Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Evaluación de implementaciones alternativas de colas concurrentes en Haskell

Tesis de Licenciatura en Ciencias de la Computación

Tomás Abel González

Director: Hernán Melgratti

Buenos Aires, 2018

EVALUACIÓN DE IMPLEMENTACIONES ALTERNATIVAS DE COLAS CONCURRENTES EN HASKELL

En este trabajo se realiza una comparación entre distintas maneras de implementar un mismo tipo de datos concurrente en el lenguaje de programación Haskell. El lenguaje provee varias alternativas para resolver los problemas de sincronización que surgen dentro del área de la programación concurrente. Entre ellas, el trabajo se enfoca en las variantes libres de *locks* como el uso de la primitiva *compare and set* y la librería STM.

Se llevo a cabo una experimentación para observar las diferencias entre las distintas implementaciones y se analizaron los resultados para determinar cuales son las implementaciones más apropiadas según varios contextos de uso. Para el análisis también se toma en cuenta la complejidad de los algoritmos y la consistencia en los resultados que producen.

Palabras claves: memoria transaccional, programación concurrente, Haskell, algoritmos optimistas

Índice general

1. INTRODUCCIÓN

En el área de la programación concurrente, un programador tiene varias alternativas al momento de implementar una estructura. En principio, uno puede utilizar estructuras como *locks* para controlar que un recurso compartido sea utilizado de una manera correcta que no sea sujeta a problemas tales como las condiciones de carrera.

Sin embargo, este tipo de algoritmos que utilizan locks pueden no ser la mejor opción en varios casos, ya que se puede perder poder de procesamiento cuando hay hilos de ejecución que deben frenar su ejecución para esperar a que un recurso sea liberado a través de su *lock*.

Ante este problema se idearon alternativas en la forma de algoritmos libres de locks. Una alternativa son los algoritmos denominados como optimistas, que no detienen la ejecución de un hilo y utilizan la primitiva de sincronización *compare and set* (CAS) [3] para modificar un recurso compartido, reintentando en el caso de que falle la comparación. Otra alternativa que ha ganado popularidad como práctica para la programación concurrente es el uso de memoria transaccional (STM) [12] donde se permite componer distintas acciones para que luego se realicen de manera atómica y así poder obtener una sincronización correcta de las modificaciones al recurso. Este método de sincronización permite al programador abstraerse del manejo de *locks* que pueden ser difíciles de mantener y resulta más intuitivo que los algoritmos optimistas para implementar e interpretar.

Dada la variedad de maneras de implementar una misma estructura de datos o interfaz, han surgido varios trabajos que comparan distintas variantes de implementación para comprender qué manera es preferible dependiendo del contexto de uso. Por ejemplo, en el trabajo de [8] se implementaron distintas versiones de una tabla hash sobre el lenguaje Haskell para luego realizar experimentos sobre ellas, de una manera similar al trabajo de [13] que tomó una lista enlazada como la estructura de datos a investigar.

En este trabajo se presentan distintas implementaciones para un mismo tipo de datos (pila) concurrente en el lenguaje de programación Haskell. Se presentan tres estructuras de datos distintas para representar la pila concurrente, dos de las cuales utilizan algoritmos optimistas y se encuentran presentadas en [11] con implementaciones en código Java, mientras que la restante es una implementación que aprovecha las operaciones de la librería STM de Haskell para facilitar la implementación. Además, para las estructuras con algoritmos optimistas, también se puede utilizar la librería STM para implementar una versión de CAS sobre STM y que los algoritmos optimistas utilicen variables transaccionales, propias de la librería, similar a como se describe en los trabajos [8] [13] sobre estructuras de tablas hash y listas enlazadas. Esto resulta en un total de cinco implementaciones distintas para una pila de datos concurrente.

El objetivo del trabajo es luego realizar comparaciones del desempeño de las distintas estructuras y variantes con una serie de experimentos. Luego, analizar cómo las distintas implementaciones difieren en determinadas situaciones para poder distinguir cuál resulta favorable dependiendo del caso de uso, teniendo en cuenta los aspectos de implementación.

En el capítulo 2 se presentan los conceptos preliminares necesarios para el entendimiento de las distintas implementaciones: el uso de Haskell como lenguaje imperativo, algoritmos optimistas, y las primitivas de sincronización utilizadas. Luego se detalla la implementación y el comportamiento de las distintas estructuras de datos para represen-

tar una pila concurrente en el capítulo 3. Finalmente, los detalles de la experimentación realizada y sus resultados se presentan en el capítulo 4.

2. PRELIMINARES

2.1. Programación imperativa en Haskell

Al ser un lenguaje de programación funcional, no es común relacionar los conceptos de lenguajes imperativos con Haskell. No obstante, el lenguaje provee funcionalidades interesantes para utilizar el lenguaje de manera similar a un lenguaje imperativo sin dejar de ser funcional. Esto se logra con el uso de la mónada IO y un azúcar sintáctico conocido como *do notation*[9].

2.1.1. Mónadas

Una mónada es un tipo de datos que permite representar formas específicas de computaciones y capturar efectos. Estas tareas incluyen manejo de excepciones, mantener un estado, generar de números aleatorios, y realizar acciones de entrada y salida.

El comportamiento de una mónada se define por dos funciones: `return` y el operador *bind* `(>>=)` que se detallan en la Fig. 2.1. La función `return` se encarga de convertir un valor pasado por parámetro a un valor mónadico mientras que la función *bind* se ocupa de componer dos acciones monádicas, pasando el resultado de la primera acción (primer parámetro) a la próxima. Más información sobre estas funciones se encuentra detallada en la documentación de la librería Control.Monad [5].

Una de las mónadas más utilizadas es la mónada IO. Esta permite implementar acciones de entrada/salida como, por ejemplo, imprimir por pantalla o generar números aleatorios.

En la Fig. 2.2 se muestra el código para una función `randomSum` que genera dos números aleatoriamente e imprime la suma por pantalla. Para la generación de números aleatorios se utiliza la función `randomIO` de la librería `random` de Haskell [15]. El código de la función consiste en generar un número aleatorio con una llamada a la función `randomInt` y pasar el resultado a una segunda función utilizando el operador `(>>=)`. La segunda función tomará el resultado de `randomInt` como parámetro `rand1`, realizará una nueva llamada a `randomInt` y pasa el resultado (`rand2`) a una última función que imprime el resultado de la suma de `rand1` y `rand2`.

Como alternativa para la implementación de acciones monádicas, el lenguaje Haskell provee lo que se conoce como *do-notation* que permite escribir código siguiendo un estilo imperativo que luego sería interpretado como una composición de funciones. Esta notación es simplemente un azúcar sintáctico para facilitar la implementación de acciones monádicas como la presentada en la Fig. 2.2. Se puede apreciar cómo la implementación de esta función se facilita gracias a la notación `do` en la Fig. 2.3.

En [9] se describe en detalle cómo este azúcar sintáctico logra traducir lo que se escribe como código imperativo a código funcional a nivel de compilación.

2.2. Algoritmos optimistas

Los algoritmos optimistas actúan sobre un recurso compartido bajo la suposición de que no ocurrirán conflictos entre los distintos hilos de ejecución y, por lo tanto no adquieren

```
1 return :: a -> m a
2 (>>=)  :: m a -> (a -> m b) -> m b
```

Fig. 2.1: Tipado de las funciones de una mónada `m`

```
1 randomInt :: IO Int
2 randomInt = randomIO :: IO Int
3
4 printRandomSum :: IO ()
5 printRandomSum = randomInt >>= (\ rand1 -> randomInt >>= (\ rand2 -> print (rand1 +
  ↪  rand2)))
```

Fig. 2.2: Ejemplo de acción monádica `printRandomSum`

locks mientras acceden al recurso. Sin embargo, requiren verificar que ningun otro hilo ha interferido antes de consolidar sus modificaciones. Para ello, se basan en el uso de acciones atomicas, tales como CAS para comparar estado y modificarlo simultaneamente. En el caso que la llamada a CAS falle, el algoritmo vuelve a ejecutarse hasta que el recurso sea modificado con éxito.

Esta clase de algoritmos permiten que los distintos hilos de ejecución no deban frenar su ejecución para esperar la liberación de un recurso, como es el caso en algoritmos que utilizan algún tipo de lock sobre un recurso. De esta manera, uno puede obtener mejores tiempos de ejecución ya que los tiempos de espera son eliminados.

En la Fig. 2.4 se presenta, a modo de ejemplo, una estructura de datos `Counter` representada por una referencia a un entero y un algoritmo optimista `inc` para incrementar su valor. En el algoritmo, primero se declara una referencia booleana `loop` para mantener un ciclo con la función de Haskell `whileM_`. En el ciclo se realiza una lectura del valor del contador, guardando el resultado en la variable `oldValue`, para luego realizar la llamada a la función `atomCAS`.

Esta llamada realiza una nueva lectura del valor del contador y lo compara con `oldValue` para determinar si es correcto modificar el valor. Si el valor leído coincide con el de `oldValue` se modifica el valor del contador con el valor de `newValue`, retornando `True` como resultado de la función. Si no, significa que un hilo de ejecución distinto al actual logró modificar el valor del contador luego de que el hilo actual lea el valor en la línea 7 del código. En este caso no se realizan cambios y la llamada a `atomCAS` retornará `False`. Es importante notar que todas las acciones de la función `atomCAS` son realizadas de manera atómica.

Según el valor de retorno, el algoritmo de la función `inc` terminará la ejecución modificando el valor de la referencia `loop` a `False`, deteniendo el ciclo, o volverá a iniciar el ciclo como se ve en el condicional `if` de la línea 10. La línea 12 de la Fig. 2.4 que llama a la función `return` con parámetro `()` no realiza ninguna operación y simplemente permite volver a iniciar el ciclo.

En este trabajo se utiliza este esquema de algoritmos en varias de las implementaciones. También se presentan variantes para la implementación de algoritmos de este estilo ya que es posible definirlos utilizando distintas primitivas de sincronización para implementar distintas versiones de `atomCAS`. Estas versiones se detallan en la sección 2.3.

```haskell
randomInt :: IO Int
randomInt = randomIO :: IO Int

printRandomSum :: IO ()
printRandomSum = do
  rand1 <- randomInt
  rand2 <- randomInt
  print (rand1 + rand2)
```

Fig. 2.3: Código de `printRandomSum` reescrito con notación `do`

```haskell
data Counter = Ref Int

inc :: Counter -> IO ()
inc counter = do
    loop <- newRef True
    whileM_ (readRef loop) $ do
        oldValue <- readRef counter
        let newValue = oldValue + 1
        success <- atomCAS counter oldValue newValue
        if success
            then writeRef loop False
            else return ()
```

Fig. 2.4: Algoritmo optimista para incrementar un contador

2.3. Primitivas de sincronización

El lenguaje de programación Haskell ofrece distintas primitivas de sincronización para poder manejar recursos compartidos entre distintos hilos de ejecución. En este trabajo se utilizan los tipos de datos `IORef` y `TVar` de la librería `STM` para las implementaciones que se detallan en el próximo capítulo. Estos tipos de datos son distintas maneras de referenciar objetos mutables, similares a punteros en otros lenguajes de programación, y proveen funciones para ser modificadas de manera atómica.

2.3.1. IORef

El tipo de datos `IORef` es comunmente utilizado en programas Haskell como manera de referenciar objetos mutables. En la Fig. 2.5 se detallan las funciones para crear, leer, escribir y modificar atómicamente los valores contenidos en una referencia `IORef`.

La función `atomicModifyIORef` de la última línea de la Fig. 2.5 es la única función que realiza una modificación sobre una referencia del tipo `IORef` de manera atómica. La función recibe como parámetros una referencia a modificar y una función que cumple dos tareas: modificar el valor contenido en la referencia y luego retornar un valor en función del valor previo contenido en la referencia como resultado de `atomicModifyIORef`. El código que presenta en la Fig. 2.6, previamente presentado en [13], muestra un comportamiento equivalente al de `atomicModifyIORef`.

Para implementar una versión de `atomCAS` es crucial contar con una función como `atomicModifyIORef` ya que es la única función que permite manipular referencias de tipo `IORef` de manera atómica. Generalmente, una función de CAS como `atomCAS` debe recibir

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO a
atomicModifyIORef :: IORef a -> (a -> (a,b)) -> IO b
```

Fig. 2.5: Funciones para `IORef`

```
atomicModifyIORef r f = do
   a <- readIORef r
   let p = f a
   writeIORef r (fst p)
   return (snd p)
```

Fig. 2.6: Comportamiento de la función `atomicModifyIORef`

tres parámetros: la referencia a modificar, el valor previamente leído a comparar con el valor actual de la referencia atómicamente, y el valor nuevo a guardar en la referencia. La función debe retornar un booleano que indica si la comparación entre el valor leído y el valor actual de la referencia es exitosa.

En este caso, para aplicar `atomCAS` sobre una referencia de tipo `IORef`, utilizaremos la función de `atomicModifyIORef` pasándole como segundo parámetro una función que compare el valor leído con el actual y retorne una tupla con el valor a guardar en la referencia y el booleano resultante de la comparación. Si la operación es exitosa, se guarda el valor nuevo en la referencia. En caso contrario, se vuelve a guardar el valor actual. Este comportamiento se refleja en el código de la Fig. 2.7, también presentado en [13].

Implementada esta función, se puede reescribir el código del contador como se muestra en la Fig. 2.8, utilizando `IORef` como referencia y las funciones `readIORef`, `writeIORef`, y `atomCASIO`.

2.3.2. STM

La librería STM de Haskell permite al programador componer distintas funciones a ser ejecutadas de manera atómica. Para lograr esto, la librería provee el tipo de mónada STM para distinguir las funciones que se realizan dentro de esta mónada con la mónada IO y el tipo de referencia `TVar`. Es decir, las funciones de la mónada STM sólo pueden componerse con funciones de la misma mónada o funciones puras (que no causan efectos al contexto de ejecución). Una vez realizada la composición de funciones, se llama a una función `atomically` para que las acciones realizadas dentro de la mónada STM sean realizadas atómicamente y sus efectos sean visibles al nível de la mónada IO.

Intuitivamente, uno puede interpretar a la composición de funciones STM como una transacción cuyos cambios realizados sólo pueden ser visibles una vez que se completa. Dentro de una transacción, se debe utilizar un tipo de referencia distinto de `IORef`, ya que sus funciones no pueden invocarse dentro de la mónada STM. Para esto se tiene el tipo de variable transaccional `TVar` y funciones para su escritura y lectura. Las funciones provistas por la librería STM en la Fig. 2.9.

La librería STM también provee unas funciones `retry` y `orElse` que permiten manipular el flujo de una transacción para que se realice un reintento o ejecutar una transacción en vez de otra en caso de que no sea exitosa. Sin embargo, no fueron necesarias para

```haskell
atomCASIO :: Eq a => IORef a -> a -> a -> IO Bool
atomCASIO ptr old new =
  atomicModifyIORef ptr (\ cur -> if cur == old
                                  then (new, True)
                                  else (cur, False))
```

Fig. 2.7: `atomCAS` utilizando IORef

```haskell
data Counter = IORef Int

inc :: Counter -> IO ()
inc counter = do
    loop <- newIORef True
    whileM_ (readIORef loop) $ do
        oldValue <- readIORef counter
        let newValue = oldValue + 1
        success <- atomCASIO counter oldValue newValue
        if success
            then writeIORef loop False
            else return ()
```

Fig. 2.8: Algoritmo optimista para incrementar un contador utilizando `IORef`

las implementaciones que se realizaron en este trabajo que se encuentran detalladas en el próximo capítulo.

Dadas estas herramientas, se puede realizar una nueva implementación de la función `atomCAS` sobre STM. El código de la función también fue presentado en [13].

Dadas estas herramientas, uno puede implementar una versión del contador sobre STM utilizando `TVar` como tipo de referencia y utilizando las funciones `atomCASSTM`, y `readTVar`.

Es importante notar que en el código de la Fig. 2.11 la mónada continúa siendo IO ya que la llamada a la función `readTVar` es invocada dentro de la función `atomically`.

Además de poder implementar variantes de algoritmos optimistas, podemos implementar nuestro contador de una manera distinta utilizando las funciones STM como se muestra en la Fig. 2.12.

Se puede apreciar cómo este estilo de algoritmos resultan más intuitivos para interpretar e implementar que sus versiones optimistas gracias a que STM nos permite componer las funciones que necesitamos que sean ejecutadas atómicamente en vez de realizar llamadas a CAS y realizar reintentos en caso de que falle.

