Push-Button Verification of File Systems via Crash Refinement

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, Xi Wang
University of Washington

姓名：陶鹏

学号：2017202110045

专业：计算机软件与理论

1. Introduction

文件系统是保存存储设备上的数据所必需的操作系统组件。编写一个没有BUG 的文件系统是非常重要的，因为它们必须正确地实现和维护复杂的磁盘数据结构，即使存在系统崩溃和磁盘操作的重新排序，但是这是很困难的，尤其是碰到断电或者系统崩溃，文件来不及保存，这种情况在目前的很多文件系统都存在，导致数据损失。

这篇文章提出了一个新的工具：Yggdrasil. 作者对其定义是一种用来帮助编写文件系统的工具，使用了 push-button 验证机制。这个工具不需要人工标注和证明文件系统实现代码的正确性，但是如果代码里面存在某个 BUG,这个工具会产生一个相反的示例。这个工具之所以可以实现这种自动化的原因在于一种新的定义：Crash Refinement，一种新的对文件系统的正确性的定义。Crash Refinement 需要当前文件系统所可能存在的状态必须是一种规范里的所有状态的子集。而这种规范也是实现 Yggdrasil 的基础。

Yggdrasil 需要 3 个输入：

1. 预期行为的规范化(a specification of the expected behavior)
2. 一种实现（an implementation）
3. 指示文件系统映像是否处于一致状态的一致性不变量(consistency invariants indicating whether a file system image is in a consistent state)

在编写代码的人员输入这三个参数后，工具就会自动验证输入的实现是否满足输入的规范，如果存在 BUG，Yggdrasil 会产生一个相反的示例来帮助编写人员确认以及修改出错的地方；如果实现满足了规范，工具就会生成可执行代码。由此可知，这个工具取代了编写人员验证其实现的正确性的步骤，正真实现了正确性的自动验证。

如之前提到的，Yggdrasil 之所以可以实现自动验证，在于它是基于 Crash Refinement,这种定义适用于有效的可满足性模块理论推理（SMT） ， Yggdrasil 将文件系统的验证当做一个 SMT 问题，并且采用最新的 SMT solver 解决这个问题，从而实现了正确性的自动验证。

此外 Crash Refinement 允许编写人员在栈的层次上抽象实现文件系统：如果一个文件系统遵循 Crash Refinement 规范，那么它对于更高层次来说是不可分辨的，更高层可以直接使用低层的实现而不需要推导它的正确性。而这种模块性的设计使得 Yggdrasil 验证文件系统的正确性时，只需要在一个层次上验证所有的执行路径，而不需要遍历两个或多个层次之间所有的执行路径，大大减轻了验证时的负担，同时也缩短了时间。相应的，如果一个实现满足了某个规范，那么这个实现可以直接被其他需要满足这种规范的文件系统重用。

总的来说，这篇文章有三个贡献：

1. 文件系统 Crash Refinement 规范
2. 使用了 Crash Refinement 规范实现的工具 Yggdrasil
3. 使用 Yggdrasil 编写的其他文件系统的案例研究（如：Yxv6 file system）

2. Yggdrasil 使用与 YminLFS 实现

2.1 Specification

首先，Yggdrasil 规定一个文件系统由三部分组成：

1. 表达逻辑布局的抽象数据结构
2. 基于数据结构上的一系列操作（用来定义预期的行为）
3. 一个用来定义实现是否满足规范的等价谓词

文章以构建 YminLFS 系统为例，从 3 个方面描述这几个组成。

2.1.1 abstract data structure

YminLFS 的抽象数据结构表示如下：

```
class FSSpec(BaseSpec):
  def __init__(self):
    self._childmap  = Map((InoT, NameT), InoT)
    self._parentmap = Map(InoT, InoT)
    self._mtimemap  = Map(InoT, U64T)
    self._modemap   = Map(InoT, U64T)
    self._sizemap   = Map(InoT, U64T)
```

