宽和收窄）即时 TD **并配置单个工作线程来恢复。这些求解器已集成到用于多线程 C语言的开源静态分析器**Goblint 中。

并行化的缺点在于，由于结果的运行时间 和精度 可能取决于右端项 的求值顺序，不同的并行运行可能会产生不同的行为，不过可以保证所有返回的结果都是可靠的。

因此，我们的目标是回答以下研究问题。
研究问题1： 第 2 节中的改进对顺序 性能有何影响？**TLTD**？**研究问题2：** 即时和独立方法相比如何？随着工作线程数量增加，它们的扩展性如何？**研究问题3：** 粒度的选择，即需求的放置方式，如何影响并行化效率？**研究问题4：** 两种方法的精度如何变化？不同次运行之间又有何差异？

为回答这些问题，我们配置 Goblint，以进行上下文敏感的过程间分析，并进行一定程度的循环展开。分析器会计算指针的指向信息，以及 int 变量的区间集合，同时跟踪符号线程 id [29] 和确定持有的互斥锁集合。

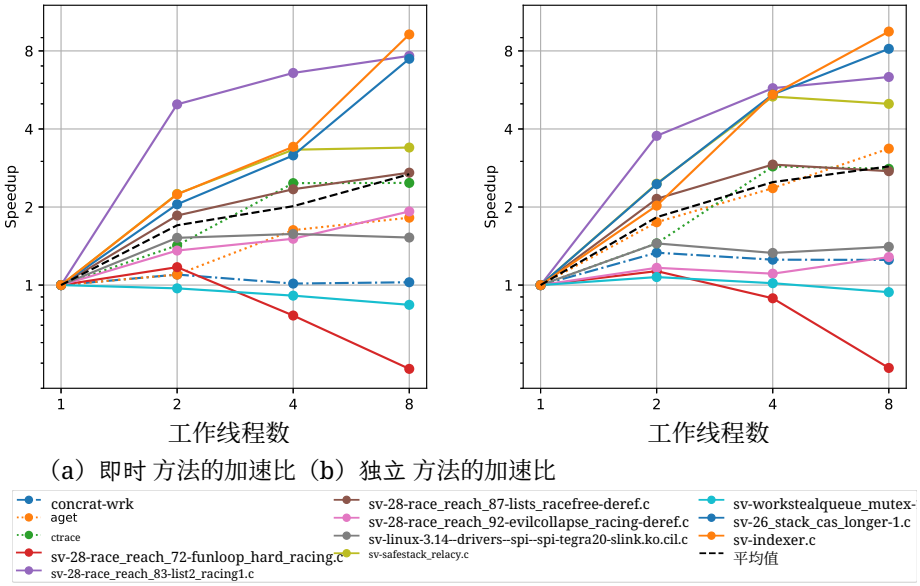
我们从 SV- Comp 的 *Concurrency* 基准套件 [5] 和 Concrat [13]基准测试中收集了基准程序。我们从这些套件中筛选出非平凡程序，剔除了未经修改的 **TD** 在五秒内就结束的输入程序。我们还将四个开源 Unix 实用工具 `aget`、`ctrace`、`knot`和 `pfscan` 纳入基准测试，以衡量其在真实应用中的性能。出于实际考虑，我们将实验超时时间设为 20 分钟。因此，我们剔除了 24 个未经修改的**TD** 运行超时的程序，以及另一个运行独立 求解器超时的程序。在两个案例中，来自独立 求解器的求解器线程的并解虽然可靠，却并非不动点。由于这种行为会导致Goblint 框架异常终止，我们也将这些程序排除在外。

我们的最终基准测试包含 47 个程序（30 个 SV-Comp、13 个 Concrat、4 个 Unix 实用工具），逻辑代码行数从 23 行到 12.5k 行不等。实验在 Ubuntu 24.04.2 LTS 上进行，使用两颗 Intel(R) Xeon(R) Platinum 8260 CPU，每颗有 24 个核心；单个核心的计算能力相当于普通消费级机器中的一个核心。

RQ1. 我们发现，将 Goblint 提供的原始不动点算法中的各种数据结构合并到一个哈希映射中，已能带来可测量的加速，平均提升 1.18× ，这是因为遍历和维护数据结构所花费的时间减少了。我们观察到，不动点迭代的运行时间取决于执行顺序。这并不令人意外，因为如果一个未知量 x 直接或间接依赖于另一个未知量 y，先计算 y 的值会更快，否则就必须重新计算 x 。从经验上看，在每条调用边上查询被调用函数的值之前，将其端点提升到顶层工作集，带来了 0.41× 至 9.49× 的综合加速，平均为 1.85×。仅在 pthread_ 创建时才产生未知量需求的情况下，加速效果更明显，范围为 0.6× 至 10.42×，平均为 2.25×。我们得出结论，可以通过在顶层迭代某些未知量来缩短 **TD** 的运行时间，而粒度会对整体性能产生显著影响。

研究问题 2。 我们评估了两种并行化方法，分别使用 1、2、4 和 8 个工作线程，并通过 `pthread_`创建需求。对于每个基准测试程序，我们计算了各方法相对于单工作线程运行时间的加速比序列。图6 展示了这些加速比。为便于观察结果，我们从完整的基准测试套件中按单工作线程运行时间排序，每隔四个程序选取一个代表性子集，仅展示这些程序的结果。包含所有运行时间的表格见附录。

在所有基准测试上，即时方法相对于使用单个工作者运行时，在2、4和8个工作线程下显示出平均值（最大值）加速比分别为1.7× 、2.01× 和2.68× （4.98×、6.58×、9.26×）；独立方法则分别为1.83×、2.5× 和2.87× （4.25×、7.71×、9.5×）。因此，我们持续发现两种方法都具有显著的加速比，且这些加速比会增加