```haskell
atomically :: STM a -> IO a
newTVar :: TVar a
readTVar :: a -> STM (TVar a)
writeTVar :: TVar a -> STM a -> a -> STM ()
```

Fig. 2.9: Funciones de la librería STM

```haskell
atomCASSTM :: Eq a => TVar a -> a -> a -> IO Bool
atomCASSTM ptr old new = atomically $ do
  cur <- readTVar ptr
  if cur == old
  then do
    writeTVar ptr new
    return True
  else return False
```

Fig. 2.10: `atomCAS` utilizando STM

```haskell
data Counter = TVar Int

inc :: Counter -> IO ()
inc counter = do
    loop <- newIORef True
    whileM_ (readIORef loop) $ do
        oldValue <- atomically $ readTVar counter
        let newValue = oldValue + 1
        success <- atomCASSTM counter oldValue newValue
        if success
            then writeIORef loop False
            else return ()
```

Fig. 2.11: Algoritmo optimista para incrementar un contador utilizando STM

```haskell
data Counter = TVar Int

inc :: Counter -> IO ()
inc counter = atomically $ do
    oldValue <- readTVar counter
    let newValue = oldValue + 1
    writeTVar counter newValue
```

Fig. 2.12: Algoritmo STM para incrementar un contador

3. IMPLEMENTACIÓN

Para este trabajo, se decidió experimentar sobre distintas implementaciones para un mismo tipo de datos abstracto. El tipo elegido, una pila, es representado por una referencia a un nodo que contiene un valor y una referencia al nodo que le sigue (que se encuentra debajo del nodo en la pila). Para el último nodo de la pila, la referencia al nodo siguiente será denominada como nula. El código de esta representación se presenta en la figura 3.1 con sus variantes de implementación IO y STM.

Las operaciones a realizar sobre una pila son las de apilar (`push`) y desapilar (`pop`). Todas las variantes de implementación que realicemos cumplen la misma interfaz para estas funciones, presentada en la figura 3.2.

La función `push` inserta un nuevo nodo al tope de la pila con un valor recibido por parámetro. Por el otro lado, la función pop lee el valor que se encuentra en el tope de la pila y remueve el nodo, retornando el valor contenido en el nodo como resultado. En el caso que no haya ningún elemento en la pila, la ejecución deberá arrojar una excepción.

En este capítulo se encuentran detalladas las distintas implementaciones utilizadas para la evaluación de performance: `LockFreeStack` (sección 3.1), `EliminationBackoffStack` (sección 3.2), y `StackSTM` (sección: 3.3). Las primeras dos estructuras son maneras conocidas para la implementación de una pila concurrente y se encuentran detalladas en [11] que hacen uso de la primitiva de sincronización *compare and set*. La estructura restante es una implementación *naive* de una pila concurrente utilizando las herramientas provistas la librería STM de Haskell.

En el código utilizado para la experimentación, cada versión de `push` y `pop` es nombrada de manera que se puede identificar a qué implementación pertenece. Por ejemplo, las funciones de un `LockFreeStack` implementado sobre IO tienen los nombres `pushLFSIO` y `popLFSIO`.

3.1. LockFreeStack

La estructura de datos `LockFreeStack` (LFS), también conocida como TreiberStack [16], consiste de una pila concurrente libre de locks implementada utilizando algoritmos optimistas. Sus algoritmos consisten en intentar realizar la operación sin tener que pausar la ejecución para esperar un recurso y si hay una falla en el intento, reintentar. En este caso, los algoritmos realizan un backoff previo a realizar un reintento.

La representación de un `LockFreeStack` consiste de una referencia al tope de la pila y una estructura auxiliar de backoff que será la encargada de determinar la cantidad de tiempo que un hilo debe esperar antes de reintentar.

La idea detrás de los algoritmos es que cada hilo de ejecución intente realizar una operación de escritura o lectura sobre la pila. Si el hilo no es exitoso en el intento, debe volver a intentar luego de esperar una cantidad de tiempo determinada por una estructura de backoff. Este comportamiento se puede apreciar en la figura 3.4 donde se muestra el código de las funciones `pushLFSIO` y `popLFSIO`. En ellas, se encuentran las llamadas a las funciones `tryPushIO` y `tryPopIO` respectivamente que serán detalladas en la subsección 3.1.2, así también la llamada a la función `backoff` que se detalla en la subsección 3.1.1.

```haskell
data NodeIO a = NdIO { val :: a, next :: IORef (NodeIO a) } | Null
data NodeSTM a = NdSTM { val :: a, next :: TVar (NodeSTM a) } | Null
```

Fig. 3.1: Representación de un nodo en Haskell sobre IO y sobre STM

```haskell
push :: Eq a => Stack a -> a -> IO ()
pop :: Eq a => Stack a -> IO a
```

Fig. 3.2: Interfaz para las funciones `push` y `pop`

3.1.1. Backoff

En la implementación, la estructura de backoff es del tipo de backoff exponencial. Al realizar la operación de backoff, la estructura toma un número aleatorio sobre un rango entre 0 y el límite establecido por parámetro y detiene la ejecución del thread por esa cantidad de milisegundos. Luego la estructura duplica el valor del límite para la próxima vez que se realice la operación. Esta estructura de datos también se encuentra presentada en [11].

La especificación de la función `backoff` según [11] establece que el número generado aleatoriamente sea la cantidad de milisegundos que debe esperar el hilo de ejecución. Por eso, es necesario convertir esa cantidad a microsegundos, tal como aparece en la última linea de la figura 3.5, ya que la función `threadDelay` de Haskell toma microsegundos como parámetro [4].

3.1.2. Intento de escritura o lectura

El intento de escritura o lectura se ve implementado en las funciones `tryPush` y `tryPop` respectivamente. Ambas siguen la misma idea: observar el valor del tope de la pila y luego ejecutar CAS comparando lo que observa actualmente con lo observado en primer lugar. Para el caso del intento de apilar, se retorna el resultado de la ejecución de CAS, y para el intento de desapilar se retorna `Null` en el caso de fracaso, o el valor leído del tope de la pila en caso de éxito.

En el caso particular en la que un hilo desee remover un objeto en la pila, se debe realizar una verificación que la pila no se encuentre vacía. Si no hay ningún objeto para remover, la ejecución arroja una excepción `EmptyException`.

3.2. EliminationBackoffStack

Un `EliminationBackoffStack` (EBS) es una extensión para un `LockFreeStack` cuya idea principal es aprovechar el tiempo previo a un reintento para intentar un intercambio con otro hilo que esté esperando para ejecutar la operación inversa. Es decir, un hilo lector intentará realizar un intercambio con un hilo escritor y vice versa.

Para lograr este objetivo, se necesita de tres estructuras de datos auxiliares: `LockFreeExchanger`, `EliminationArray`, y `RangePolicy`. La representación de estas estructuras en sus variantes IO y STM se ve reflejada en la figura 3.9

La estructura EBS fue propuesta en [10] como una alternativa de mejor escalabilidad al LFS y luego mencionada con ejemplos de código Java en [11]. Estos últimos ejemplos

```
data LockFreeStackIO a = LFSIO { top :: IORef (NodeIO a), backoffLFS :: Backoff }
pushLFSIO :: Eq a => LockFreeStackIO a -> a -> IO ()
popLFSIO :: Eq a => LockFreeStackIO a -> IO a

data LockFreeStackSTM a = LFSSTM { top :: TVar (NodeSTM a), backoffLFS :: Backoff }
pushLFSSTM :: Eq a => LockFreeStackSTM a -> a -> IO ()
popLFSSTM :: Eq a => LockFreeStackSTM a -> IO a
```

Fig. 3.3: Representación de un LFS en sus implementaciones IO y STM y sus operaciones

de código son la base para la implementación que se detalla a continuación.

3.2.1. LockFreeExchanger

El `LockFreeExchanger` (LFE) consiste de una referencia que guarda información de estado (tipo de datos `State`) y un valor a intercambiar. El valor a intercambiar es representado por el tipo `Maybe` para poder representar el caso en el que el LFE esté vacío y no haya hilos tratando de realizar un intercambio. El estado puede tomar 3 posibles valores:

- `EMPTY`: no hay ningún valor asociado a la referencia
- `WAITING`: un hilo ha escrito un valor en la referencia y está esperando a que un hilo lo lea
- `BUSY`: un hilo a leído el valor que se encuentra en la referencia y escribe un valor para intercambiar

La única función que realiza esta estructura de datos es la operación `exchange`. El método consiste en leer el estado del LFE y realizar la acción correspondiente al estado. La operación debe ocurrir dentro de una ventana de tiempo que es establecida mediante un parámetro `timeout` al invocar la función.

Para la implementación de la función `exchange` utilizamos una función auxiliar `getSlot` para simular el comportamiento de la operación `get` de la clase `AtomicStampedReference` [2], ya que la componente `slot` de un LFE tiene este tipo en el código Java que se encuentra en [11]. La función toma una referencia a una tupla de tipo `(Maybe a,State)` (`slot`) y una referencia `IORef` a un valor de tipo `State` (`stampHolder`). La función `getSlot` se encarga de leer la tupla de la referencia `slot` para luego escribir su estado en la referencia `stampHolder` y retornar el valor de la tupla como resultado de la función. Los tipos de la función en sus variantes IO y STM en la figura y sus código se encuentran en las figuras 3.12 y 3.13 respectivamente.

A continuación se lista el comportamiento del algoritmo según los distintos casos del estado que tome el LFE. En cada ítem se notarán las líneas de código de la figura 3.14 en las que se puede apreciar el comportamiento detallado.

- `EMPTY`: el hilo actual intenta escribir el valor con una llamada a `atomCAS` el valor y el estado pasa a `WATING` en caso de éxito. Si la llamada a `atomCAS` falla (retorna `False`), se vuelve al inicio del ciclo en la línea 9.

  Si la llamada a `atomCAS` es exitosa, se espera a que el estado pase a `BUSY` dentro del `timeout` establecido. Esto se modela con el ciclo con la condición `emptyCaseLoopCondition` en las líneas 21 a 30.

```haskell
pushLFSIO lfs value = do
  ret <- newIORef True
  node <- newNodeIO value
  whileM_ (readIORef ret) $ do
    b <- tryPushIO lfs node
    if b
      then writeIORef ret False
      else backoff $ backoffLFS lfs
```

(a) Código `pushLFS` sobre IO

```haskell
pushLFSSTM lfs value = do
  ret <- newIORef True
  node <- newNodeSTM value
  whileM_ (readIORef ret) $ do
    b <- tryPushSTM lfs node
    if b
      then writeIORef ret False
      else backoff $ backoffLFS lfs
```

(b) Código `pushLFS` sobre STM

```haskell
popLFSIO lfs = do
  ret <- newIORef True
  res <- newIORef Nothing

  whileM_ (readIORef ret) $ do
    returnNode <- tryPopIO lfs
    if returnNode /= Null
      then do
        writeIORef res $ Just (val
        ↪  returnNode)
        writeIORef ret False
      else backoff (backoffLFS lfs)

  readIORef res >>= return.fromJust
```

(c) Código `popLFS` sobre IO

```haskell
popLFSSTM lfs = do
  ret <- newIORef True
  res <- newIORef Nothing

  whileM_ (readIORef ret) $ do
    returnNode <- tryPopSTM lfs
    if returnNode /= Null
      then do
        writeIORef res $ Just (val
        ↪  returnNode)
        writeIORef ret False
      else backoff (backoffLFS lfs)

  readIORef res >>= return.fromJust
```

(d) Código `popLFS` sobre STM

Fig. 3.4: Funciones de `LockFreeStack`

Si el estado pasa a `BUSY` antes de que se cumpla el timeout, significa que otro hilo ha leído el valor y está escribiendo el suyo para intercambiar con el hilo actual. Luego el hilo actual lee el valor escrito por el otro, dejando el valor del LFE en `Nothing` y su estado en `EMPTY`, y termina la ejecución de la función (líneas 24 a 30). Para esta función, finalizar la ejecución significa escribir el valor `False` en la variable `ret` para no ejecutar nuevamente el ciclo y escribir el valor de retorno en la varible `res` para después leerla en la última línea (55) de la función.

Si el estado no pasa a `BUSY` en el ciclo de las líneas 21 a 30, la ejecución saldrá del ciclo en el momento que el tiempo de ejecución supere el `timeout` establecido. Luego se verificará con una llamada a `atomCAS` (línea 35) que no haya ocurrido un cambio a último momento en el LFE para que vuelva a su estado inicial `EMPTY` con valor `Nothing`. Sin embargo, si hay un cambio significa que otro hilo ha llegado a leer el valor a último momento y se puede realizar el intercambio y finalizar la ejecución como se ve en las líneas 40 a 42.

- `WAITING`: el hilo actual lee el valor escrito y escribe el suyo, luego termina la ejecución de la función. Este comportamiento se ve reflejado en las líneas 45 a 52 de la figura 3.14.
- `BUSY`: en este caso, ya hay dos hilos realizando un intercambio. Por lo tanto, el hilo actual debe reintentar, volviendo al inicio del ciclo en la línea 9. Este comportamiento se ve reflejado en las líneas 53 y 54 de la figura 3.14, donde el algoritmo simplemente

```haskell
data Backoff = BCK {minDelay :: Int, maxDelay :: Int, limit :: IORef Int}

newBackoff :: Int -> Int -> IO Backoff
newBackoff min max = (newIORef min) >>= return.(BCK min max)

backoff :: Backoff -> IO ()
backoff b = do
  backoffLimit <- readIORef $ limit b
  delayInMilliseconds <- randomRIO (0, backoffLimit)
  writeIORef (limit b) (min (maxDelay b) (2 * backoffLimit))
  threadDelay (toMicroseconds delayInMilliseconds)
    where toMicroseconds = (*) 1000
```

Fig. 3.5: Estructura de backoff

```haskell
tryPushIO :: Eq a => LockFreeStackIO a -> NodeIO a -> IO Bool
tryPopIO :: Eq a => LockFreeStackIO a -> IO (NodeIO a)

tryPushSTM :: Eq a => LockFreeStackSTM a -> NodeSTM a -> IO Bool
tryPopSTM :: Eq a => LockFreeStackSTM a -> IO (NodeSTM a)
```

Fig. 3.6: Tipo de las funciones de intento de lectura y escritura en sus variantes IO y STM

llama a la función `return ()` para luego entrar a una nueva iteración del ciclo de ejecución.