数据结构由 5 个 abstract maps 组成，是一个 key-value 对。其中 childmap 的 key 由目录索引号和名称组成，value 是子索引号；parentmap 的 key 是索引号，value 是其父索引号；其余的是用来存储节点元数据。其中 InoT 和 U64T 是整数类型，NameT 是字符串类型。

对于这种结构，有个问题就是可能在 childmap 中，节点 D 包括了节点 F，但是在 parentmap 里面，节点 D 并没有包括节点 F，这种数据结构并没有定义一致性，因此，可以添加一个限制，保证这种数据的一致性。限制实现如下：

```
 def invariant(self):
  ino, name = InoT(), NameT()
  return ForAll([ino, name], Implies(
   self._childmap[(ino, name)] > 0,
   self._parentmap[self._childmap[(ino, name)]] == ino))
```

2.1.2 file system operations

对于一个文件系统，常见的操作有：

只读操作：查询（lookup），状态（stat）等，其实现逻辑如图：

```
def lookup(self, parent, name):
  ino = self._childmap[(parent, name)]
  return ino if ino > 0 else -errno.ENOENT

def stat(self, ino):
  return Stat(size=self._sizemap[ino],
              mode=self._modemap[ino],
              mtime=self._mtimemap[ino])
```

修改更新操作，保证数据不会因为崩溃或断电等原因导致部分修改成功，所以这部分必须为原子操作，加入 transaction：

```
def mknod(self, parent, name, mtime, mode):
  # Name must not exist in parent.
  if self._childmap[(parent, name)] > 0:
    return -errno.EEXIST

  # The new ino must be valid & not already exist.
  ino = InoT()
  assertion(ino > 0)
  assertion(Not(self._parentmap[ino] > 0))

  with self.transaction():
    # Update the directory structure.
    self._childmap[(parent, name)] = ino
    self._parentmap[ino] = parent
    # Initialize inode metadata.
    self._mtimemap[ino] = mtime
    self._modemap[ino]  = mode
    self._sizemap[ino]  = 0

  return ino
```

2.1.3 state equivalence predictate

一个正确的实现必须和抽象数据结构表示的文件存储一致

```
def equivalence(self, impl):
  ino, name = InoT(), NameT()
  return ForAll([ino, name], And(
    self.lookup(ino, name) == impl.lookup(ino, name),
    Implies(self.lookup(ino, name) > 0,
      self.stat(self.lookup(ino, name)) ==
      impl.stat(impl.lookup(ino, name)))))
```

2.2 implementation

使用 Yggdrasil 实现一个文件系统，必须要做 3 件事：

1. 选择一个磁盘模型
2. 为每个操作编写实现代码
3. 写入磁盘布局的一致性不变量

2.2.1 磁盘模型（disk model）

Yggdrasil 有若干个可选模型：YminLFS (以及 Yxv6)使用了异步磁盘模型，同样的还有同步磁盘模型。

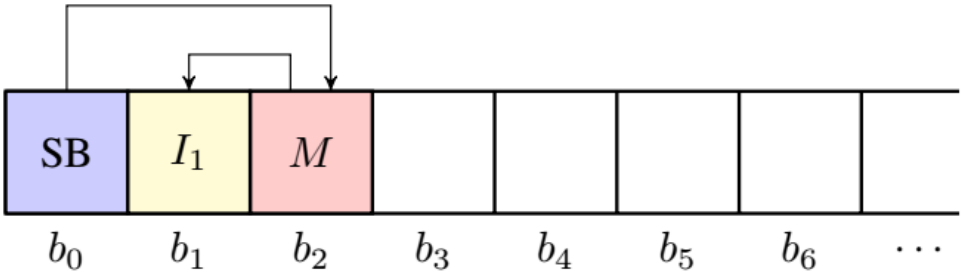
异步磁盘模型指定了一个块设备，这个块设备是无边界的、易失性的并且允许任意的重排。它提供了几个接口：d.write(a,v) 将数据块 v 写入到地址为 a 的部分；d.read(a) 返回地址为 a 的数据块；d.flush() 清空磁盘缓存。并且磁盘的读写是个原子操作。