（a）即时 方法的加速比（b）独立 方法的加速比

图 6：pthread_create 中需求下的相对加速比

随着工作线程数的增加。对于我们的大多数基准程序，加速比往往在工作线程超过四个后逐渐减弱。我们在某些情况下观察到超线性加速比，这同样可以用右端项执行顺序的变化来解释。为比较这两种方法，图7 针对所考虑的每一种工作线程数，显示了即时相对于独立求解器在相同设置下的运行时间之比。独立方法与中间方法的运行时间相近，几乎从不以大于2的因子相差。这些差异可朝任一方向出现；两种方法均不占优。

RQ3. 为了评估需求点选择的影响，我们还运行了即时方法：除了在分析程序通过 `pthread_create` 启动新线程时放置需求外，还会为每个被调用函数端点所对应的未知量放置需求。图8 以与上文相同的方式展示了这些结果。在**RQ1**中，我们注意到，仅在`pthread` 创建时谨慎使用需求节点，可以带来更好的单线程性能。相比之下，我们发现，根节点数量的增加能够带来更好的工作线程数扩展性。这表明，并行性的潜力取决于需求点的选择。在所输入的程序中，如果上述策略引入的并行性潜力不足，随着求解器线程数量增加而产生的并行化开销也会增大，从而导致加速比下降。

研究问题 4。正如预期，不同求解器配置计算出的结果并不总是完全一致。不过，我们通常发现，对于

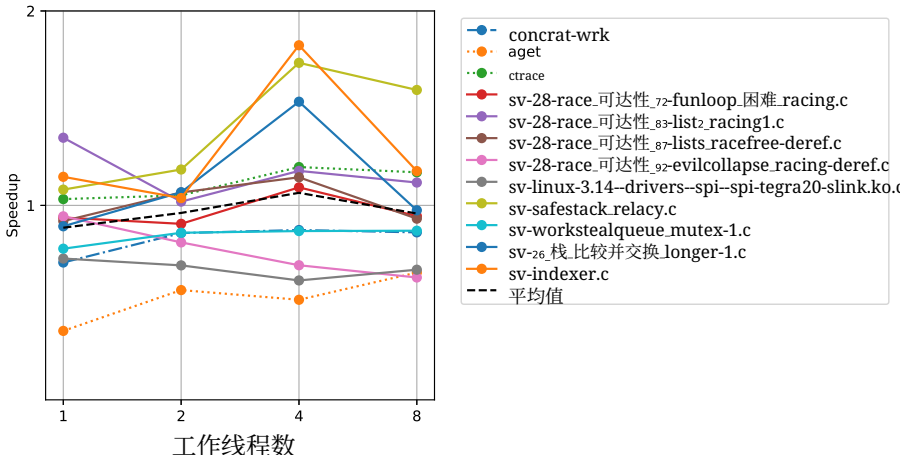


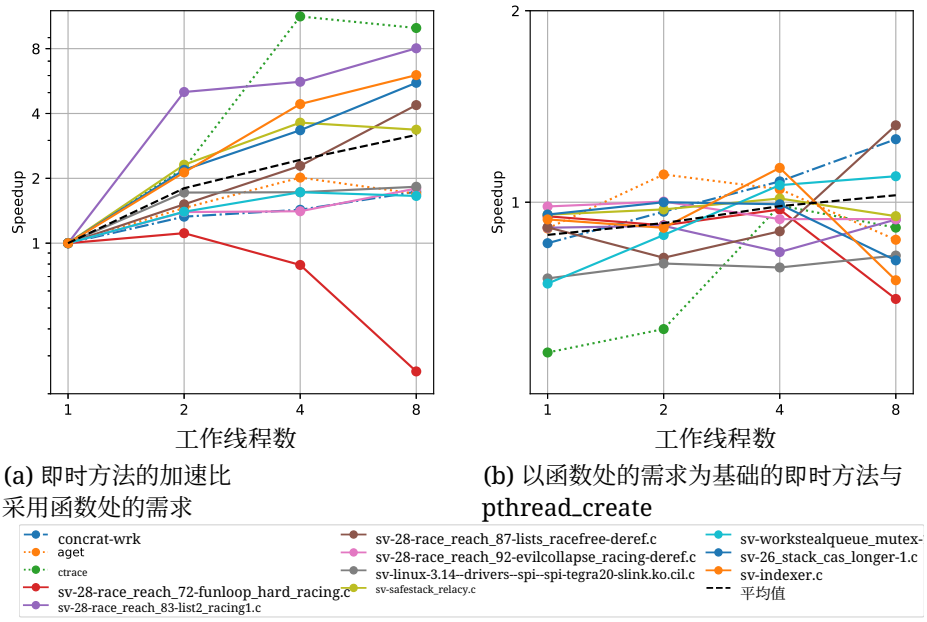
图7：比较独立 与即时

绝大多数未知量（见表1）。平均而言，与未修改的**TD**求得的解相比，获得更精确值的未知量略多于获得精度较低值的未知量。几乎没有未知量的值无法比较。

方法	需求	相等	更精确	较不精确
即时	pthread	0.92	0.04	0.03
即时	pthread，函数	0.92	0.04	0.03
独立	pthread	0.89	0.06	0.05

表1：未 修改前各方法的精确率，与未 修改的 **TD**。

有效性威胁。 一个根本问题是，我们收集的基准测试在多大程度上能够代表相关应用。我们纳入了此前已提出的套件中的非平凡程序，即 SV-Comp 中的非平凡并发基准测试和 Concrat 套件中的基准测试；这些示例已被社区认定为合理。我们还加入了更多多线程开源 Unix 实用工具，尝试用更多常用程序补充现有套件。事实上，我们对程序的研究发现与其来源相当独立。在报告并行求解器运行结果时，我们必须注意，由于操作系统的调度策略或 OCaml 运行时的内存管理，可能会出现非确定性行为。我们报告了单次运行的数量，以揭示性能和精度方面的不规则性。不过，我们观察到，平均