```haskell
tryPushIO lfs node = do
  oldTop <- readIORef (top lfs)
  writeIORef (next node) oldTop
  atomCASIO (top lfs) oldTop node
```

(a) Código `tryPushIO`

```haskell
tryPushSTM lfs node = do
  oldTop <- atomically $ readTVar (top lfs)
  atomically $ writeTVar (next node) oldTop
  atomCASSTM (top lfs) oldTop node
```

(b) Código `tryPushSTM`

```haskell
tryPopIO lfs = do
  oldTop <- readIORef (top lfs)
  if oldTop == Null
    then
      throw EmptyException
    else do
      newTop <- readIORef (next oldTop)
      b <- atomCASIO (top lfs) oldTop newTop
      if b
        then return oldTop
        else return Null
```

(a) Código `tryPopIO`

```haskell
tryPopSTM lfs = do
  oldTop <- atomically $ readTVar (top lfs)
  if oldTop == Null
    then
      throw EmptyException
    else do
      newTop <- atomically $ readTVar (next oldTop)
      b <- atomCASSTM (top lfs) oldTop newTop
      if b
        then return oldTop
        else return Null
```

(b) Código `tryPopSTM`

Fig. 3.8: Variantes de implementación para los intentos de escritura y lectura de un LFS

El código de la variante `exchangeSTM` es muy similar al que se muestra en la figura 3.14. La diferencia se encuentra en los renombres de las funciones `getSlotSTM` y `atomCASSTM`, junto con la diferencia en cómo los algoritmos escriben la componente `slot` del LFE. En el caso de IO, vemos en el código que basta con un simple `writeIORef`, mientras que en STM, esta llamada es reemplazada con la llamada a la función `writeTVar` dentro de un bloque STM englobado por la función `atomically` para que los cambios se vean reflejados en el contexto de ejecución.

Para la medición de tiempos se utilizó la función `getTime` de la librería `clock` para Haskell [14]. El comportamiento de esta función se explica más adelante en la subsección 4.1.3 del capítulo 4 ya que la función es utilizada para medir los tiempos de ejecución de la experimentación.

3.2.2. EliminationArray

Un `EliminationArray` es una simple estructura de datos que mantiene un arreglo de instancias de `LockFreeExchanger` y la información pertinente a la duración del `timeout` que deberá cumplir cada una de las llamadas a `exchange` que se realicen. La estructura tiene sólo una función `visit` que elige aleatoriamente cual de los exchangers de su arreglo utilizar para realizar el intercambio. Luego ejecuta la función `visit` para el exchanger elegido.

En la figura 3.15 se muestra el código para un `EliminationArrayIO`. El código es idéntico al de un `EliminationArraySTM`, excepto por los renombres correspondientes a las funciones `exchangeSTM` (línea 6), `EASTM` (línea 1), y `visitSTM` junto con el tipado `LockFreeExchangerSTM` de `exchanger` en la línea 1.

3.2.3. RangePolicy

```
1 data EliminationBackoffStackIO a = EBSIO { top :: IORef (NodeIO a), capacity :: Int,
  ↪ eliminationArray :: EliminationArrayIO a, policy :: TLS RangePolicy}
2 data EliminationArrayIO a = EAIO {exchanger :: [LockFreeExchangerIO a], duration ::
  ↪ Integer}
3 data LockFreeExchangerIO a = LFEIO {slot :: IORef (Maybe a, State)}
4
5 data EliminationBackoffStackSTM a = EBSSTM {top :: TVar (NodeSTM a), capacity :: Int,
  ↪ eliminationArray :: EliminationArraySTM a, policy :: TLS RangePolicy}
6 data EliminationArraySTM a = EASTM {exchanger :: [LockFreeExchangerSTM a], duration ::
  ↪ Integer}
7 data LockFreeExchangerSTM a = LFESTM {slot :: TVar (Maybe a, State)}
```

Fig. 3.9: Variantes de implementación para las estructuras necesarias para un EBS

```
1 data State = EMPTY | WAITING | BUSY
```

Fig. 3.10: Tipo de datos `State`

Para poder optimizar las probabilidades de lograr un intercambio en ese tiempo de espera. Un EBS cuenta con otra estructura adicional conocida como `RangePolicy`. Esta estructura tiene como función manejar el rango del `EliminationArray` en el cual se eligirá el exchanger a utilizar.

La idea de esta política es aumentar el rango a medida que se realizan intercambios con éxito y reducirlo cuando falle. Al reducir el rango aumenta la probabilidad de que dos hilos elijan la misma posición del arreglo para realizar el intercambio, y al incrementarlo se permite que más intercambios sucedan al mismo tiempo si es que hay suficientes hilos como para cubrir las posiciones del arreglo.

Es importante notar que según [11], cada hilo de ejecución debe tener su propio `RangePolicy`. Para esto utilizamos la librería `thread-local-storage` [6] de Haskell para permitir que cada hilo tenga su propia instancia de `RangePolicy`. Esto se muestra en la línea 1 de la figura 3.9 donde la componente `policy` tiene tipo `TLS RangePolicy`.

3.2.4. Estrategia de backoff del EliminationBackoffStack

La estructura `EliminationBackoffStack` (EBS) tiene un comportamiento muy similar a un `LockFreeStack`. La diferencia se encuentra en las acciones que cada estructura realiza cuando la operación inicial falla, y por lo tanto debe esperar antes de reintentar. En el caso del LFS, el hilo debe esperar un tiempo sin realizar ninguna acción y reintenta, mientras que en un EBS este intenta aprovechar el tiempo de espera para realizar un intercambio con otro hilo que haya fallado su intento.

Luego, en las funciones de apilar y desapilar, se intenta realizar la operación con las mismas funciones que utiliza un LFS, y en el caso de que haya una falla, se obtiene el rango dictado por el `RangePolicy` y se llama a la función `visit` del `EliminationArray` con el rango pasado como argumento.

En el código que se encuentra en las figuras 3.17 y 3.18 se puede ver que a cada llamada de la función `visitIO` se la rodea por la función `catch` para poder analizar si hubo excepciones arrojadas. De esta manera, el algoritmo determina si el intercambio fue exitoso y modifica la política de rango con las llamadas a las funciones `recordEliminationTimeout` en caso de fracaso y `recordEliminationSuccess` en caso de éxito. También se puede ver

```
exchangeIO :: (Eq a) => LockFreeExchanger a -> Maybe a -> Integer -> IO (Maybe a)
exchangeSTM :: (Eq a) => LockFreeExchanger a -> Maybe a -> Integer -> IO (Maybe a)
```

Fig. 3.11: Tipado de la función exchange en sus variantes IO y STM

```
getSlotIO :: IORef (Maybe a, State) -> IORef State -> IO (Maybe a)
getSlotSTM :: TVar (Maybe a, State) -> IORef State -> IO (Maybe a)
```

Fig. 3.12: Tipos de las funciones `getSlot` en sus variantes IO y STM

el uso de las funciones `tryPushIO` y `tryPopIO` que son las mismas funciones que se utilizan en un LFS, detalladas en la subsección 3.1.2.

El código para estas funciones es idéntico en la variante STM del EBS, exceptuando los renombres de las funciones y estructuras utilizadas en las distintas estructuras como `EliminationBackoffStackSTM`, `newNodeSTM`, `visitSTM`, `tryPopSTM`, y `tryPushSTM`.

3.3. StackSTM

La última estructura de datos implementada es `StackSTM`. La implementación para esta estructura intenta aprovechar lo más posible las herramientas que provee la librería STM de Haskell.

El código para esta implementación se ve en la figura 3.19. Se puede apreciar cómo algoritmos para `pushStackSTM` y `popStackSTM` son reducidos a pocas líneas de código comparado a sus otras implementaciones ya que la librería STM permite englobar un bloque de código transaccional dentro de una llamada a la función `atomically` que asegura que el código será ejecutado atómicamente sin posibilidad que haya conflictos con otros hilos de ejecución.

```haskell
getSlotIO slot stampHolder = do
  (val, state) <- readIORef slot
  writeIORef stampHolder state
  return val
```

```haskell
getSlotSTM slot stampHolder = do
  (val, state) <- atomically $ readTVar slot
  writeIORef stampHolder state
  return val
```

Fig. 3.13: Código para `getSlot` en sus variantes IO y STM

```haskell
exchangeIO :: (Eq a) => LockFreeExchangerIO a -> Maybe a -> Integer -> IO (Maybe a)
exchangeIO lfe myItem timeout = do
  ret <- newIORef True
  res <- newIORef Nothing
  let nanos = timeout * (10 ^ 6) -- timeout unit is millisecs
  timeBound <- systemNanoTime >>= return.((+) nanos)
  stampHolder <- newIORef EMPTY
  whileM_ (readIORef ret) $ do
    timeoutDone <- systemNanoTime >>= return.((<) timeBound)
    if timeoutDone
      then do
        throw TimeoutException
      else do
        yrItem <- getSlotIO (slot lfe) stampHolder
        stamp <- readIORef stampHolder
        case stamp of
          EMPTY -> do
            b <- atomCASIO (slot lfe) (yrItem, EMPTY) (myItem, WAITING)
            if b
              then do
                whileM_ (emptyCaseLoopCondition ret timeBound) $ do
                  yrItem <- getSlotIO (slot lfe) stampHolder
                  stampBusy <- (readIORef stampHolder) >>= return.((==) BUSY)
                  if stampBusy
                    then do
                      writeIORef (slot lfe) (Nothing, EMPTY)
                      writeIORef ret False
                      writeIORef res yrItem
                    else
                      return ()
                breakFromWhile <- readIORef ret >>= return.not
                if breakFromWhile
                  then return ()
                  else do
                    b <- atomCASIO (slot lfe) (myItem, WAITING) (Nothing, EMPTY)
                    if b
                      then do
                        throw TimeoutException
                      else do
                        yrItem <- getSlotIO (slot lfe) stampHolder
                        writeIORef (slot lfe) (Nothing, EMPTY)
                        writeIORef res yrItem
              else do
                return ()
          WAITING -> do
            b <- atomCASIO (slot lfe) (yrItem, WAITING) (myItem, BUSY)
            if b
              then do
                writeIORef ret False
                writeIORef res yrItem
              else
                return ()
          BUSY -> do
            return ()
  readIORef res

  where emptyCaseLoopCondition ret timeBound = do
          timeoutNotDone <- systemNanoTime >>= return.((>) timeBound)
          (readIORef ret) >>= return.((&&) timeoutNotDone)

        systemNanoTime = (getTime Monotonic) >>= return.toNanoSecs
```

Fig. 3.14: Código exchangeIO

```haskell
data EliminationArrayIO a = EAIO {exchanger :: [LockFreeExchangerIO a], duration ::
  ↪  Integer}

visitIO :: Eq a => EliminationArrayIO a -> Maybe a -> Int -> IO (Maybe a)
visitIO elimArr value range = do
  slot <- randomRIO (0, range)
  exchangeIO ((exchanger elimArr) !! slot) value (duration elimArr)
```

Fig. 3.15: Código para `EliminationArrayIO`

```haskell
data RangePolicy = RgPlcy {maxRange :: IORef Int, currentRange :: IORef Int}

newRangePolicy :: Int -> IO RangePolicy
newRangePolicy maxRange = do
  maxRg <- newIORef maxRange
  currRg <- newIORef 0
  return $ RgPlcy maxRg currRg

recordEliminationSuccess :: RangePolicy -> IO ()
recordEliminationSuccess rp = do
  max <- readIORef $ maxRange rp
  curr <- readIORef $ currentRange rp
  if curr < max
      then writeIORef (currentRange rp) (curr + 1)
      else return ()

recordEliminationTimeout :: RangePolicy -> IO ()
recordEliminationTimeout rp = do
  curr <- readIORef $ currentRange rp
  if curr > 0
      then writeIORef (currentRange rp) (curr - 1)
      else return ()

getRange :: RangePolicy -> IO Int
getRange = readIORef.currentRange
```

Fig. 3.16: Código de un `RangePolicy`

```haskell
pushEBSIO :: Eq a => EliminationBackoffStackIO a -> a -> IO ()
pushEBSIO ebs value = do
  ret <- newIORef True
  rangePolicy <- getTLS (policy ebs)

  range <- getRange rangePolicy
  node <- newNodeIO value
  whileM_ (readIORef ret) $ do
    b <- tryPushIO ebs node
    if b
      then writeIORef ret False
      else (catch (tryExchangePush ebs node value range ret rangePolicy) $ \( e ::
       ↪  TimeoutException) -> do
              recordEliminationTimeout rangePolicy)

  where tryExchangePush ebs node value range ret rangePolicy = do
          otherValue <- visitIO (eliminationArray ebs) (Just value) range
          if otherValue == Nothing
            then do
              recordEliminationSuccess rangePolicy
              writeIORef ret False
            else return ()
```

Fig. 3.17: Código de pushEBSIO