2.2.2 a log-structured file system

本小节讲述的是 YminLFS 的磁盘布局，YminLFS 是一个日志结构的文件系统，以 copy-on-write 的方式工作，它不会覆盖已存在的块（除了在 0 块处的 superblock，它的内容随着数据更新而更新，当有新的数据写入时，它的指向会改变），YminLfS 没有垃圾回收机制，所以可想而知，这个文件系统在用完块，节点以及目录项时，新数据写入就会失败。

YminFLS 提供一个接口 mkfs，用来初始化磁盘以及读取和修改文件系统的状态（所谓的写操作）。初始化操作涉及到 3 个块，superblock，两个 iNode，我的理解是 superblock 是存储的开始部分（之前提到的 0 块位置），一个 iNode 指向根目录，另一个映射到 M,M 块存着 inode 号到块号的映射。

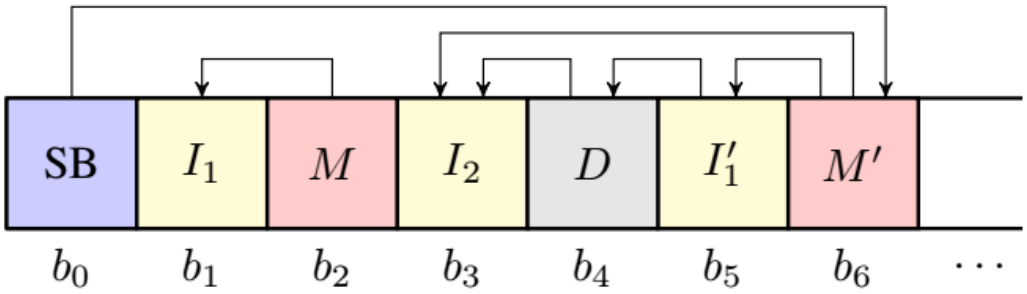
磁盘的初始化状态如下图所示：



(a) The initial disk state of an empty root directory.

此时 I_1 指向为空，M 存着 inode 号到 block 号的映射，所以此时 M 中只有一条映射：1—>b_1,superblock 指向 M，它储存两个计数器：下一个可用的 inode 号以及下一个可用的 block 号。

如果添加一个文件到根路径下面，mkdod(我的理解是 mkfs 下面的一个操作)对上述图的改变如下：



(b) The disk state after adding one file.

步骤描述如下：

1) 为新文件添加一个新的 inode 块，inode 号为 2（I_2）
2) 为根目录添加一个数据块 D，存储文件名到文件 inode 号的映射
3) 此时 I_1 指向需要更新，由于这个文件系统除了 SB 块外，其余块都是不可覆盖的，所以只能在新的块上面添加一个 I_1'用来更新根目录的指向，此时指向 D
4) 由于 inode 号和 block 号都有所改变，需要更新 M 中的映射，同样由于 3)中的原因，只能分配一个新块来执行更新操作，添加 M'，M'中有两条映射：1→b_5，2→b_3
5) SB 是唯一可更新的，所以最后更新 SB,指向 M'，同时更新 SB 中的计数器

另外，mknod 操作保证磁盘刷新是 crash-safe 的。

2.2.3 consistency invariants

文件系统实现的一致不变性类似于其规范的格式不变性。它是一个确定给定的磁盘状态是否对应于有效的文件系统映像的谓词。我对此的理解是设置一些确

保文件系统映像有效的限制条件

Yggdrasil 使用 consistency invariants 主要有两个原因：push-button verification 和运行时 sck 层的检查。为了验证，Yggdrasil 在 mkfs 之后检查不变量是否适合于初始文件系统状态; 此外，它假定一致性不变作为每个操作的前提条件的一部分，并检查不变量是否为后置条件的一部分。一旦实现被验证，Yggdrasil 可以从这些不变式生成一个类似 fsck 的检查器（尽管检查器不能修复损坏的文件系统）。 这样的检查器即使对于无缺陷的文件系统也是有用的，因为系统的其他部分中的硬件故障和错误可能会损坏文件系统。