(a) 即时方法的加速比采用函数处的需求

(b) 以函数处的需求为基础的即时方法与 pthread_create

图 8：在函数处设置需求

三次运行的结果中，无论是运行时间，还是在精度相同的情况下未知量所占的比例，差异都不到 2%。

另一个问题是我们对分析配置的选择。为了获得可比较的结果，我们没有报告使用不同抽象域或其他区分多个调用上下文的方法所得到的结果。不过，我们的实验表明，这些方法的结果基本相同。最后，很难直接将实验结果与最先进的并行化方法进行比较，因为底层分析器在需求和局限性方面往往存在根本差异。此外，我们认为最相似的方法是为不同编程语言的分析器实现的，因此无法在同一个基准测试套件上对它们进行评估。

6 相关工作

用于加速程序分析任务的并行化广泛应用于模型检查以及动态或静态分析。我们专注于静态分析，具体而言，关注并行化分析器本身的技术，而非例如并行组合方法（即并行运行多个分析器或其不同配置） [4, 17, 23], , 或将分析规约编译为Datalog程序，再使用并发求值引擎（例如 [1, 14, 26, 27]）。

计算过程自底向上摘要的分析易于并行化，因为所有叶节点都可以并行分析。Shi 和 Zhang [33]报告称，

许多分析器 [3, 7, 18, 32, 43]都采用了这一思路。基于这一观察，Shi 和 Zhang [33] 提出将单个过程的分析拆分为多个任务，以提高并行性。

Monniaux [21] 通过并行计算两个分支的抽象状态，对运行时间较长的 *if-then-else* 语句并行化 Astrée 分析器。具体而言，他关注事件驱动程序中的 *dispatch* 语句：程序会根据待处理的事件，执行不同且可能运行时间较长的代码路径。Kim 等人 [15] 进一步放宽 Bourdoncle 弱拓扑序 [6]，将其改为弱偏序（wpo），从而使程序中彼此无序的部分能够并行执行，进而实现 Ikos 抽象解释器的并行化。他们还证明了其算法具有确定性：任何与 wpo 一致的调度选择，都会得到与 Bourdoncle 原始算法相同的结果。这些方法都要求事先显式提供完整的方程系统——而在上下文敏感分析，尤其是按需分析中，情况并非如此，因为隐式提供的方程系统只会被部分探索。因此，需要对方程系统进行预处理的方法并不适用。

与我们最密切相关的两条研究路线分别来自 Albarghouthi 等人 [2] 和 Van Es 等人 [40]（后者后来发表于扩展版本 [38]）。这两种方法都要求预先定义一组模块（函数，后者还包括线程）。然而，在我们的设定中，这类预定义模块并非随时可用，因为我们的目标是对通用求解器进行并行化。Van Es 等人 [40] 将工作列表视为待分析组件的任务池。对每个组件的分析都会产生需要分析的新组件（可能是因为它们是新发现的，也可能是因为它们依赖当前组件的结果而需要重新分析）。因此，只有在某个组件的分析完成后，才会将更新传递给依赖项；而在我们的方法中，全局变量的更新会在任务之间持续传递。另一方面，Albarghouthi 等人 [2],采用了更结构化的并发方法，在映射 阶段和归约 阶段之间交替进行。在映射 阶段，工作列表中的查询会持续处理，直到产生结果或因依赖关系而阻塞。在归约 阶段，所有查询的状态都会更新，并将解除阻塞的查询重新加入工作列表。在我们的方法中，当任务查询一个仍由另一任务计算中的顶层未知量时，任务不会因此阻塞。相反，它们会获得关于该未知量当前的最佳知识，并乐观地继续工作；但如果该未知量之后发生更新，它们将被重新调度。

Rodriguez 和 Lhoták [25] 并行化 IFDS 风格分析 [24]，即为每个节点定义一个参与者，将数据流事实的传播实现为参与者之间的消息传递，并把消息传播委托给运行时系统。Helm 等人 [12] 同样实现了 IFDS 风格分析的并行化，采用一种半隐式编程模型，在单元之间传播值，开发者显式给出单元间依赖关系，并将某些单元标记为需要非并发更新。他们要求单调右端项以及

满足升链条件的有界格，并且不支持加宽和收窄。

Sun 等人 [39] 提出基于 Apache Giraph，在大型计算集群上对过程间控制流图进行数据流分析。为此，他们设计了一种 分布式 不动点算法，通过仅传播新的数据流事实来减少计算节点之间的通信量，这类似于 半朴素求值。这种方法同样 不 适用于按需分析，也不支持加宽和收窄。

Sherman 和 Dwyer [31] 提出通过拆分控制流图，并并行考虑图中的一些路径，以细粒度方式并行化分析。同样，这种方法不适用于未知量之间存在动态依赖关系的按需分析。

Dewey 等人 [10] 采用了不同的方法，将 JavaScript 程序分析视为一种易于并行化的问题：并行分析所有路径，同时由一个独立进程控制要合并哪些抽象程序状态。其动机在于，对于 JavaScript 这类高度动态的语言，控制流图并不容易获得。这种正向探索方法不容易扩展到我们在此研究的按需分析。

7 结论与未来工作

我们提出了两种并行化通用不动点引擎**TLTD** 的方法，并在静态分析器 Goblint 提供的线程模块化分析中，展示了它们在多个档位（即不同粒度）下的有效性。为并行化而对 Goblint 原有**TD** 所做的修改，已经带来了最高达 10 倍的加速比，平均加速比为 1.85×。在进行并行化后，我们观察到进一步的加速比最高可达 11.29× （4 个工作线程，即时 方法，函数处的需求），以及 9.5×（8 个工作线程，独立 方法，线程创建处的需求）；平均加速比分别为 3.18× （即时方法，8 个工作线程）和 2.87× （独立方法，8 个工作线程）。两种并行化方法各有优势，难分高下。不过，对于较小的粒度（例如过程级），即时 方法似乎更可取。