```haskell
popEBSIO :: Eq a => EliminationBackoffStackIO a -> IO a
popEBSIO ebs = do
  res <- newIORef Nothing
  ret <- newIORef True
  rangePolicy <- getTLS (policy ebs)

  range <- getRange rangePolicy
  whileM_ (readIORef ret) $ do
    returnNode <- tryPopIO ebs
    if returnNode /= Null
      then do
        writeIORef res $ Just (val returnNode)
        writeIORef ret False
      else (catch (exchangePop ebs range ret res rangePolicy) $ \( e :: TimeoutException)
       ↪  -> do
          recordEliminationTimeout rangePolicy)
  readIORef res >>= return.fromJust

  where exchangePop ebs range ret res rangePolicy = do
          otherValue <- visitIO (eliminationArray ebs) Nothing range
          case otherValue of
            Just v -> do
              recordEliminationSuccess rangePolicy
              writeIORef res (Just v)
              writeIORef ret False
            otherwise -> return ()
```

Fig. 3.18: Código de popEBSIO

```haskell
data StackSTM a = ST {top :: TVar (Node a)}

newStackSTM :: IO (StackSTM a)
newStackSTM = atomically (newTVar Null) >>= return.ST

pushStackSTM:: StackSTM a -> a -> IO ()
pushStackSTM st value = do
  node <- newNode value
  atomically $ do
    oldTop <- readTVar $ top st
    writeTVar (next node) oldTop
    writeTVar (top st) node

popStackSTM:: Show a => StackSTM a -> IO a
popStackSTM st = atomically $ do
    resNode <- readTVar $ top st
    case resNode of
      Nd v nxt -> do
        newTop <- readTVar nxt
        writeTVar (top st) newTop
        return v
      Null -> throw EmptyException
```

Fig. 3.19: Implementación completa de StackSTM

4. EXPERIMENTACIÓN

Los experimentos de este trabajo tienen como objetivo comparar el tiempo de ejecución de los algoritmos en sus distintas variantes de implementación.

Se realizaron varios experimentos preliminares para analizar cómo los parámetros de cada estructura utilizada puede alterar el tiempo de ejecución. Por ejemplo, se realizó un análisis del rendimiento de un LFS según sus distintos parámetros (tiempos límites mínimos y máximos de backoff) para luego usar la configuración óptima al momento de comparar el rendimiento de la estructura con las demás. Lo mismo ocurrió para la estructura EBS y sus parámetros de capacidad del arreglo y duración del timeout. Estos experimentos se describen en el apéndice A.

Finalmente, se realizaron tres experimentos comparando las implementaciones, alterando distintas variables como la cantidad de hilos de ejecución, los núcleos de procesamiento, y la proporción de hilos según la operación a realizar (apilar o desapilar).

4.1. Detalle de los experimentos

Para la experimentación se implementó un programa Haskell para ejecutar un escenario en el que varios hilos de ejecución realizan operaciones sobre una pila compartida y se mide el tiempo que transcurre entre el momento de creación de los hilos y el momento en el que todos los hilos terminan de realizar operaciones sobre la pila.

En este programa, cada hilo realizará una cantidad fija de operaciones: los hilos denominados "lectores" invocarán llamadas a la función `pop` mientras que los hilos "escritores" apilarán un valor aleatorio a la pila utilizando la función `push` de la implementación. Una vez que todos los hilos terminan, concluye la ejecución.

Cada programa fue ejecutado un número preciso de veces para luego realizar un promedio y poder obtener una apreciación de la consistencia de los resultados con diagramas *boxplot*.

Los experimentos realizados fueron tres. Uno estudia cómo varía el tiempo de ejecución respecto de la variación de la proporción de hilos escritores, manteniendo constante la cantidad de hilos totales a lo largo del experimento. Los otros dos estudian cómo la variación de la cantidad de hilos totales afecta el tiempo de ejecución pero con una distinción, en un experimento la cantidad total de operaciones es distribuída entre los hilos mientras que en el otro se define una cantidad de operaciones que cada hilo debe realizar. Esto implica que en el segundo experimento, cada hilo que se agrega aumenta la cantidad de operaciones totales. En ambos, la proporción de hilos escritores es constante durante todo el experimento. A continuación se listan los nombres utilizados para identificar a los experimentos:

- `pushPercentages`: proporción de hilos escritores vs. tiempo de ejecución.
- `numberOfThreads`: cantidad de hilos vs. tiempo de ejecución. En este caso la cantidad de operaciones totales aumenta ya que cada hilo debe realizar una cantidad fija de operaciones.

- `numberOfThreadsDist`: mismo análisis que `numberOfThreads` con la distinción que la cantidad de total de operaciones realizada se mantiene constante para cada cantidad de hilos. Es decir, a medida que aumentan los hilos, cada hilo de ejecución tiene menos operaciones a realizar ya que el total es distruibuido entre más hilos.

4.1.1. Hipótesis

La hipótesis manejada fue que la implementación STM `StackSTM` mejoraría el tiempo de respuesta a varias, si no todas, las demás implementaciones ya que ha habido mucho trabajo sobre el compilador GHC para poder mejorar la performance de STM.

Esta hipótesis se alinea con resultados similares en [7], donde se comparan implementaciones de una cola bloqueante, una utilizando STM y la otra utilizando algoritmos que utilizan locks sobre la estructura de datos. En [7], los resultados muestran que la implementación STM supera a la otra a medida que aumenta la cantidad de núcleos a utilizar por el procesador.

También estuvieron en consideración los resultados de [13] donde se comparan implementaciones similares de una lista simplemente encadenada. En estos resultados, la implementación sobre STM no obtiene buenos tiempos de ejecución comparada al resto. Sin embargo, esto se debe a que una lista encadenada se debe recorrer utilizando muchas llamadas a la función `atomically`. En el caso de la implementación de `StackSTM` utilizada en nuestra experimentación, cada operación realiza sólo una llamada a `atomically`, ya que las operaciones trabajan sólamente con el tope de la pila, y es por eso que es de esperar que los resultados en nuestro caso no se correspondan con los obtenidos en [13].

4.1.2. Instrumentación para realizar los experimentos

Las implementaciones comparadas en los experimentos son `LockFreeStackIO`, `LockFreeStackSTM`, `EliminationBackoffStackIO`, `EliminationBackoffStackSTM`, y `StackSTM`. Para cada una de ellas, tendremos un programa Haskell que tomará los siguientes parámetros por línea de comando:

- `min` y `max`: Los parámetros para la estructura `LockFreeStack`. Estos consisten de los límites mínimos y máximos de la estructura de backoff que se encuentra detallada en la subsección 3.1.1.

- `count` y `duration`: Los parámetros para la estructura `EliminationBackoffStack`. Estos son la capacidad del `EliminationArray` y la duración del timeout a pasar por parámetro en las llamdas a la función `exchange` de cada `LockFreeExchanger`.

- `threadCount`: Cantidad de hilos de ejecución que serán creados para operar sobre una pila.

- `distributeOperations`: un valor booleano que determina si el parámetro `operationCount` se interpreta como la cantidad total de operaciones a ser distribuida entre los hilos o la cantidad que cada hilo debe realizar por separado.

- `operationCount`: Cantidad de operaciones a realizar en el experimento. Estas pueden ser cantidad de operaciones por hilo, o cantidad de operaciones totales que luego serán distribuidas entre la cantidad de hilos según el experimento.

- `pushPercentage`: La proporción de hilos que realizarán operaciones de escritura (`push`). Se tomará la cantidad de hilos de ejecución y se calculará la cantidad de hilos que realizarán operaciones de escritura sobre la pila según la proporción, el resto realizará operaciones de lectura (`pop`).

El programa se encarga de crear una instancia de pila según la implementación y sus parámetros, luego inserta 100000 elementos para evitar que ocurran excepciones por pila vacía durante la ejecución. Una vez instanciada la pila, se calcula la cantidad de hilos escritores a crear según la proporción dada por `pushPercentage` y la cantidad total dada por `threadCount`. Resta el cálculo de la cantidad de operaciones que debe realizar cada hilo tomando en cuenta los parámetros `operationCount` y `distributeOperations`.

Finalmente se inicia un reloj al crear los hilos escritores y lectores para que realicen sus operaciones, y al finalizar todos los hilos, se imprime por pantalla la cantidad de tiempo transcurrido en segundos. Este es el comportamiento de la función `timeExperiment` que aparece en la penúltima línea de la Fig. 4.1.

En la Fig. 4.1 se presenta el código Haskell del programa para la implementación `EliminationBackoffStackIO`.

Cada uno de los tres experimentos tienen un script bash para ejecutar los programas Haskell variando los parámetros de acorde al experimento, la implementación de pila deseada, y también la cantidad de núcleos del procesador a utilizar. Una vez determinada la combinación implementación/cantidad de núcleos, se ejecuta el programa Haskell de manera que la variación de parámetros coincida con lo que se quiere observar del experimento. Es decir, si el experimento en cuestión busca estudiar cómo la proporción de hilos escritores varía el tiempo de ejecución, el script bash se encargará de ejecutar, para cada combinación de implementación-cantidad de núcleos, el programa Haskell de esa implementación variando el valor del parámetro `pushPercentages` y manteniendo el resto constantes.

Para cada valor que puede tomar la variable `pushPercentages` en este ejemplo, se realizan varias corridas del mismo programa Haskell y se escribirán los resultados en un archivo CSV para cada combinación implementación-cantidad de núcleos. Luego los scripts llaman a un programa Python que creará los gráficos utilizando los datos de los archivos CSV.

4.1.3. Medición de tiempos

Para medir el tiempo que toma el programa Haskell en ejecutar todos los hilos de ejecución se utilizó la librería `clock` [14] de Haskell que provee al programador con distintos tipos de reloj. Se decidió optar por el tipo de reloj `Monotonic` que no depende del reloj del sistema y por lo tanto no puede ser alterado como el tipo de reloj `Realtime`.

La función `timeExperiment` utiliza la función `getTime` de la librería `clock` que retorna un valor de tipo `TimeSpec` el cual luego es convertido a un entero que representa la cantidad de nanosegundos que han transcurrido desde un punto fijo en el pasado como, por ejemplo, el momento de inicio del sistema. Se realizan dos llamadas a `getTime`: una antes de la ejecución de la acción a medir e inmediatamente después de su finalización. Finalmente la diferencia es impresa por pantalla en segundos como muestra la Fig. 4.2.

4.1.4. Detalles de compilación

Se decidió deshabilitar la funcionalidad de *parallel garbage collection* como se menciona en [13]. Según [13], el *garbage collector* de GHC presentó dificultades de desempeño cuando se trataba de estructuras que utilizaban el tipo `TVar`. Dado que esto se debe a un problema del compilador, y no de la librería STM, se deshabilitó el recolector de basura para no perjudicar los resultados de las implementaciones STM. Comentarios similares a este se encuentran presentes en [7].

Para deshabilitar esta funcionalidad, basta con utilizar la opción `-I` de GHC que establece cada cuantos segundos debería correr el recolector. Si se le pasa 0 como parámetro a la opción, queda deshabilitada la recollección de basura en tiempo idle [1].

4.2. Resultados

Los experimentos fueron realizados utilizando una laptop MacBook Pro del año 2016 con las siguientes especificaciones:

- **Procesador**: 2.0GHz dual-core Intel Core i5, Turbo Boost up to 3.1GHz, with 4MB shared L3 cache
- **Memoria**: 16 GB 1867 MHz LPDDR3
- **Sistema operativo**: macOS Mojave, versión 10.14

Dado que el equipo en cuestión tiene un procesador con cuatro núcleos, nuestros experimentos fueron limitados a ser realizados utilizando uno, dos, y cuatro núcleos.

4.2.1. Primer experimento: `pushPercentages`

El experimento `pushPercentages` consiste en tomar los tiempos de ejecución de cada implementación de pila concurrente, variando la proporción de hilos escritores en cada iteración. A continuación se listan los parámetros que se mantuvieron constantes durante el experimento.

- Cantidad de hilos totales: 20
- Cantidad de operaciones realizadas por hilo de ejecución: 10000
- Límite mínimo de backoff para las implementaciones de LFS: 100
- Límite máximo de backoff para las implementaciones de LFS: 1000
- Capacidad del `EliminationArray` para las implementaciones de EBS: 1
- Duración del timeout para las implementaciones de EBS: 100
- Cantidad de repeticiones de cada corrida: 10

Los resultados, presentados en la Fig. 4.3 muestran que las implementaciones de LFS, tanto IO como STM, no parecen variar considerablemente sus tiempos de ejecución según la variación en la proporción de hilos escritores/hilos lectores. Las pequeñas variaciones que se presentan para estas dos implementaciones pueden ser causa de distintos tiempos

de *backoff* determinado de manera aleatoria. Se puede observar este comportamiento en el gráfico de la subfigura 4.5c y 4.5d.

En cuanto a la implementación de pila sobre STM, `StackSTM`, vemos que los tiempos de ejecución aumentan a medida que se aumenta la cantidad de hilos escritores. Una causa posible podría radicar en el hecho que las operaciones de apilar un elemento son más costosas que las de desapilar dado que requieren declarar un nuevo nodo para apilar, lo cual podría resultar en mayores tiempos de ejecución por operación. La implementación de EBS sobre STM también presenta una tendencia similar, aunque toma tiempos de ejecución mayores a los de `StackSTM`.

Por otra parte, la implementación de EBS sobre IO tiene resultados menos consistentes que la versión STM. En las corridas realizadas utilizando 4 núcleos, representados en la Fig. 4.3c, los tiempos de ejecución son mayores al inicio del experimento, es decir, cuando la proporción de hilos escritores es del 10 %.

Sobre las implementaciones de LFS, los resultados no presentan una tendencia tan marcada como las demás. A medida que la variación en la proporción de hilos escritores varía, los tiempos de ejecución en las implementaciones LFS se mantienen constantes o incrementan levemente como muestran los gráficos de la Fig 4.3. También se puede apreciar que una vez que el programa es ejecutado sobre más de un núcleo del procesador, los resultados para LFS no sufren una variación considerable, mientras que el resto aumenta a medida que aumenta la cantidad de núcleos.

Se presenta un gráfico con los resultados para EBS sobre IO con diagramas *boxplot* para analizar la consistencia de la implementación en la Fig. 4.4. En el gráfico se puede ver que los resultados para una proporción de hilos escritores de 10 % presentan un rango muy amplio y es por eso que se obtiene un valor promedio mayor en esta configuración.

La consistencia de cada implementación para producir los mismos resultados se puede apreciar con los diagramas *boxplot* que se encuentran en la figura 4.5. Estos gráficos muestran los resultados para los experimentos ejecutados utilizando dos núcleos del procesador. Los resultados para uno y cuatro núcleos se encuentran disponibles en la sección B.1 del apéndice B.

4.2.2. Segundo experimento: `numberOfThreads`

Para este experimento se analizó el comportamiento del tiempo de ejecución a medida que se aumentaron la cantidad de hilos de ejecución y la cantidad de operaciones totales. Los parámetros tomaron los siguientes valores:

- Proporción de hilos escritores: 0.75
- Cantidad de operaciones realizadas por hilo de ejecución: 10000
- Límite mínimo de backoff para las implementaciones de LFS: 100
- Límite máximo de backoff para las implementaciones de LFS: 1000
- Capacidad del `EliminationArray` para las implementaciones de EBS: 1
- Duración del timeout para las implementaciones de EBS: 100
- Cantidad de repeticiones de cada corrida: 10

En este experimento, a medida que incrementa la cantidad de hilos, la cantidad de operaciones totales realizadas sobre la pila incrementa ya que cada hilo nuevo realiza unas 10000 operaciones adicionales sobre la pila.

Se pueden observar resultados similares en los tres gráficos. Una diferencia es que cuando los experimentos son ejecutados utilizando cuatro núcleos del procesador, la implementación `StackSTM` no resulta ser la óptima y al aumentar la cantidad de hilos de ejecución el tiempo de ejecución es mayor que las implementaciones de LFS como muestra la subfigura 4.6c. En la misma subfigura, vemos que la implementación de EBS sobre IO supera a la implementación sobre STM mientras que en el caso de 1 y 2 núcleos, ambas implementaciones no presentan diferencias considerables en tiempo de ejecución.

Para realizar un análisis más profundo de los experimentos, se analizaron los resultados de las corridas y se generaron gráficos con diagramas *boxplot* para apreciar la consistencia de cada implementación en cuanto a los tiempos de ejecución que logra a través de las corridas repetidas. Estos gráficos se encuentran en la Fig. 4.7.

En la figura 4.7 se muestran los resultados para cada implementación utilizando dos núcleos del procesador. En las subfiguras se presentan diagramas *boxplot* que representan el rango de valores que resultaron de correr el programa Haskell en múltiples iteraciones. La cantidad de veces que la corrida fue repetida está determinada por el parámetro `iterations`. Observando los gráficos, se puede ver que los resultados para la implementación `StackSTM` tienen diagramas *boxplot* de menor rango, lo cual implica una mayor consistencia para producir los mismos resultados al repetir el experimento.

Notar que las implementaciones de LFS, tanto IO como STM, presentan los boxplots donde los datos tienen una mayor variación y por lo tanto los tiempos de ejecución son menos consistentes. Sin embargo, se puede observar que a medida que aumentan los núcleos, los tiempos de ejecución de las implementaciones LFS no aumentan considerablemente como las demás implementaciones. Por ejemplo, los tiempos de LFS para dos y cuatro núcleos presentan resultados muy similares, mientras que el resto aumentan sus tiempos de ejecución en el salto de dos a cuatro núcleos.

En el caso de las implementaciones de EBS, observamos también diagramas *boxplot* consistentes, aunque se pueden distinguir anomalías en los resultados, como por ejemplo, en la Subfig. 4.7a dónde se encuentra un resultado por encima de los 8 segundos.

Los resultados para uno y cuatro núcleos se encuentran disponibles en la sección **??** del apéndice B.

4.2.3. Tercer experimento: `numberOfThreadsDist`

Para este experimento se analizó el comportamiento del tiempo de ejecución a medida que se aumentaron la cantidad de hilos de ejecución mantienendo constante la cantidad total de operaciones realizadas sobre la pila. Los parámetros tomaron los siguientes valores:

- Proporción de hilos escritores: 0.75
- Cantidad de operaciones totales: 1000000
- Límite mínimo de backoff para las implementaciones de LFS: 100
- Límite máximo de backoff para las implementaciones de LFS: 1000
- Capacidad del `EliminationArray` para las implementaciones de EBS: 1

- Duración del timeout para las implementaciones de EBS: 100
- Cantidad de repeticiones de cada corrida: 10

A diferencia del experimento en la subsección 4.2.2, en este experimento se mantiene constante la cantidad de operaciones totales. Es decir, a medida que incrementa la cantidad de hilos de ejecución, la cantidad de operaciones se distribuye equitativamente entre los distintos hilos.

Los resultados presentes en la Fig. 4.8 muestran tendencias similares a los experimentos anteriores. Se mantiene que la implementación con mejores tiempos de ejecución es la de `StackSTM` para uno y dos núcleos, mientras que en el caso de cuatro núcleos es superada por las implementaciones de LFS. En este último caso, la diferencia de tiempos entre las implementaciones LFS y la de `StackSTM` es más notoria que en los experimentos anteriores.

Las tendencias parecen indicar que a medida que aumenta la cantidad de hilos el tiempo de ejecución se mantiene dentro de un mismo rango de valores a partir de los 10 hilos de ejecución. Esto se puede ver en las subfiguras de la Fig. 4.8 dónde las líneas no tienen cambios considerables en la tendencia a partir de 10 hilos. Sin embargo, se observa un comportamiento errático para los tiempos de ejecución de la implementación de EBS sobre IO cuando se corre el programa sobre un núcleo del procesador. Para observar este caso en más detalle, se presenta la siguiente Fig. 4.9 donde se muestra los diagramas *boxplot* para los resultados de la implementación de EBS sobre IO con un núcleo del procesador.

Una vez más, se puede apreciar que la implementación sobre IO de EBS ha presentado irregularidades en sus resultados en la forma de *outliers* o el rango del boxplot en los resultados con 50 hilos de ejecución. Estas irregularidades son la causa de la tendencia presentada en la subfigura 4.8a. En la Fig. 4.10 se presentan el resto de los gráficos individuales de cada implementación con los diagramas *boxplot* para cada valor.

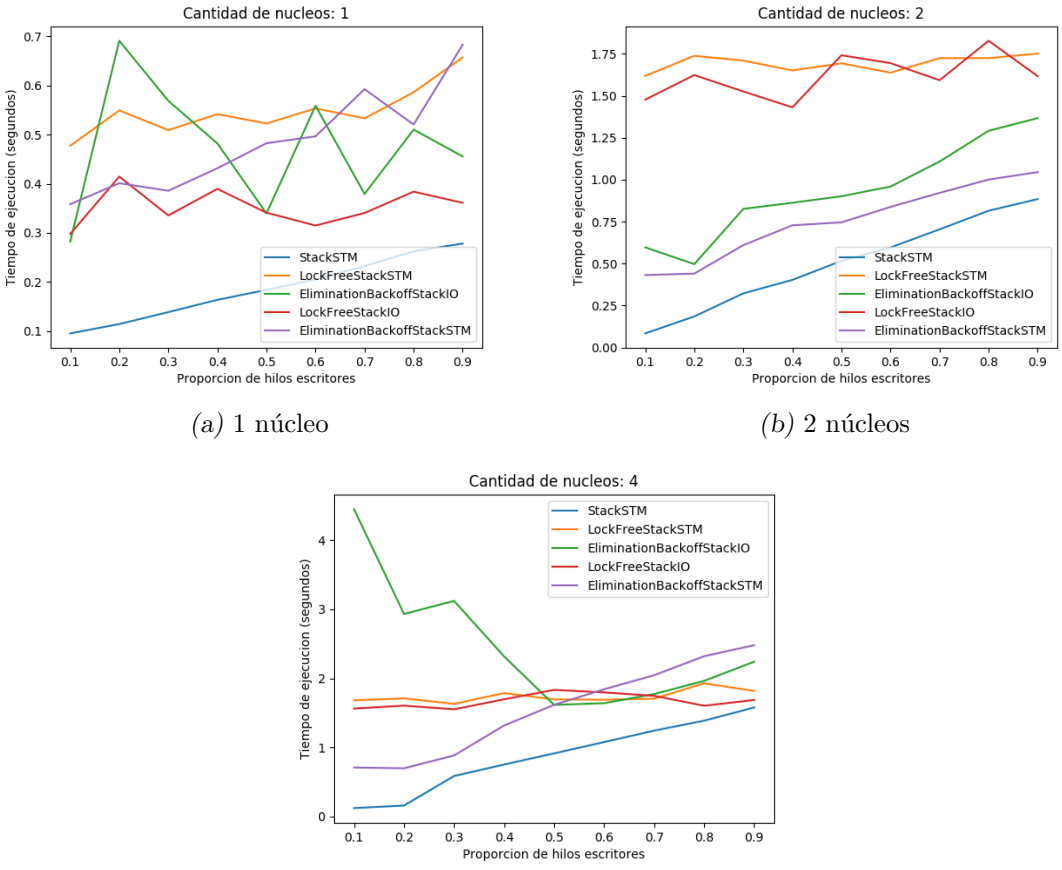
En la Fig. 4.10 se puede apreciar cómo la implementación de `StackSTM` sigue siendo la que produce resultados más consistentes a medida que se repiten las corridas. También se observa nuevamente la presencia de outliers en los resultados para la implementación de EBS sobre IO, como por ejemplo el punto que se puede observar en la subfigura 4.10a dónde una corrida con 60 hilos de ejecución tomó 16 segundos en completarse. Se pueden observar outliers en otros gráficos como las subfiguras 4.10d y 4.10b, pero estos no se encuentran muy alejados del rango de su respectivo *boxplot*.