YminLFS 的 consistency invariant 由三部分组成：SB，inode 映射的块 M,根目录块 D，每个部分都有一些限制，比如 SB，SB 有两个计数器，一个记录下一个有效的 inode 号，另一个记录下一个有效的 block 号，如果这两个计数器不添加一些限制的话可能会导致已使用的 inode 号或者 block 号被记录进去，所以添加了限制，Inode 号 > 当前已使用的 Inode 号，block 号 > 当前已使用的 block 号。其余两个部分限制与此类似。

2.3 verification

验证 YminLFS 的实现是否满足 Yggdrasil 的规范（2.1 节描述），Yggdrasil 使用了 Z3 solver 来证明一个 two-part crash refinement 理论。这个理论第一部分解决 crash-free execution。它要求实现和规范在没有崩溃的情况下表现相似：如果 YminLFS 和 FSSpec 都以相等和一致的状态开始，则它们以相等和一致的状态结束。这个理论第二部分解决 crashing execution。它要求实现不会出现比规范更多的崩溃状态（崩溃后的磁盘状态）：YminLFS 实现的每种可能状态（包括由崩溃和重新排序写入引起的状态）必须等同于 FSSpec 的某种崩溃状态。

如果实现或一致性不变式中存在任何错误，验证器将生成一个反例来帮助程序员理解错误。

```
# Pending writes
lfs.py:167 mknod write(new_imap_blkno, imap)

# Synchronized writes
lfs.py:148 mknod write(new_blkno, new_ino)
lfs.py:154 mknod write(new_parentdata, parentdata)
lfs.py:160 mknod write(new_parentblkno, parentinode)
lfs.py:170 mknod write(SUPERBLOCK, sb)

# Crash point
[..]
lfs.py:171 mknod flush()
```