本文仅展示了采用两种粒度进行实验的结果。未来的工作可进一步探索需求放置的其他可能方案，例如根据待分析程序的特征进行调整。

致谢。 本工作部分得到了欧盟和爱沙尼亚研究委员会通过项目 PRG2764 和 TEM-TA119 的资助。本工作也得益于 Helmut Seidl 和 Michael Schwarz 参与了第 25421 期 Dagstuhl 研讨会 现代软件工程中的健全静态程序分析。

数据可用性声明 该工件包含我们实现的源代码、基准测试、基准测试结果和评估脚本，已归档并可通过 DOI 获取10.5281/zenodo.18186778[16]。

参考文献

[1]Abeysinghe, S., Xhebraj, A., Rompf, T.：Flan：一种用于程序分析的表达力强且高效的 Datalog 编译器。ACM编程语言汇刊 8(POPL)（2024），https://doi.org/10.1145/3632928[2] Albarghouthi, A., Kumar, R., Nori, A.V., Rajamani, S.K.：并行化自顶向下的过程间分析。SIGPLAN Not. 47(6), 217–228 (2012), ISSN 0362-1340, https://doi.org/10.1145/2345156.2254091[3] Babic, D., Hu, A.J.：Calysto：可扩展且精确的扩展静态检查。载于：第 30 届国际软件工程会议论文集，第 211–220 页，ICSE '08，计算机协会，纽约，纽约州，美国（2008），ISBN 9781605580791，https://doi.org/10.1145/1368088. 1368118[4] Beyer, D., Kanav, S., Kleinert, T., Richter, C.：使用现成组件构建验证器组合。Form. Methods Syst. Des. 66(1), 99–130 (2025), ISSN 0925-9856, https://doi.org/10.1007/ s10703-024-00449-y[5]Beyer, D., Strejček, J.：软件验证与见证验证的改进：SV-COMP 2025。载于：TACAS 论文集（3），第 151–186 页，LNCS 15698，施普林格（2025），https://doi.org/10.1007/ 978-3-031-90660-29_[6]Bourdoncle, F.：使用扩宽算子的高效混沌迭代策略。载于：Bjørner, D., Broy, M., Pottosin,I.V.（编）《编程及其应用中的形式化方法》，第 128–141 页，Springer Berlin Heidelberg，柏林，海德堡（1993），ISBN 978-3-540-48056-3[7] Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.：通过双重反演进行组合式形状分析。《ACM 杂志》58(6)（2011），ISSN 0004-5411，https://doi.org/10.1145/2049697.2049700[8] Cousot, P., Cousot, R.：抽象解释：用于通过构造或近似不动点进行程序静态分析的统一格模型。载于：Graham, R.M., Harrison, M.A., Sethi, R.（编）《第四届 ACM 编程语言原理研讨会会议记录》，美国加利福尼亚州洛杉矶，1977 年 1 月，第 238–252 页，ACM（1977），https://doi.org/10.1145/512950.512973[9] Cousot, P., Cousot, R.：抽象解释框架。《J. Log. Comput.》2(4), 511–547 (1992), https://doi.org/10.1093/LOGCOM/2.4.511[10] Dewey, K., Kashyap, V., Hardekopf, B.：面向 JavaScript 的并行抽象解释器。载于：第 13 届 IEEE/ACM 代码生成与优化国际研讨会论文集，第 34–45 页，CGO '15，IEEE 计算机学会，美国（2015），ISBN 9781479981618[11] Grahn, H.：事务内存。《J. Parallel Distributed Comput.》70(10), 993–1008 (2010), https://doi.org/10.1016/J.JPDC.2010.06.006[12] Helm, D., Kübler, F., Kölzer, J.T., Haller, P., Eichberg, M., Salvaneschi, G., Mezini, M.：一种用于半隐式并行化的编程模型

静态分析。载于：第29届 ACM SIGSOFT软件测试与分析国际研讨会论文集，第428–439页，ISSTA 2020，计算机协会，纽约，纽约州，美国（2020），ISBN 9781450380089，https://doi.org/10.1145/3395363.3397367[13] Hong, J., Ryu, S.：Concrat：面向并发程序的自动 C到Rust锁API翻译器。载于：2023 IEEE/ACM 第45届国际软件工程大会（ICSE），第716–728页（2023），ISSN 1558-1225，https://doi.org/10.1109/ICSE48619.2023.00069[14] Jordan, H., Subotić, P., Zhao, D., Scholz, B.：针对 Datalog 专门化并行数据结构。《Concurrency and Computation: Practice and Experience》**34**(2)，e5643（2022），网址 https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5643，e5643 cpe.5643 [15] Kim, S.K., Venet, A.J., Thakur, A.V.：确定性并行不动点计算。ACM编程语言汇刊**4**(POPL)（2019），https://doi.org/10.1145/3371082[16]Kocal, A.R., Schwarz, M., Saan, S., Seidl, H.：同一引擎，多种档位：不同粒度的并行不动点迭代（2025），https://doi.org/10.5281/zenodo.18186778，网址 https://doi.org/10.5281/zenodo.18186778[17]Lemberger, T., Wachowitz, H.：Nacpa：通过并行组合与分析进行原生检查（竞赛贡献）。载于：系统构建与分析的工具和算法：第31届国际会议 TACAS 2025，作为软件理论与实践国际联合会议 ETAPS 2025 的一部分举行，汉密尔顿，安大略省，加拿大，2025年5月3–8日，论文集，第三部分，第236–241页，Springer-Verlag，柏林，海德堡（2025），ISBN 978-3-031-90659-6，https://doi.org/10.1007/978-3-031-90660-2_18[18] McPeak, S., Gros, C.H., Ramanathan, M.K.：可扩展的增量式软件错误检测。载于：2013年第9届软件工程基础联合会议论文集，第554–564页，ESEC/FSE 2013，计算机协会，纽约，纽约州，美国（2013），ISBN 9781450322379，https://doi.org/10.1145/2491411.2501854[19] Michael, M.M., Scott, M.L.：简单、快速且实用的非阻塞与阻塞并发队列算法。载于：Burns, J.E., Moses, Y.（编），第十五届 ACM 分布式计算原理年度研讨会论文集，费城，宾夕法尼亚州，美国，1996年5月23–26日，第267–275页，ACM（1996），https://doi.org/10.1145/248052.248106[20] Monniaux, D.：Astrée静态分析器的并行实现。载于：Yi, K.（编），程序设计语言与系统，第86–96页，Springer Berlin Heidelberg，柏林，海德堡（2005），ISBN 978-3-540-32247-4[21] Monniaux, D.：Astrée静态分析器的并行实现。载于：Yi, K.（编），程序设计语言与系统，第86–96页，Springer Berlin Heidelberg，柏林，海德堡（2005），ISBN 978-3-540-32247-4[22] Oki, B., Pfluegl, M., Siegel, A., Skeen, D.：信息总线：一种可扩展分布式系统架构。《SIGOPS Oper. Syst. Rev.》**27**(5)，58–68（1993），ISSN 0163-5980，https://doi.org/10.1145/173668.168624