```haskell
callPushes stack operationCount = do
    if operationCount > 0
        then do
            (randomIO :: IO Int) >>= pushEBSIO stack
            callPushes stack (operationCount - 1)
        else do
            return ()

callPops stack operationCount = do
    if operationCount > 0
        then do
            _ <- catch (popEBSIO stack) (\(e :: EmptyException) -> return 1)
            callPops stack (operationCount - 1)
        else do
            return ()

threadAction isPushThread = if isPushThread then callPushes else callPops

createThreads stack threadCount pushThreadCount operationCount tids = do
    let isPushThread = if pushThreadCount > 0 then True else False
    if threadCount > 0
        then do
            tid <- async (threadAction isPushThread stack operationCount)
            createThreads stack (threadCount - 1) (pushThreadCount - 1) operationCount
             ↪  (tid : tids)
        else
            mapM_ wait tids

parseCommandLineArguments args = do
    -- Get arguments from command line
    capacity <- readIO (args !! 0) :: IO Int
    duration <- readIO (args !! 1) :: IO Integer
    operationCount <- readIO (args !! 2) :: IO Int
    pushPercentage <- readIO (args !! 3) :: IO Float
    threadCount <- readIO (args !! 4) :: IO Int
    distributeOperations <- readIO (args !! 5) :: IO Bool
    -- Distribute operationCount
    operationCount <- return $ if distributeOperations then quot operationCount
     ↪  threadCount else operationCount
    -- Create stack and calculate amount of writer/push threads
    stack <- newEBSIO capacity duration
    let pushThreadCount = floor $ (fromIntegral threadCount :: Float) * pushPercentage
    return (stack, threadCount, operationCount, pushThreadCount)

main = do
    args <- getArgs
    (stack, threadCount, operationCount, pushThreadCount) <- parseCommandLineArguments
     ↪  args
    callPushes stack 1000000
    timeExperiment "" $ createThreads stack threadCount pushThreadCount operationCount []
```

Fig. 4.1: Código Haskell para la experimentación sobre la implementación `EliminationBackoffStackIO`