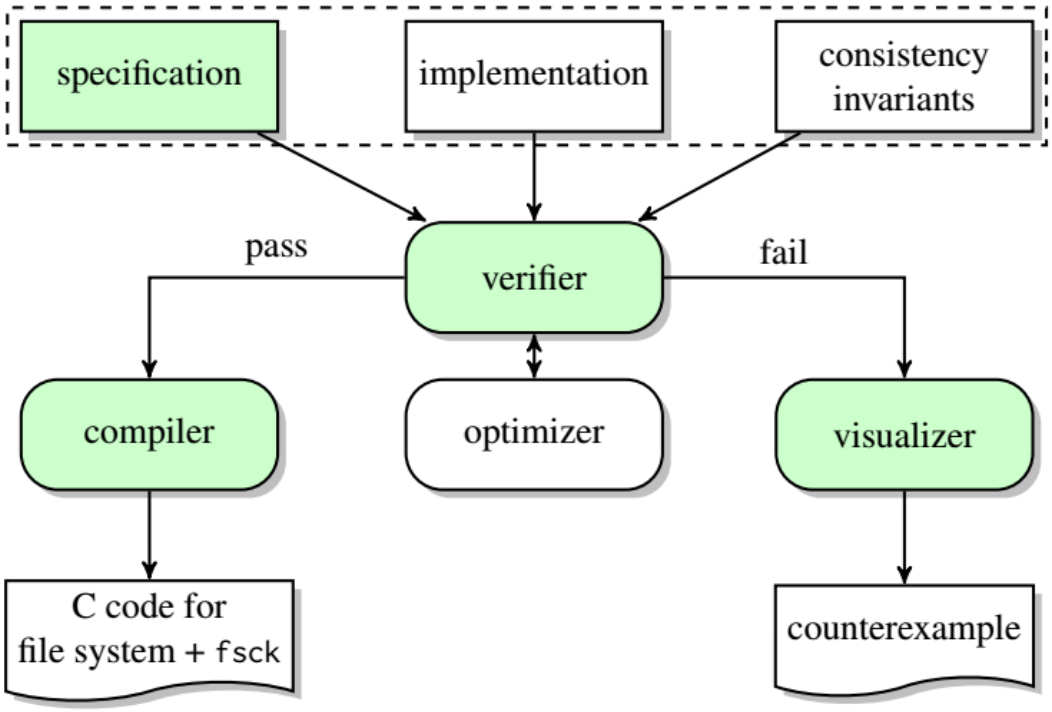
2.4 optimization and compilation

如 2.2.2 所述，YminLFS 的 mknod 实现使用了五次磁盘刷新。 Yggdrasil 提供了一个贪婪的优化器，试图删除每个磁盘刷新并重新验证代码。 在 mknod 代码上运行优化程序会在三分钟内删除五次刷新中的三次，同时仍能保证正确性。

在 Python 中优化和验证的 YminLFS 实现是可行的，但速度很慢。 Yggdrasil 调用 Cython 编译器从 Python 生成 C 代码以获得更好的性能。它还提供了一个小桥梁，将生成的 C 代码连接到 FUSE。结果是一个单线程的用户空间文件系统。

2.5 Yggdrasil 使用总结

文章第二节展示了如何使用 Yggdrasil 来实现一个简单的日志结构的文件系统，可以发现，使用了 Yggdrasil 后代码实现的正确性不再需要手动去验证。大大减轻了编写人员的负担，使得编写人员的重点转移到实现 Yggdrasil 的规范上面。然后，使用的流程图展示如下：



接下来，文章讲述了 Yggdrasil 的具体结构。

3. Yggdrasil architecture

3.1 理论基础

这一节主要是讲述对系统的建模以及工具的理论基础

3.1.1 建模

使用函数 f 对系统操作的行为进行建模，该函数采用三个输入：

- 其当前状态 s;
- 外部输入 x，如要写入的数据;
- crash-schedule b，这是一组布尔值表示发生 crash 事件。

对这些输入应用 f，写成 f（s; x; b），产生系统的下一个状态。

Yggdrasil 中的异步磁盘模型生成一对布尔值（on; sync）作为 crash-schedule。On 通过将其数据存储到易失性高速缓存中来指示写入操作是否成功完成。 sync 指示写入效果是否已从易失性缓存同步到稳定存储。

3.1.2 crash refinement

这节没怎么看懂，所以有些地方可能描述有问题。

Crash-free equivalence：给定两个函数 f_0 和 f_1 以及它们的系统一致性不变量 I_0 和 I_1，如果满足以下条件，则我们说 f_0 和 f_1 是 Crash-free 等价的：

$$\forall s_0, s_1, \boldsymbol{x}.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{true}) \text{ and } s_1' = f_1(s_1, \boldsymbol{x}, \boldsymbol{true}).$$

Crash refinement without recovery：函数 f_1 是 f_0 的 Crash refinement without recovery 如果

（1）f_0 和 f_1 是 Crash-free equivalence 的，并且

（2）以下成立

$$\forall s_0, s_1, \boldsymbol{x}, \boldsymbol{b}_1.\ \exists \boldsymbol{b}_0.\ (s_0 \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1) \Rightarrow (s_0' \sim_{\mathcal{I}_0,\mathcal{I}_1} s_1')$$
$$\text{where } s_0' = f_0(s_0, \boldsymbol{x}, \boldsymbol{b}_0) \text{ and } s_1' = f_1(s_1, \boldsymbol{x}, \boldsymbol{b}_1).$$

Recovery idempotence：A recovery function r is idempotent if the following holds（不怎么理解）

$$\forall s, \boldsymbol{b}.\ r(s, \boldsymbol{true}) = r(r(s, \boldsymbol{b}), \boldsymbol{true}).$$