[23] Peringer, P., Šoková, V., Vojnar, T.：Predatorhp 的改进版本（不仅）适用于区间大小的内存区域和内存重新分配（竞赛贡献）。载于：Biere, A., Parker, D.（编），系统构建与分析的工具和算法，第408–412页，Springer International Publishing，Cham（2020），ISBN 978-3-030-45237-7[24] Reps, T.W., Horwitz, S., Sagiv, S.：通过图可达性实现精确的过程间数据流分析。载于：Cytron, R.K., Lee, P.（编），POPL'95 会议记录：第22届 ACM SIGPLAN-SIGACT 编程语言原理研讨会，旧金山，加利福尼亚州，美国，1995年1月23–25日，第49–61页，ACM Press（1995），https://doi.org/10.1145/199448.199462[25]Rodriguez, J., Lhoták, O.：基于参与者的并行数据流分析。载于：Knoop, J.（编），编译器构造，第179–197页，Springer Berlin Heidelberg，柏林，海德堡（2011），ISBN 978-3-642-19861-8[26] Sahebolamri, A., Gilray, T., Micinski, K.：通过宏实现无缝演绎推理。载于：第31届 ACM SIGPLAN 编译器构造国际会议论文集，第77–88页，CC 2022，计算机协会，纽约，纽约州，美国（2022），ISBN 9781450391832，https://doi.org/10.1145/3497776.3517779[27] Scholz, B., Jordan, H., Subotić, P., Westmann, T.：关于 Datalog 中快速大规模程序分析。载于：第25届编译器构造国际会议论文集，第196–206页，CC '16，计算机协会，纽约，纽约州，美国（2016），ISBN 9781450342414，https://doi.org/10.1145/2892208.2892226[28] Schwarz, M.：线程模块化抽象解释：局部视角。博士论文，慕尼黑工业大学（2025），网址 https://d-nb.info/137113345X/34[29] Schwarz, M., Saan, S., Seidl, H., Erhard, J., Vojdani, V.：结合局部轨迹的聚类式关系线程模块化抽象解释。载于：欧洲程序设计研讨会，第28–58页，施普林格（2023），https://doi.org/10.1007/978-3-031-30044-8_2[30] Seidl, H., Vogler, R.：对自顶向下求解器的三项改进。《Math. Struct. Comput. Sci.》**31**(9)，1090–1134（2021），https://doi.org/10.1017/S0960129521000499[31] Sherman, E., Dwyer, M.B.：基于结构定义的条件数据流静态分析。载于：Beyer, D., Huisman, M.（编），系统构建与分析的工具和算法，第249–265页，Springer International Publishing，Cham（2018），ISBN 978-3-319-89963-3[32] Shi, Q., Xiao, X., Wu, R., Zhou, J., Fan, G., Zhang, C.：Pinpoint：面向百万行代码的快速精确稀疏值流分析。《SIGPLAN Not.》**53**(4)，693–706（2018），ISSN 0362-1340，https://doi.org/10.1145/3296979.3192418[33] Shi, Q., Zhang, C.：流水线化自底向上数据流分析。载于：ACM/IEEE 第42届国际软件工程大会论文集，第835–847页，ICSE '20，计算机协会，纽约，纽约州，美国（2020），ISBN 9781450371216，https://doi.org/10.1145/3377811.3380425