```haskell
timeExperiment action = do
  startTime <- (getTime Monotonic) >>= return.toNanoSecs
  action
  endTime <- (getTime Monotonic) >>= return.toNanoSecs
  let inSecs = (fromIntegral (endTime - startTime)) / (10 ** 9)
  print inSecs
```

Fig. 4.2: Código de la función `timeExperiment`



(a) 1 núcleo

(b) 2 núcleos

(c) 4 núcleos

Fig. 4.3: Resultados para el experimento `pushPercentages`

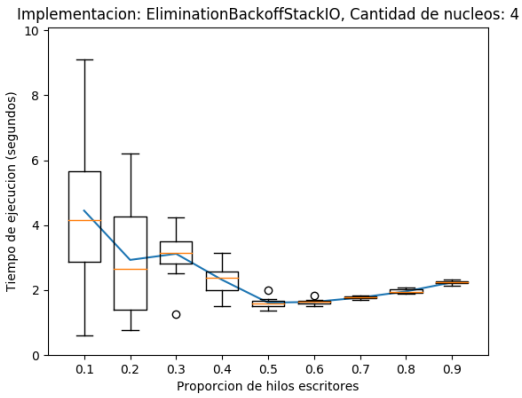
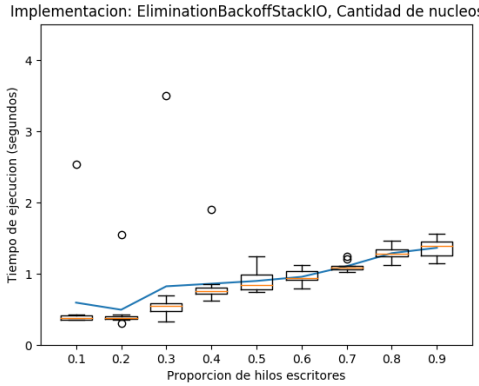
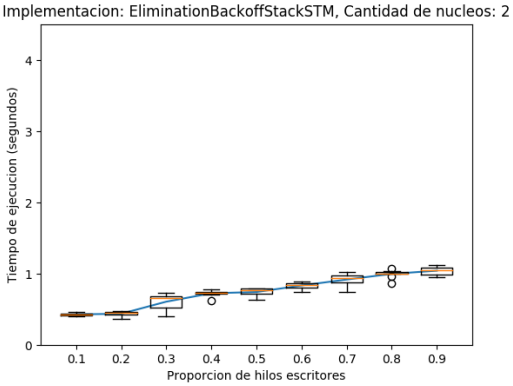


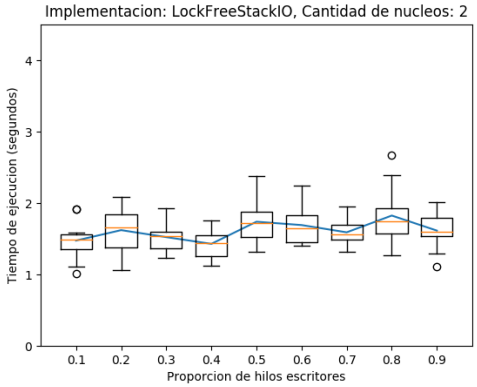
Fig. 4.4: Resultados de EBS sobre IO, con cuatro núcleos de ejecución



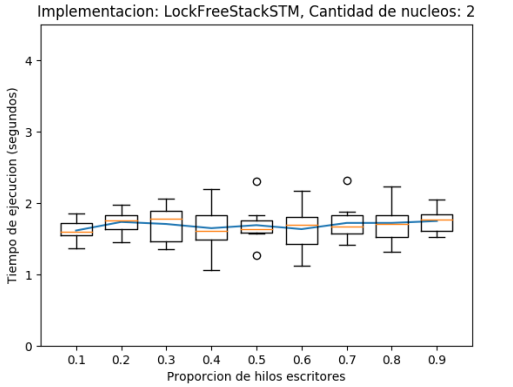
(a) Resultados para EBS sobre IO



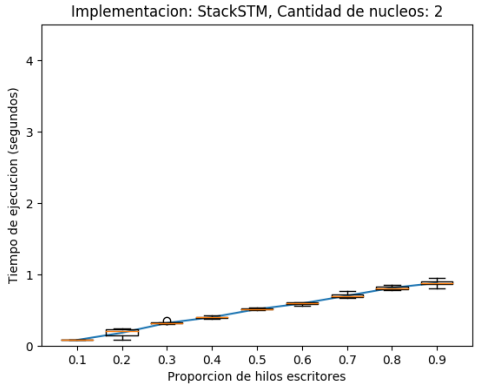
(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO

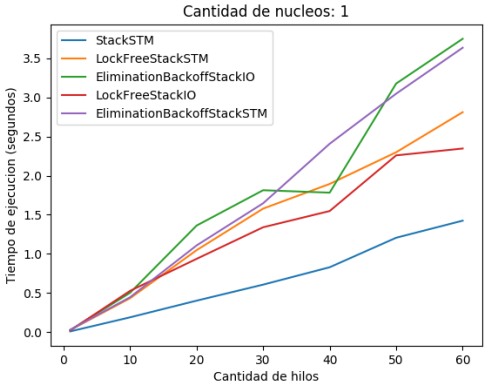


(d) Resultados para LFS sobre STM

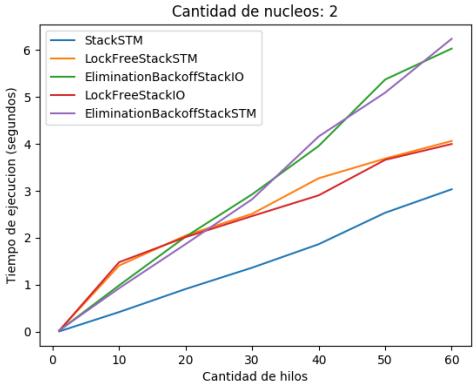


(e) Resultados para LFS sobre STM

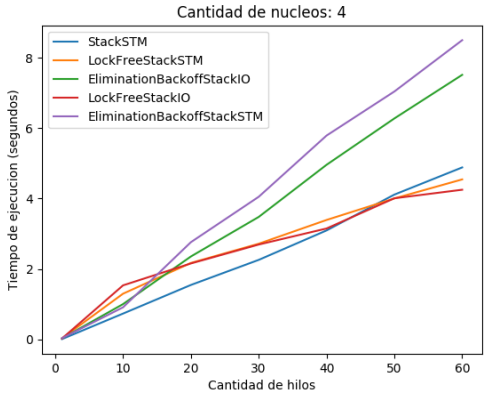
Fig. 4.5: Comparación de implementaciones IO vs STM del experimento `pushPercentages`



(a) 1 núcleo

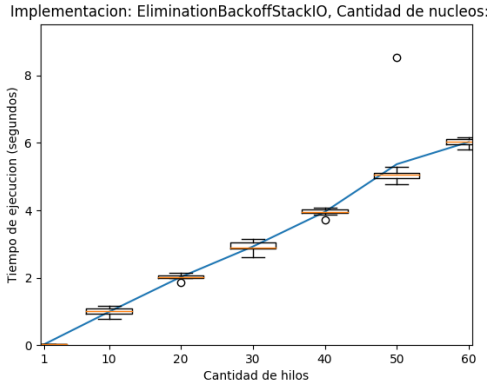


(b) 2 núcleos

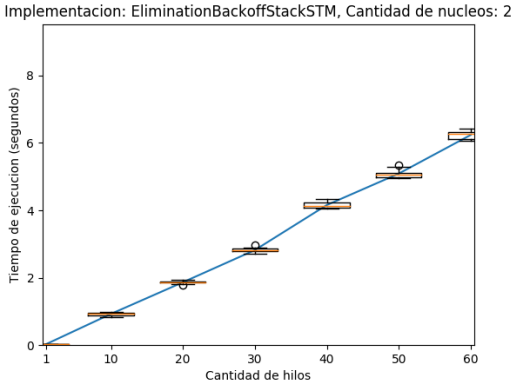


(c) 4 núcleos

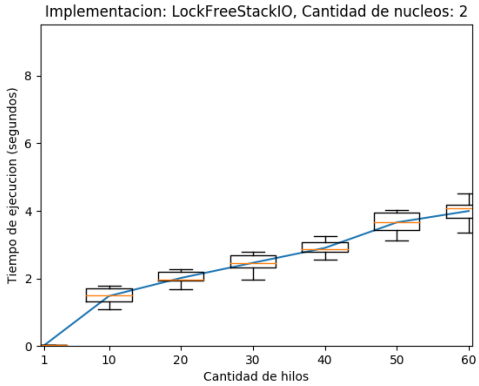
Fig. 4.6: Resultados para el experimento `numberOfThreads`



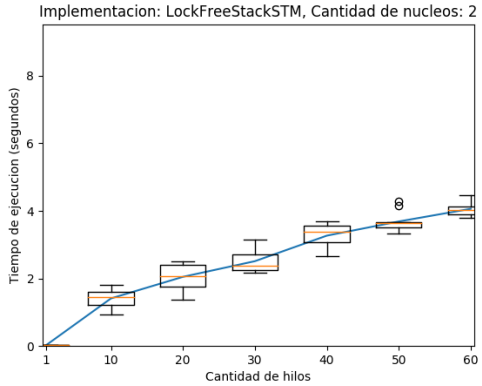
(a) Resultados para EBS sobre IO



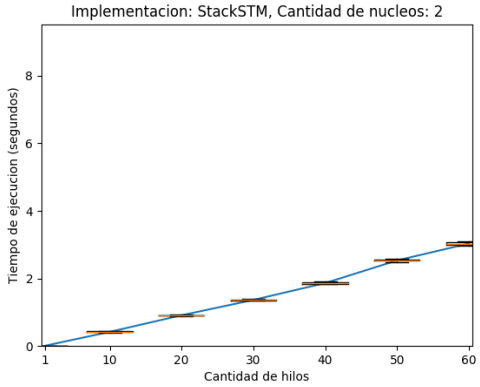
(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO

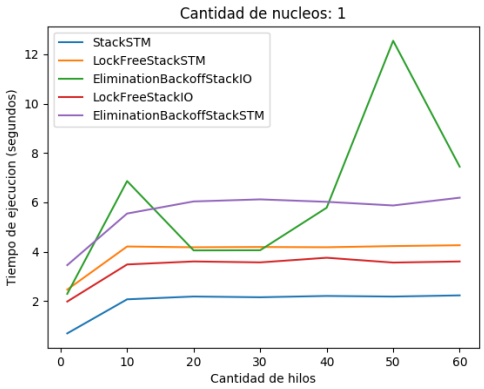


(d) Resultados para LFS sobre STM

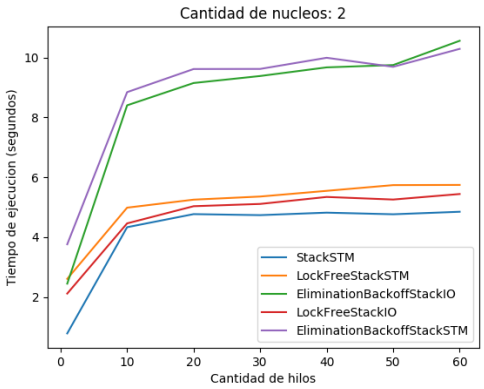


(e) Resultados para LFS sobre STM

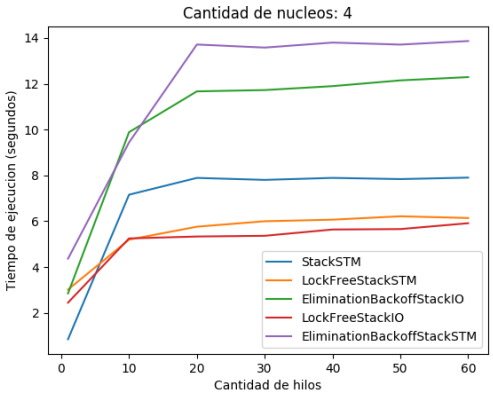
Fig. 4.7: Comparación de implementaciones IO vs STM del experimento `numberOfThreads`



(a) 1 núcleo



(b) 2 núcleos



(c) 4 núcleos

Fig. 4.8: Resultados para el experimento `numberOfThreadsDist`

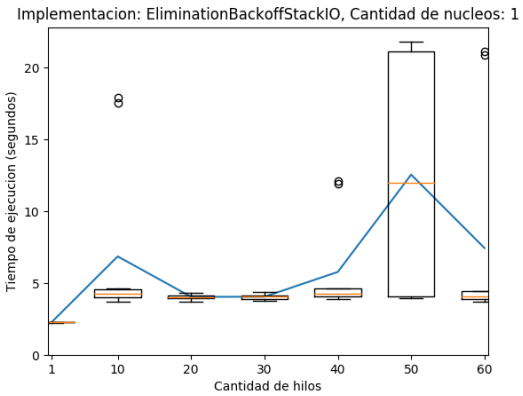
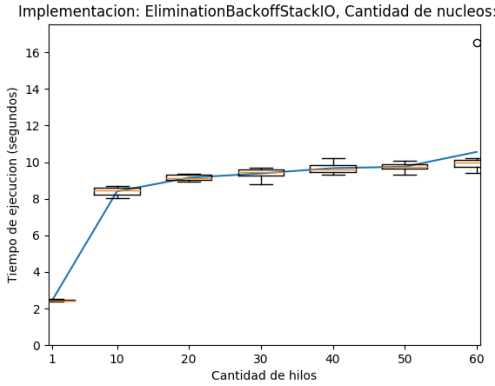
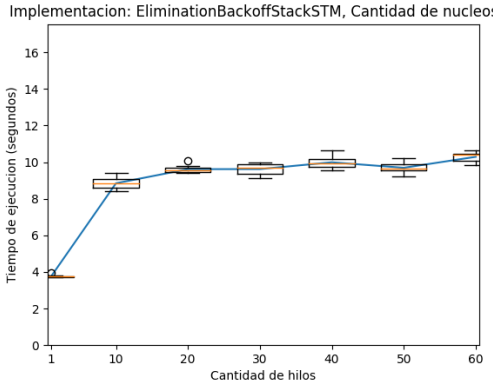


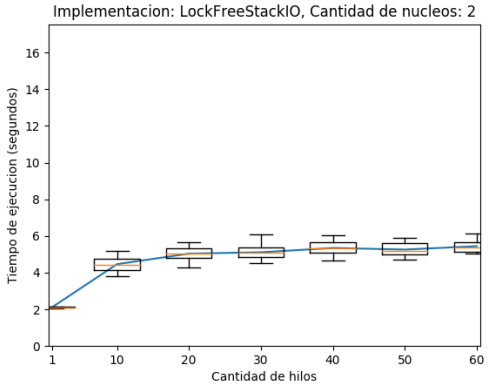
Fig. 4.9: Resultados del experimento `numberOfThreadsDist` para EBS sobre IO



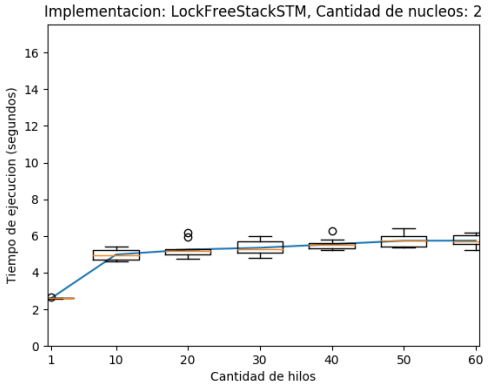
(a) Resultados para EBS sobre IO



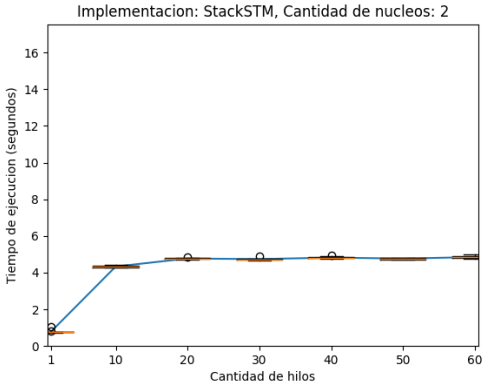
(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO



(d) Resultados para LFS sobre STM



(e) Resultados para LFS sobre STM

Fig. 4.10: Comparación de implementaciones IO vs STM del experimento `numberOfThreadsDist`

5. CONCLUSIÓN

Entre las cinco versiones implementadas, uno puede concluir que las implementaciones de EBS no son las más adecuadas en términos de performance para ser utilizados, especialmente la versión IO que presentó varias ocurrencias de anomalías en sus resultados. Este resultado soprende dado que la estructura EBS fue propuesta como una alternativa de mayor escalabilidad que LFS cuando, en nuestros resultados para los experimentos que variaron la cantidad de hilos, se encontró que las implementaciones LFS mantuvieron tiempos de ejecución bajos en comparación a las implementaciones EBS. En cuanto a la implementación STM, esta resulta ser la mejor opción cuando la cantidad de hilos es baja, acercándose a tiempos de ejecución similares a las implementaciones LFS a medida que se agregan hilos. Otra observación importante es que las implementaciones de LFS no son afectadas por el incremento en la cantidad de núcleos de ejecución como el resto de las implementaciones.

Los experimentos realizados dan lugar a analizar distintos casos de uso. Por ejemplo, si el caso de uso para una pila concurrente es un escenario donde la cantidad de hilos de ejecución es constante y lo que varía son el tipo de operaciones que realizan los hilos uno se encuentra en una situación parecida al experimento `pushPercentages` y puede optar por la implementación que mejores resultados presentó. En este caso, la implementación STM es una opción clara dado que obtuvo los menores tiempos de ejecución. Si el caso de uso para la pila concurrente requiere de mayor estabilidad en los tiempos de ejecución a con una alta variación en la cantidad de hilos que ejecutan sobre la pila, cualquiera de las implementaciones LFS son buenos candidatos a elegir para el escenario. La implementación sobre STM también es una buena opción, pero uno debería tener cuidado si la cantidad de hilos se vuelve muy grande ya que la tendencia de los resultados muestra que la implementación puede llegar a tiempos de ejecución mayores a los de LFS.