除了上述几个定义外，文章还给出了 Crash refinement with recovery、No-op、System crash refinement 几个定义，由于没有怎么理解，描述起来感觉有些困难，

所以如果想仔细研读的话，最好还是直接看看文章。

3.2 The verifier

这一节主要是用 3.1.2 里面的一系列定义对实现进行验证其正确性，具体描述过于繁杂，所以想要仔细了解还是去研读这篇文章。

3.3 The counterexample visualizer

为了使有效性的反例更容易理解，Yggdrasil 为异步磁盘模型提供了可视化工具。给定定义 4 中的公式的反例模型，可视化器如下生成具体的盘事件轨迹（例如 2.2.3）。首先，它使用 crash-schedule b1 来标识布尔变量，表示系统崩溃的位置，并将该位置与具有堆栈跟踪的实现源代码相关联。其次，它评估指示写入是否与磁盘同步的布尔同步变量，并用相应的源位置输出未决的写入，以帮助识别意外的重新排序。

3.4 The optimizer

Yggdrasil 优化器提高了实现代码的运行时性能。Yggdrasil 将优化器视为不可信，并重新验证其生成的优化代码。这个简单的设计，通过 push-button 成为可能，程序员可以插入自定义优化，而无需提供正确的证明。通过重写 Python 抽象语法树提供了一个内置的优化，贪婪地删除磁盘刷新操作（2.4 节）。

4. conclusion and discussion

文章还提到了其他系统的实现，如 Yxv6 file system，其他使用 Yggdrasil 构建的存储软件，由于其实现过程和章节二中 YminLFS 类似，只不过功能更复杂，加入了其他的机制，是一个更加完整的应用，所以不再重复，直接讨论结论和 Yggdrasil 的优劣。

首先，使用 Yggdrasil 构建的系统不支持并发，多线程，因为 Yggdrasil 属于单线程实现。再者，Yggdrasil 依靠 SMT 解算器进行自动化测试，仅限于一阶逻辑。更重要的是，Yggdrasil 核心在 SMT 解算器，所以他的正确性由 SMT 决定，这是个很大的限制，虽然，文章为了改进这点使用了其他两种解算器交叉验证，但还是存在一种局限性。

使用 Yggdrasil 好处就在于不需要手动验证代码的正确性，通常一个简单的文件系统要验证他的正确性需要遍历他所有可能的路径，对于人工证明来说是个很大的工作量；使用 Yggdrasil 能减少代码量，编写人员只需要集中在规范的实现上，而不需要验证低层的正确性。

使用 Yggdrasil 构建的系统代码量如图所示：

component	specification	implementation	consistency inv
Yxv6	250	1,500	5
YminLFS	25	150	5
Ycp	15	45	0
Ylog	35	60	0
infrastructure	–	1,500	–
FUSE stub	–	250	–

Figure 5: Lines of code for the Yggdrasil toolkit and storage systems built using it, excluding blank lines and comments.

除了代码量较少的优点之外，它们在 SSD 和 RAM disk 上面的表现也十分友好，比较是在使用 Yggdrasil 构建的 yxv6 文件系统和 FSCQ 与 ext4 之间进行的，结果如图所示：