[34] Sivaramakrishnan, KC.、Dolan, S.、White, L.、Jaffer, S.、Kelly, T.、Sahoo, A.、Parimala, S.、Dhiman, A.、Madhavapeddy, A.：为 OCaml 引入并行性。ACM编程语言汇刊 4（ICFP），113:1–113:30（2020），https://doi.org/10.1145/3408995[35] Stade, Y.、Tilscher, S.、Seidl, H.：自顶向下求解器的部分正确性。形式化证明档案（2024）[36]Stade, Y.、Tilscher, S.、Seidl, H.：经过验证的自顶向下求解器：增强对静态分析器的信心。载于：Gurfinkel, A.、Ganesh, V.（编），计算机辅助验证——第 36 届国际会议，CAV 2024，加拿大魁北克省蒙特利尔，2024 年 7 月 24–27 日，论文集，第 I 部分，计算机科学讲义，第 14681 卷，第 303–324 页，施普林格（2024），https://doi.org/10.1007/978-3-031-65627-9_15，URL https://doi.org/10.1007/978-3-031-65627-9_15[37] Stemmler, F.、Schwarz, M.、Erhard, J.、Tilscher, S.、Seidl, H.：清理有毒垃圾：在混合流敏感静态分析中恢复精度。ACM编程语言汇刊 9（PLDI）（2025）[38] Stiévenart, Q.、Van Es, N.、Van der Plas, J.、De Roover, C.：用于静态模块化分析的并行工作列表算法及其探索启发式。系统与软件杂志 181（C）（2021），ISSN 0164-1212，https://doi.org/10.1016/j.jss.2021.111042[39] Sun, Z.、Xu, D.、Zhang, Y.、Qi, Y.、Wang, Y.、Zuo, Z.、Wang, Z.、Li, Y.、Li, X.、Lu, Q.、Peng, W.、Guo, S.：Bigdataflow：一个分布式过程间数据流分析框架。载于：第 31 届 ACM 欧洲软件工程会议与软件工程基础研讨会论文集，第 1431–1443 页，ESEC/FSE 2023，计算机协会，纽约，美国（2023），ISBN 9798400703270，https://doi.org/10.1145/3611643.3616348[40] Van Es, N.、Stiévenart, Q.、Van der Plas, J.、De Roover, C.：一种用于模块化分析的并行工作列表算法。载于：2020 IEEE 第 20 届源代码分析与操作国际工作会议，第 1–12 页（2020），https://doi.org/10.1109/SCAM51674.2020.00006[41] de Vilhena, P.E.、Pottier, F.、Jourdan, J.：间谍游戏：在 Iris 中验证局部通用求解器。ACM编程语言汇刊 4（POPL），33:1–33:28（2020），https://doi.org/10.1145/3371101[42] Vojdani, V.、Apinis, K.、Rõtov, V.、Seidl, H.、Vene, V.、Vogler, R.：设备驱动程序的静态数据竞争检测：Goblint 方法。载于：第 31 届 IEEE/ACM 自动化软件工程国际会议论文集，ASE 2016，第 391–402 页，ACM（2016），https://doi.org/10.1145/2970276.2970337[43] Xie, Y.、Aiken, A.：使用布尔可满足性进行可扩展的错误检测。载于：第 32 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集，第 351–363 页，POPL '05，计算机协会，纽约，美国（2005），ISBN 158113830X，https://doi.org/10.1145/1040305.1040334

A 附录：所有运行时间

下列表格按 **TD** 未作任何修改时的运行时间排序。

表 2：所有基准测试在**TD** 和**TLTD** （无任何需求）下的运行时间。因此，差异纯粹源于优化数据结构，而非求值顺序。

file	**TD**	**TLTD**
sv-28-race _可达性_87-lists _racefree-deref.c	5.02	4.42
sv-28-race _可达性_87-lists _racefree.c	5.09	4.43
sv-28-race _可达性_86 列表 _racing.c	5.11	4.53
sv-28-race _可达性_94-分配_区域 _racing.c	5.19	3.44
sv-safestack _relacy.c	5.59	4.93
sv-26 _栈 _cas_longest-1-race.c	5.77	4.54
sv-26 _栈 _cas_longer-1-race.c	5.80	4.57
sv-26 _栈 _cas_longest-1.c	6.30	5.07
sv-26 _栈 _cas_longer-1.c	6.31	5.04
sv-09-regions _17-arrayloop _nr.c	6.46	5.73
sv-linux-3.14–drivers–usb–misc–adutux.ko.cil.c	6.69	5.01
concrat-snoopy-mini	6.86	6.76
sv-28-race _可达性_83-list2 _racing1.c	8.19	6.75
sv-28-race _可达性_85-list2 _racefree.c	8.35	7.00
sv-workstealqueue _mutex-2.c	9.01	8.15
concrat-snoopy	9.14	8.71
sv-workstealqueue _mutex-1.c	9.31	8.35
concrat-EasyLogger-mini	14.45	11.84
sv-28-race _可达性_84-list2 _racing2.c	16.12	13.80
concrat-uthash	19.99	16.92
sv-linux-3.14–drivers–spi–spi-tegra20-slink.ko.cil.c	29.85	22.67
concrat-libaco	36.54	32.21
concrat-vanitygen	42.70	33.85
sv-linux-3.14–drivers–net–irda–w83977af _ir.ko.cil.c	45.42	27.99
sv-28-race _可达性_92-evilcollapse _racing-deref.c	48.62	44.35
sv-28-race _可达性_91-arrayloop2 _racefree.c	49.00	45.47
sv-28-race _可达性_93-evilcollapse _racefree.c	50.05	45.23
sv 消除 _退避 _stack-race.c	68.74	63.14
sv-indexer.c	77.24	51.19
sv-28-race _可达性_92-evilcollapse _racing.c	84.81	81.33
sv 消除 _退避 _stack.c	89.40	83.18
pfscan	103.25	97.45
ctrace	124.02	105.27
concrat-pingfs	163.10	102.96
concrat-nnn-mini	179.71	149.25

concrat-stud	193.68	175.32
aget	193.97	182.34
sv-28-race _可达性_90-arrayloop2 _racing.c	202.13	192.71
knot	226.51	198.67
concrat-ProcDump-for-Linux-mini	233.36	232.03
concrat-wrk	244.01	179.00
sv-28-race _可达性_70-funloop _racefree.c	414.92	365.53
concrat-EasyLogger	445.13	411.45
sv-28-race _可达性_73-funloop _hard_racefree.c	454.80	419.66
sv-28-race _可达性_72-funloop _hard_racing.c	469.78	417.31
concrat-cava	479.46	458.32
sv-28-race _可达性_71-funloop _racing.c	543.79	480.52