Es importante analizar también la complejidad de las estructuras al momento de implementarlas. Las implementaciones de EBS son sin duda las más complejas dadas las estructuras auxiliares de las que depende como el exchanger y el arreglo de eliminación. Por el otro lado, la implementación de STM es la más fácil de lograr una vez obtenido conocimiento sobre el funcionamiento de la librería STM.

BIBLIOGRAFÍA

[1] *4.17. Running a compiled program.* URL: https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/runtime-control.html.

[2] *AtomicStampedReference (Java Platform SE 7).* URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicStampedReference.html.

[3] *Compare-and-swap - Wikipedia.* URL: https://en.wikipedia.org/wiki/Compare-and-swap.

[4] *Control.Concurrent.* https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Concurrent.html.

[5] *Control.Monad.* URL: https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Monad.html#t:Monad.

[6] *Data.TLS.GHC.* URL: https://hackage.haskell.org/package/thread-local-storage-0.2/docs/Data-TLS-GHC.html.

[7] Anthony Discolo y col. "Lock Free Data Structures Using STM in Haskell". En: *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*. 2006, págs. 65-80. DOI: 10.1007/11737414_6. URL: https://doi.org/10.1007/11737414%5C_6.

[8] Rodrigo Medeiros Duarte y col. "Concurrent Hash Tables for Haskell". En: *Programming Languages*. Ed. por Fernando Castor y Yu David Liu. Cham: Springer International Publishing, 2016, págs. 110-124. ISBN: 978-3-319-45279-1.

[9] *Haskell/do notation - Wikibooks, open books for an open world.* URL: https://en.wikibooks.org/wiki/Haskell/do_notation.

[10] Danny Hendler, Nir Shavit y Lena Yerushalmi. "A Scalable Lock-free Stack Algorithm". En: *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '04. Barcelona, Spain: ACM, 2004, págs. 206-215. ISBN: 1-58113-840-7. DOI: 10.1145/1007912.1007944. URL: http://doi.acm.org/10.1145/1007912.1007944.

[11] Maurice Herlihy y Nir Shavit. *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123705916, 9780123705914.

[12] *Software transactional memory - Wikipedia.* URL: https://en.wikipedia.org/wiki/Software_transactional_memory.

[13] Martin Sulzmann, Edmund S.L. Lam y Simon Marlow. "Comparing the Performance of Concurrent Linked-list Implementations in Haskell". En: *SIGPLAN Not.* 44.5 (oct. de 2009), págs. 11-20. ISSN: 0362-1340. DOI: 10.1145/1629635.1629643. URL: http://doi.acm.org/10.1145/1629635.1629643.

[14] *System.Clock*. URL: https://hackage.haskell.org/package/clock-0.7.2/docs/System-Clock.html.

[15] *System.Random*. URL: https://hackage.haskell.org/package/random-1.1/docs/System-Random.html.

[16] R. Kent Treiber. *Systems Programming: Coping with Parallelism*. Inf. téc. RJ 5118. IBM Almaden Research Center, abr. de 1986.

Apéndice A

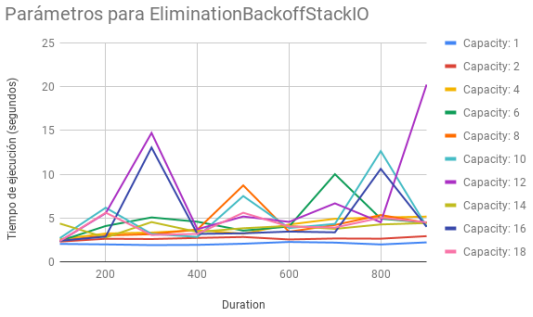
EXPERIMENTOS PARA DETERMINAR PARÁMETROS

Estos experimentos preliminares utilizan el mismo programa Haskell que la experimentación realizada en el capítulo 4 y varía los parámetros de las distintas implementaciones (límites de backoff de un LFS, y capacidad y duración de un EBS) para encontrar la configuración óptima para luego utilizarla en los experimentos detallados en el capítulo 4.

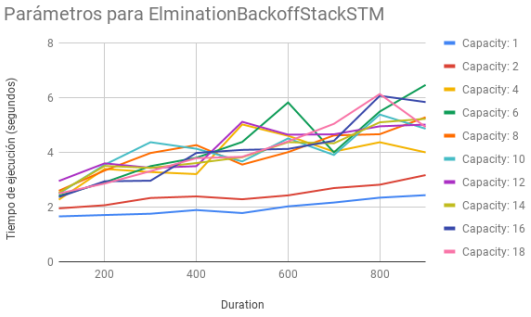
Las variables de control del experimento tomaron los siguientes valores:

- Proporción de hilos escritores: 0.75
- Cantidad de hilos de ejecución: 30
- Cantidad de operaciones realizadas por hilo: 10000
- Cantidad de núcleos del procesador utilizados: 2
- Cantidad de repeticiones de cada corrida: 5

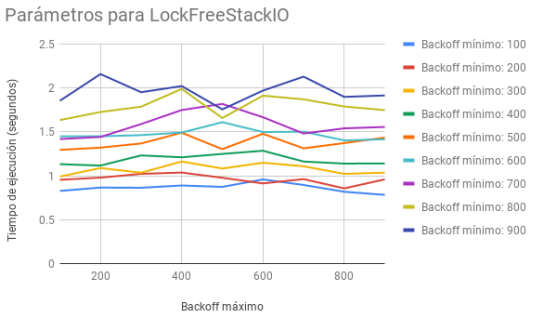
El objetivo del experimento es determinar qué combinación de parámetros era la más adecuada para cada estructura. Es decir, qué combinación de tiempos de backoff mínimo y máximo es óptima para las implementaciones de LFS y qué combinación de capacidad y duración de timeout es óptima para las implementaciones de EBS.



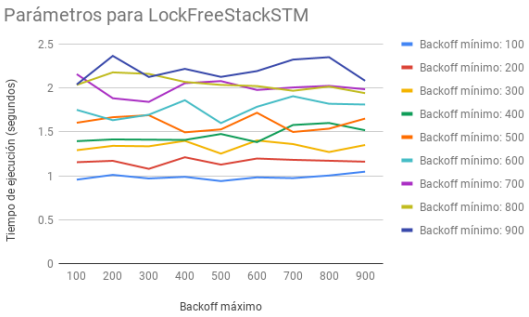
(a) Resultados para EBS sobre IO



(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO



(d) Resultados para LFS sobre STM

Fig. A.1: Resultados de los experimentos para determinar parámetros

De los resultados se observa que, en un LFS, los tiempos de ejecución son menores cuando el límite mínimo de backoff tiene un valor de 100 representados en la líneas de "Backoff mínimo: 100" en las subfiguras A.1d y A.1c. Para las implementaciones de EBS, los resultados muestran que el valor óptimo para el parámetro de capacidad es 1.

Según los resultados, no parece haber mucha variación de los tiempos de ejecución según límite máximo de backoff cuando el límite mínimo es de 100. Es por esto que al momento de experimentar se utilizó un valor de 1000 en el límite máximo de backoff para perimitir que el límite tenga un buen margen para crecer.