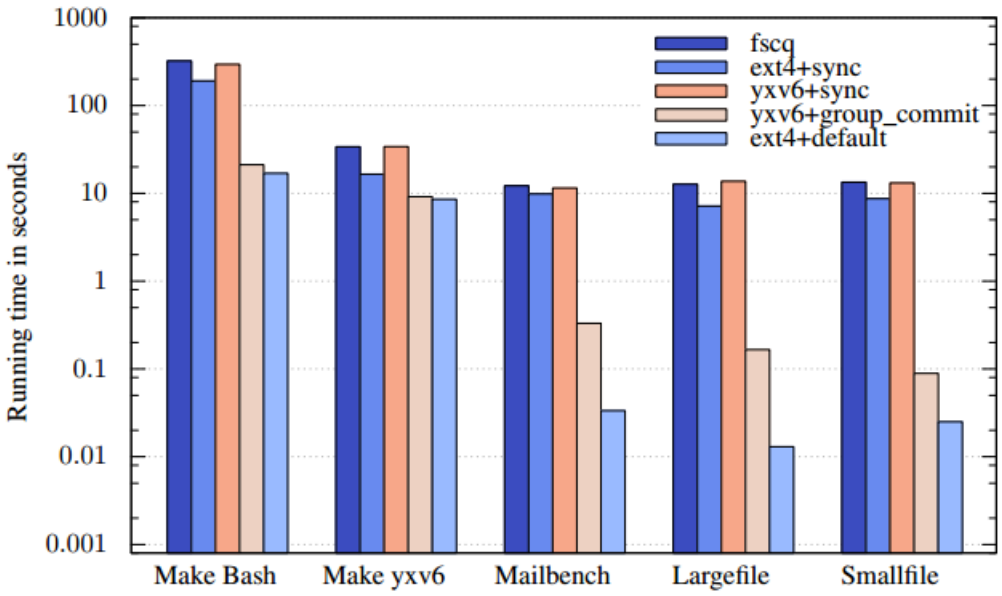


Figure 6: Performance of file systems on an SSD, in seconds (log scale; lower is better).

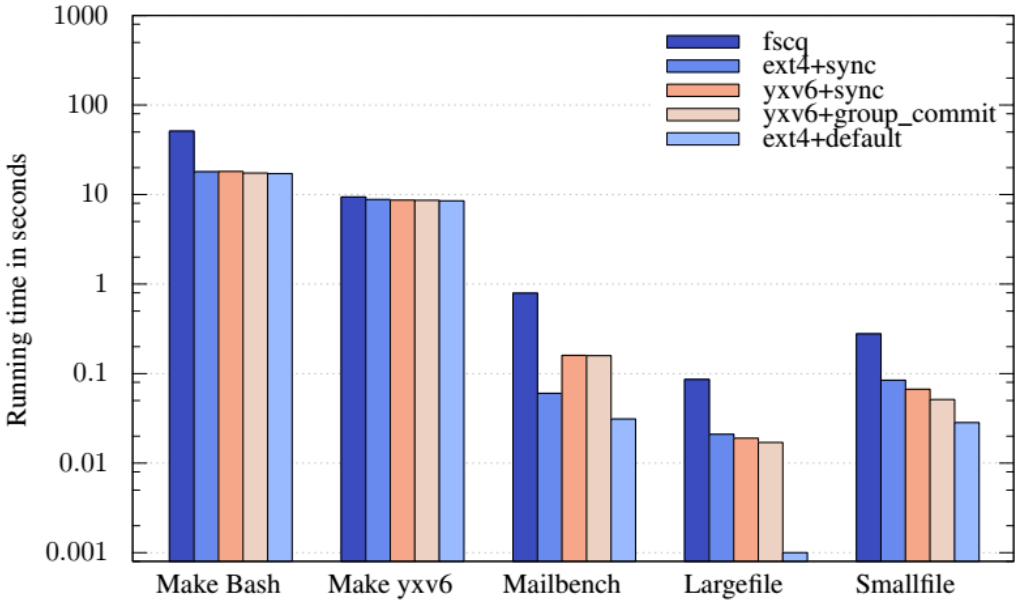


Figure 7: Performance of file systems on a RAM disk, in seconds (log scale; lower is better).

Yggdrasil 提出了一种构建文件系统的新方法。通过文件系统 crash-refinement 的定义来保证正确性，这种定义适合于高效的 SMT 解决方案。 它引入了一些扩展自动化验证的技术，包括抽象堆栈和数据表示分离，在减轻证明的负担同时提供了一个有力的正确性保证。