表3：使用在 pthread_create 处进行需求处理的即时求解器时，所有基准测试的运行时间，按工作线程数划分

file	1	2	4	8
sv-28-race _可达性_87-lists _racefree-deref.c	4.45	2.40	1.90	1.64
sv-28-race _可达性_87-lists _racefree.c	4.44	2.29	1.97	1.20
sv-28-race _可达性_86 列表 _racing.c	4.48	2.58	2.20	1.38
sv-28-race _可达性_94-分配_区域 _racing.c	1.42	1.20	1.11	1.02
sv-safestack _relacy.c	1.73	0.77	0.52	0.51
sv-26 _栈 _cas_longest-1-race.c	2.63	1.28	0.77	0.33
sv-26 _栈 _cas_longer-1-race.c	2.61	1.29	0.80	0.36
sv-26 _栈 _cas_longest-1.c	2.86	1.50	0.73	0.42
sv-26 _栈 _cas_longer-1.c	2.91	1.42	0.92	0.39
sv-09-regions _17-arrayloop _nr.c	1.87	2.09	2.47	2.33
sv-linux-3.14–drivers–usb–misc–adutux.ko.cil.c	4.53	2.96	2.47	2.62
concrat-snoopy-mini	2.86	1.98	1.27	1.21
sv-28-race _可达性_83-list2 _racing1.c	13.09	2.63	1.99	1.71
sv-28-race _可达性_85-list2 _racefree.c	10.38	2.69	2.35	1.56
sv-workstealqueue _mutex-2.c	5.88	6.39	6.20	7.57
concrat-snoopy	4.82	3.08	2.50	1.96
sv-workstealqueue _mutex-1.c	6.11	6.29	6.71	7.28
concrat-EasyLogger-mini	6.14	2.08	2.16	2.27
sv-28-race _可达性_84-list2 _racing2.c	15.98	3.94	3.36	2.70
concrat-uthash	18.96	11.98	14.90	16.13
sv-linux-3.14–drivers–spi–spi-tegra20-slink.ko.cil.c	11.70	7.69	7.43	7.67
concrat-libaco	15.79	6.42	6.59	7.19
concrat-vanitygen	36.36	38.39	39.90	39.32
sv-linux-3.14–drivers–net–irda–w83977af _ir.ko.cil.c	19.28	13.22	13.01	13.77
sv-28-race _可达性_92-evilcollapse _racing-deref.c	26.54	19.53	17.59	13.80
sv-28-race _可达性_91-arrayloop2 _racefree.c	27.00	19.16	17.76	17.96
sv-28-race _可达性_93-evilcollapse _racefree.c	27.45	19.39	16.39	15.64

sv 消除 _退避 _stack-race.c	35.36	21.30	25.98	29.96
sv-indexer.c	7.41	3.31	2.17	0.80
sv-28-race _可达性_92-evilcollapse _racing.c	47.03	33.84	32.69	35.50
sv 消除 _退避 _stack.c	53.05	29.26	36.44	43.46
pfscan	25.12	14.57	11.36	8.82
ctrace	36.62	25.76	14.81	14.78
concrat-pingfs	62.66	65.88	62.76	62.45
concrat-nnn-mini	53.94	72.06	100.21	102.74
concrat-stud	139.74	130.84	120.08	123.34
aget	96.06	87.52	58.97	52.80
sv-28-race _可达性_90-arrayloop2 _racing.c	117.97	71.87	61.94	54.56
knot	211.21	136.87	79.01	43.39
concrat-ProcDump-for-Linux-mini	248.43	131.16	106.85	113.44
concrat-wrk	121.71	110.83	120.09	118.68
sv-28-race _可达性_70-funloop _racefree.c	137.00	137.41	205.86	365.11
concrat-EasyLogger	198.02	136.22	184.98	164.50
sv-28-race _可达性_73-funloop _hard_racefree.c	137.94	129.24	202.67	387.21
sv-28-race _可达性_72-funloop _hard_racing.c	149.75	127.71	196.23	315.09
concrat-cava	800.08	494.38	467.60	491.69
sv-28-race _可达性_71-funloop _racing.c	144.95	143.62	221.97	302.83

表4：使用带有需求集函数的即时求解器时，所有基准测试的运行时间（按工作线程数划分）

file	1	2	4	8
sv-28-race _可达性_87-lists _racefree-deref.c	5.12	3.38	2.24	1.17
sv-28-race _可达性_87-lists _racefree.c	5.35	3.01	2.14	1.16
sv-28-race _可达性_86 列表 _racing.c	4.81	3.28	2.32	1.19
sv-28-race _可达性_94-分配_区域 _racing.c	1.49	1.34	1.09	0.94
sv-safestack _relacy.c	1.85	0.80	0.51	0.55
sv-26 _栈 _cas_longest-1-race.c	2.88	1.32	0.73	0.30
sv-26 _栈 _cas_longer-1-race.c	2.90	1.24	0.82	0.32
sv-26 _栈 _cas_longest-1.c	3.10	1.66	0.72	0.39
sv-26 _栈 _cas_longer-1.c	3.11	1.42	0.93	0.56
sv-09-regions _17-arrayloop _nr.c	3.19	2.27	1.77	1.40
sv-linux-3.14–drivers–usb–misc–adutux.ko.cil.c	5.84	3.38	2.59	3.05
concrat-snoopy-mini	2.74	1.72	1.33	1.00
sv-28-race _可达性_83-list2 _racing1.c	15.10	3.00	2.69	1.88
sv-28-race _可达性_85-list2 _racefree.c	11.25	3.18	2.14	2.06
sv-workstealqueue _mutex-2.c	9.75	7.51	6.88	6.12
concrat-snoopy	4.70	3.01	2.07	1.74
sv-workstealqueue _mutex-1.c	10.62	7.59	6.16	6.41
concrat-EasyLogger-mini	6.43	2.18	2.19	2.30
sv-28-race _可达性_84-list2 _racing2.c	18.74	5.28	4.45	2.75