En cuanto a la duración del timeout para las estructuras EBS, a medida que el valor del timeout aumenta el tiempo de ejecución también lo hace. Se decidió entonces utilizar el un timeout con valor de 100 en la experimentación para reducir los tiempos de ejecución y para que también coincida con el límite mínimo de backoff del LFS.

Apéndice B

RESULTADOS DE LOS EXPERIMENTOS

B.1. Experimento `pushPercentages`

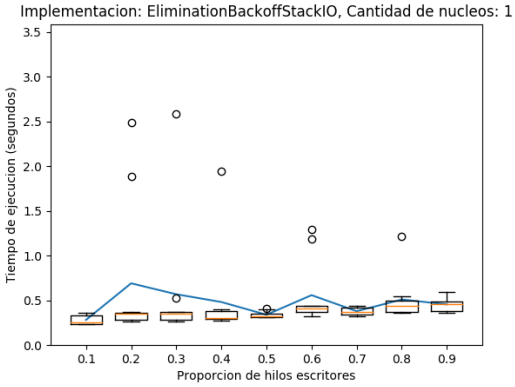
El experimento `pushPercentages` estudia el comportamiento del tiempo de ejecución a medida que varía la proporción de hilos escritores. En las figuras B.1 y B.2 se presentan los resultados de cada implementación con diagramas *boxplot* para las corridas realizadas en 1 y 4 núcleos respectivamente.

B.2. Experimento `numberOfThreads`

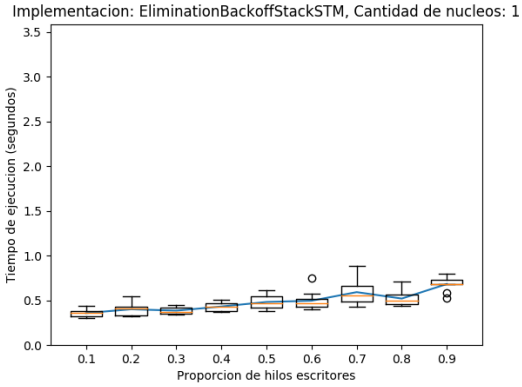
El experimento `numberOfThreads` estudia el comportamiento del tiempo de ejecución a medida que varía la cantidad total de hilos del experimento. En este experimento, la cantidad de operaciones totales aumenta a medida que aumentan los hilos ya que cada hilo que se agrega realiza una cantidad fija de operaciones. En las figuras B.3 y B.4 se presentan los resultados de cada implementación con diagramas *boxplot* para las corridas realizadas en 1 y 4 núcleos respectivamente.

B.3. Experimento `numberOfThreadsDist`

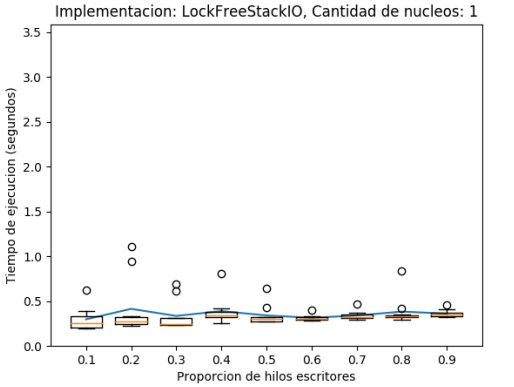
El experimento `numberOfThreadsDist` estudia el comportamiento del tiempo de ejecución a medida que varía la cantidad total de hilos del experimento. En este experimento, la cantidad de operaciones totales se mantiene constante y se distribuye entre los hilos de ejecución. En las figuras B.5 y B.6 se presentan los resultados de cada implementación con diagramas *boxplot* para las corridas realizadas en 1 y 4 núcleos respectivamente.



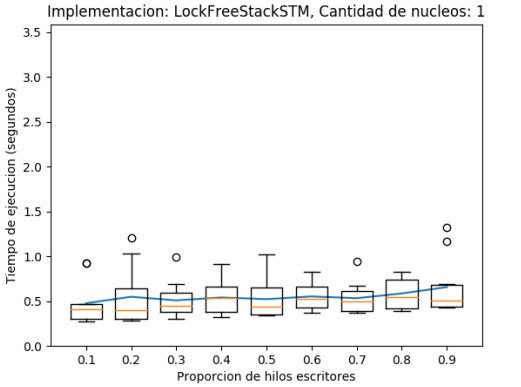
(a) Resultados para EBS sobre IO



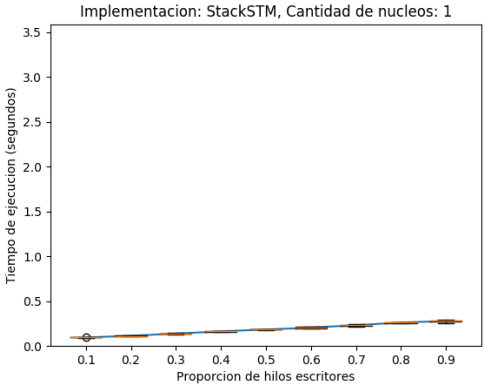
(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO

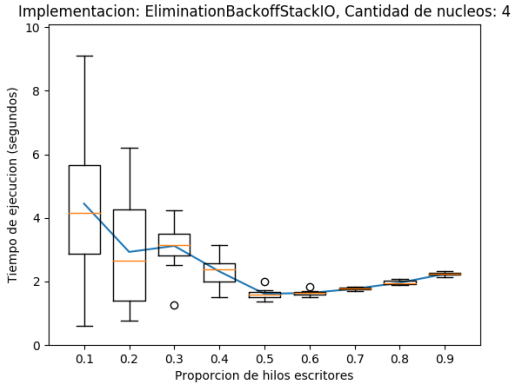


(d) Resultados para LFS sobre STM

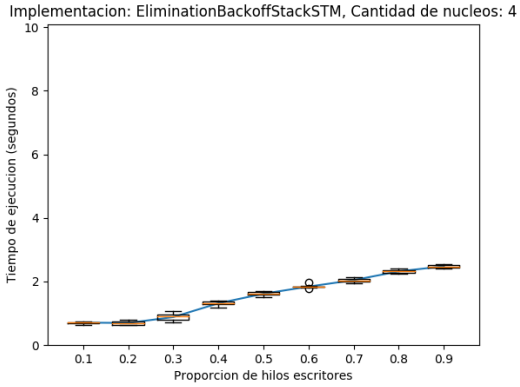


(e) Resultados para LFS sobre STM

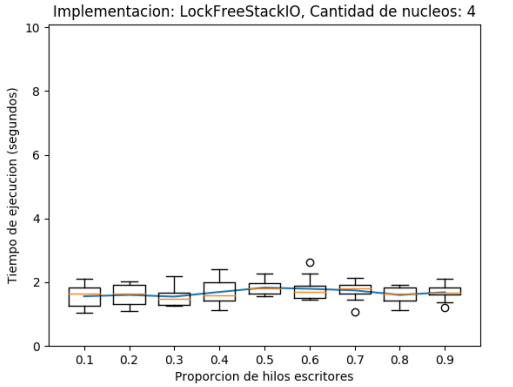
Fig. B.1: Comparación de implementaciones IO vs STM del experimento `pushPercentages` (1 núcleo de ejecución)



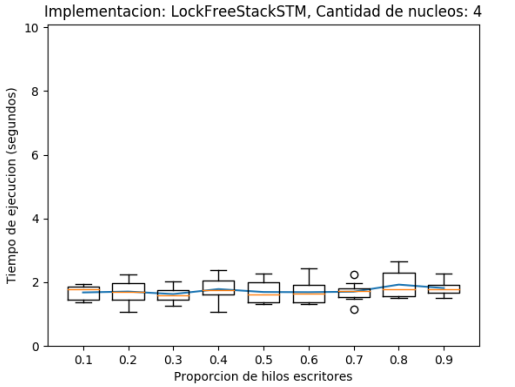
(a) Resultados para EBS sobre IO



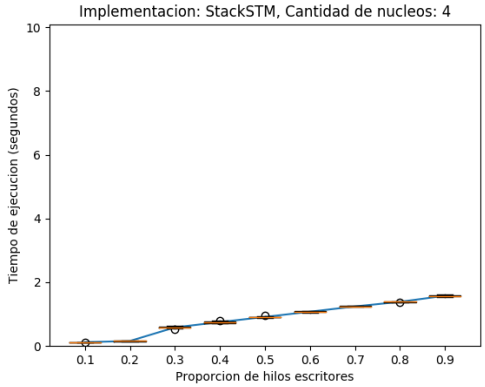
(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO

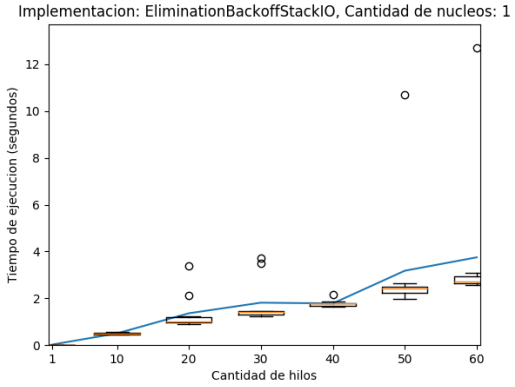


(d) Resultados para LFS sobre STM

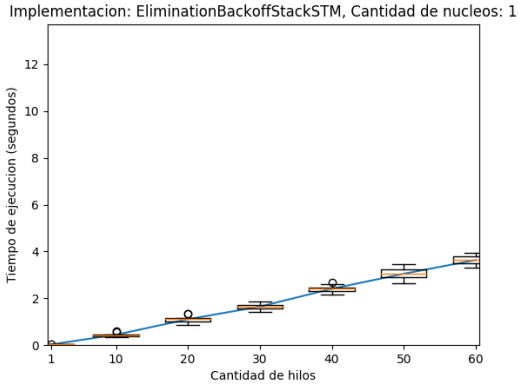


(e) Resultados para LFS sobre STM

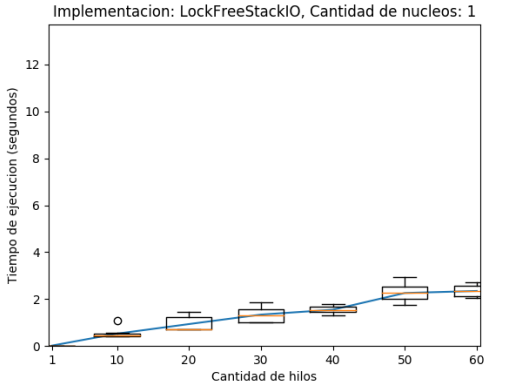
Fig. B.2: Comparación de implementaciones IO vs STM del experimento `pushPercentages` (4 núcleos de ejecución)



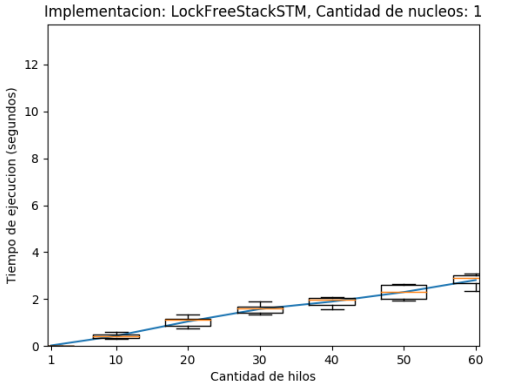
(a) Resultados para EBS sobre IO



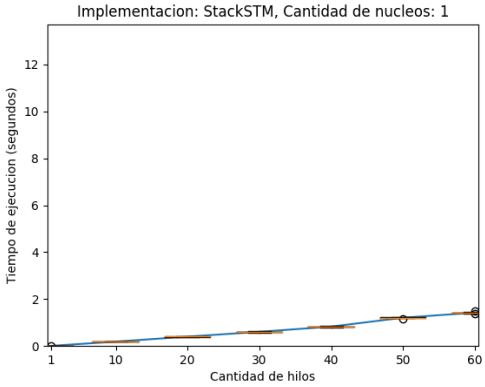
(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO

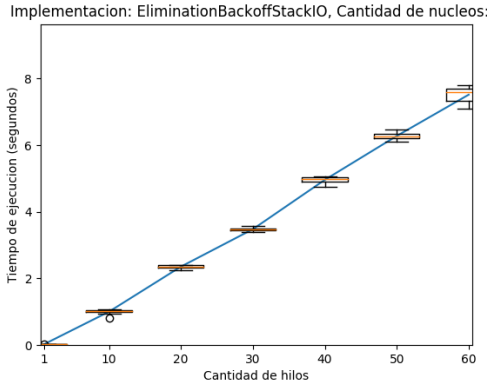


(d) Resultados para LFS sobre STM

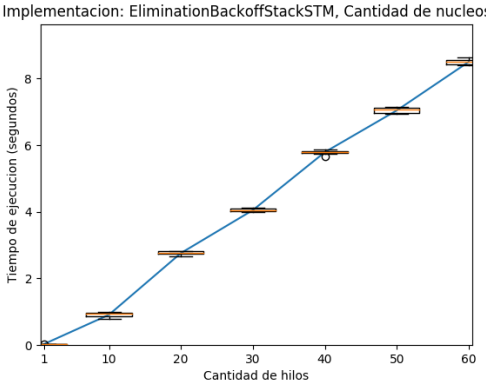


(e) Resultados para LFS sobre STM

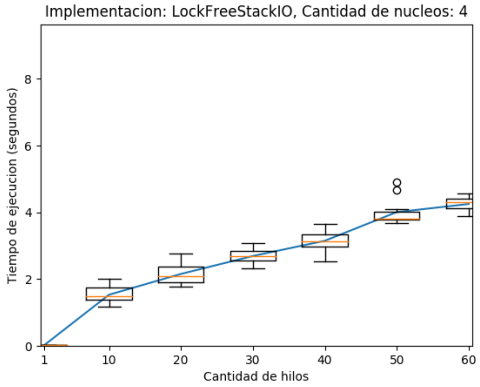
Fig. B.3: Comparación de implementaciones IO vs STM del experimento `numberOfThreads` (1 núcleo de ejecución)



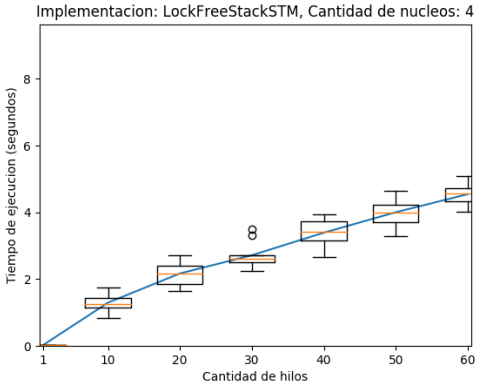
(a) Resultados para EBS sobre IO



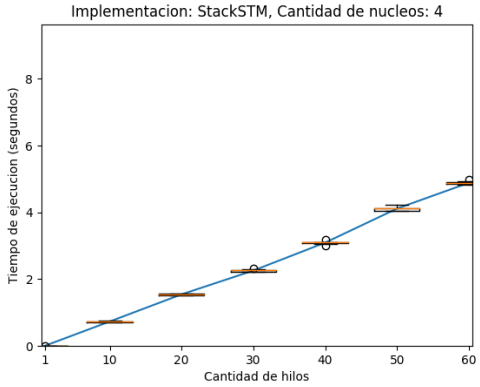
(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO

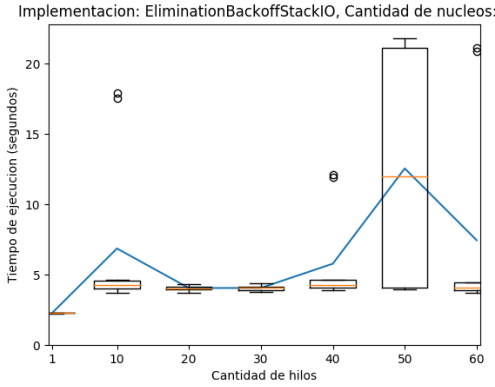


(d) Resultados para LFS sobre STM

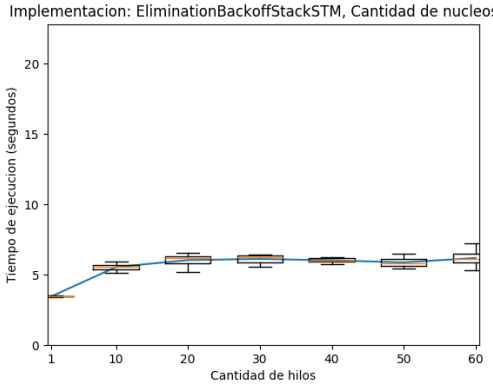


(e) Resultados para LFS sobre STM

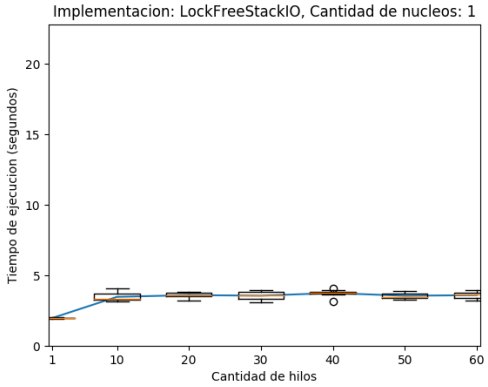
Fig. B.4: Comparación de implementaciones IO vs STM del experimento `numberOfThreads` (4 núcleos de ejecución)



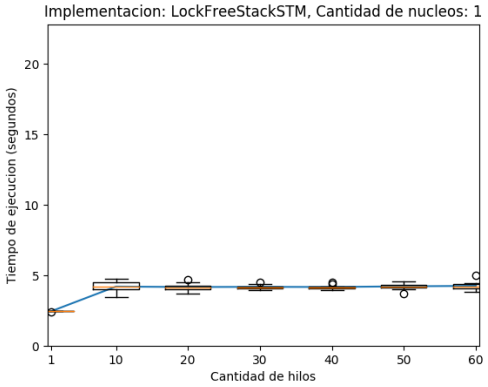
(a) Resultados para EBS sobre IO



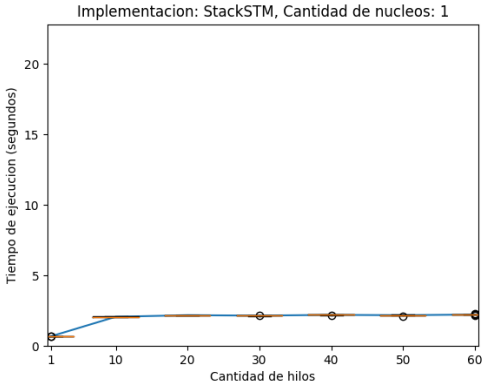
(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO

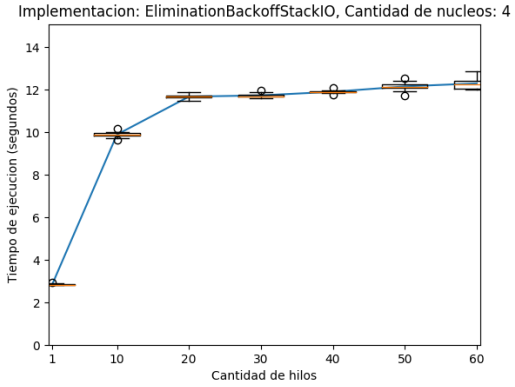


(d) Resultados para LFS sobre STM

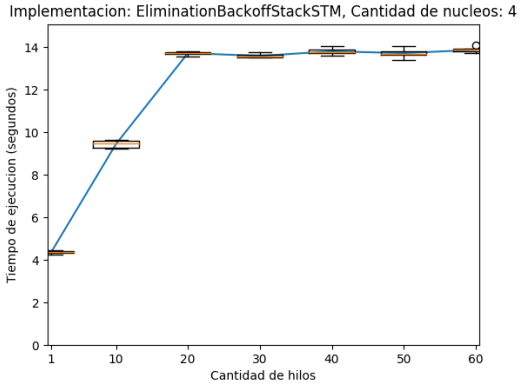


(e) Resultados para LFS sobre STM

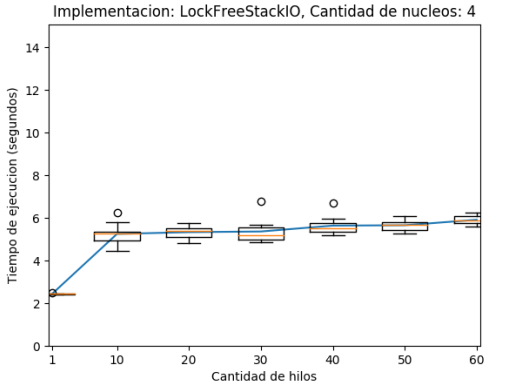
Fig. B.5: Comparación de implementaciones IO vs STM del experimento `numberOfThreadsDist` (1 núcleo de ejecución)



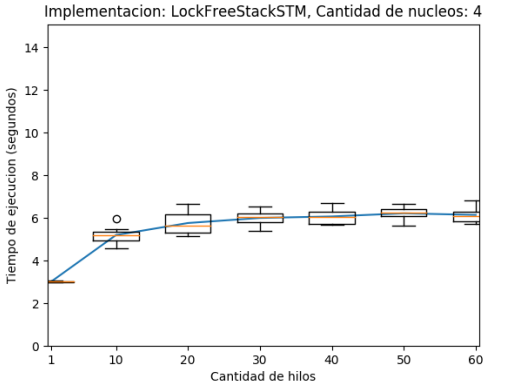
(a) Resultados para EBS sobre IO



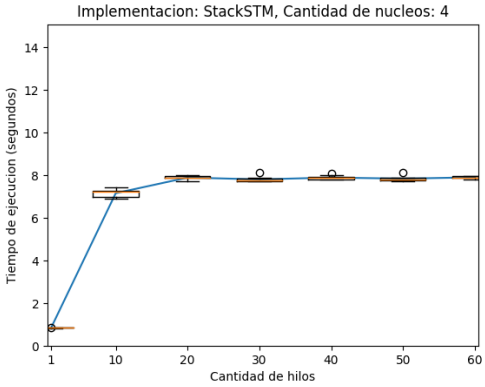
(b) Resultados para EBS sobre STM



(c) Resultados para LFS sobre IO



(d) Resultados para LFS sobre STM



(e) Resultados para LFS sobre STM

Fig. B.6: Comparación de implementaciones IO vs STM del experimento `numberOfThreadsDist` (4 núcleos de ejecución)