concrat-uthash	19.13	12.00	14.03	13.58
sv-linux-3.14–drivers–spi–spi-tegra20-slink.ko.cil.c	19.43	11.31	11.27	10.63
concrat-libaco	17.32	6.91	6.49	6.54
concrat-vanitygen	39.33	26.35	30.60	26.22
sv-linux-3.14–drivers–net–irda–w83977af _ir.ko.cil.c	39.58	26.67	24.26	17.23
sv-28-race _可达性_92-evilcollapse _racing-deref.c	27.14	19.46	19.29	15.13
sv-28-race _可达性_91-arrayloop2 _racefree.c	27.87	18.58	18.07	16.08
sv-28-race _可达性_93-evilcollapse _racefree.c	28.37	19.52	18.49	13.31
sv 消除 _退避 _stack-race.c	55.09	26.49	26.53	30.81
sv-indexer.c	8.14	3.82	1.84	1.35
sv-28-race _可达性_92-evilcollapse _racing.c	48.00	34.37	31.69	30.86
sv 消除 _退避 _stack.c	83.26	45.62	37.47	52.14
pfscan	25.43	15.22	11.71	8.66
ctrace	169.98	76.28	15.06	17.04
concrat-pingfs	81.56	67.92	63.11	73.40
concrat-nnn-mini	144.00	274.39	102.30	32.18
concrat-stud	155.47	136.76	132.53	121.25
aget	111.33	76.44	55.22	65.66
sv-28-race _可达性_90-arrayloop2 _racing.c	120.50	69.41	51.86	42.86
knot	226.80	149.80	90.93	59.48
concrat-ProcDump-for-Linux-mini	323.52	156.11	151.45	152.56
concrat-wrk	154.83	116.62	108.31	89.30
sv-28-race _可达性_70-funloop _racefree.c	160.45	147.62	226.91	758.70
concrat-EasyLogger	238.74	148.14	154.16	152.31
sv-28-race _可达性_73-funloop _hard_racefree.c	162.47	149.13	226.44	583.89
sv-28-race _可达性_72-funloop _hard_racing.c	161.62	145.22	203.96	636.51
concrat-cava	1178.85	522.54	492.47	478.89
sv-28-race _可达性_71-funloop _racing.c	164.79	145.31	232.06	499.13

表5：使用独立 求解器、在 pthread_create 处设置需求时，各基准测试的运行时间（按工作线程数划分）

file	1	2	4	8
sv-28-race _可达性_87-lists _racefree-deref.c	4.84	2.25	1.66	1.76
sv-28-race _可达性_87-lists _racefree.c	4.81	2.40	1.57	1.91
sv-28-race _可达性_86 列表 _racing.c	4.85	3.12	1.86	1.87
sv-28-race _可达性_94-分配_区域 _racing.c	1.14	0.61	0.70	0.57
sv-safestack _relacy.c	1.60	0.65	0.30	0.32
sv-26 _栈 _cas_longest-1-race.c	2.93	0.69	0.38	0.32
sv-26 _栈 _cas_longer-1-race.c	2.94	0.80	0.40	0.33
sv-26 _栈 _cas_longest-1.c	3.23	1.42	0.61	0.41
sv-26 _栈 _cas_longer-1.c	3.26	1.33	0.60	0.40
sv-09-regions _17-arrayloop _nr.c	1.91	2.12	2.56	2.46
sv-linux-3.14–drivers–usb–misc–adutux.ko.cil.c	4.23	3.52	2.46	2.85

concrat-snoopy-mini	5.25	3.96	2.10	1.61
sv-28-race _可达性_83-list2 _racing1.c	9.71	2.58	1.69	1.53
sv-28-race _可达性_85-list2 _racefree.c	10.74	2.72	2.22	1.57
sv-workstealqueue _mutex-2.c	7.83	7.33	7.81	7.97
concrat-snoopy	7.10	4.26	3.17	2.17
sv-workstealqueue _mutex-1.c	7.86	7.32	7.73	8.37
concrat-EasyLogger-mini	9.56	2.96	3.07	3.17
sv-28-race _可达性_84-list2 _racing2.c	16.02	5.08	3.08	2.63
concrat-uthash	16.79	10.32	10.87	11.47
sv-linux-3.14–drivers–spi–spi-tegra20-slink.ko.cil.c	16.10	11.12	12.10	11.46
concrat-libaco	12.96	5.01	5.23	5.29
concrat-vanitygen	34.97	36.41	37.77	38.83
sv-linux-3.14–drivers–net–irda–w83977af _ir.ko.cil.c	36.10	31.06	34.31	29.77
sv-28-race _可达性_92-evilcollapse _racing-deref.c	28.12	24.12	25.41	21.93
sv-28-race _可达性_91-arrayloop2 _racefree.c	29.43	23.50	22.64	22.11
sv-28-race _可达性_93-evilcollapse _racefree.c	28.81	21.07	22.63	22.69
sv 消除 _退避 _stack-race.c	33.00	17.29	15.30	16.77
sv-indexer.c	6.46	3.19	1.19	0.68
sv-28-race _可达性_92-evilcollapse _racing.c	47.73	36.83	41.05	34.02
sv 消除 _退避 _stack.c	50.17	21.68	22.38	37.59
pfscan	23.41	16.94	10.64	8.07
ctrace	35.47	24.49	12.37	12.63
concrat-pingfs	101.64	105.06	108.10	112.63
concrat-nnn-mini	113.09	142.58	155.05	167.36
concrat-stud	143.26	118.47	126.43	119.04
aget	270.91	155.04	114.58	80.65
sv-28-race _可达性_90-arrayloop2 _racing.c	130.57	87.97	63.62	88.51
knot	351.63	201.23	128.36	71.19
concrat-ProcDump-for-Linux-mini	411.38	185.67	165.17	166.32
concrat-wrk	172.30	129.22	137.50	137.78
sv-28-race _可达性_70-funloop _racefree.c	152.24	121.79	185.16	321.56
concrat-EasyLogger	350.83	210.12	221.65	244.90
sv-28-race _可达性_73-funloop _hard_racefree.c	157.23	142.04	172.59	328.60
sv-28-race _可达性_72-funloop _hard_racing.c	159.72	141.08	179.58	333.34
concrat-cava	850.30	389.75	413.46	404.21
sv-28-race _可达性_71-funloop _racing.c	159.44	145.27	203.60	381.46