Recolección de basura en D

Leandro Matías Lucarella

Dirección Lic. Rosa Wachenchauzer

Padrón 77.891

Tesis de Grado en Ingeniería en Informática

Departamento de Computación

Facultad de Ingeniería

Universidad de Buenos Aires

Octubre 2010

Índice general

Índice de figuras

Índice de cuadros

Resumen

El manejo de memoria es un problema recurrente en los lenguajes de programación; dada su complejidad es muy propenso a errores y las consecuencias de dichos errores pueden ser muy graves. La *recolección de basura* es el área de investigación que trata las técnicas de manejo automático de memoria. D es un lenguaje de programación compilado, con tipado estático y multi-paradigma que combina el poder de lenguajes de programación de bajo nivel, como C, con la facilidad de los de alto nivel, como Python o Java. D provee recolección de basura pero ha sido frecuentemente criticada por sus varias falencias. Dadas las particularidades del lenguaje, plantea un desafío casi único en cuanto al diseño de un recolector.

Este trabajo hace un recorrido por el estado del arte en recolección de basura teniendo en cuenta los requerimientos de D; analiza la implementación del recolector actual y propone mejoras con el objetivo principal de minimizar los tiempos de pausa. Finalmente se construye un banco de pruebas para verificar los resultados, que muestran una disminución de hasta 200 veces en el tiempo de pausa del recolector y de hasta 3 veces en el tiempo total de ejecución.

Agradecimientos

A mis viejos y hermanos por soportarme y apoyarme, a Ali por estar siempre para darme un empujoncito cuando me estanco, a Rosita por introducirme en el tema, aguantar mis baches y ayudarme a mantener el foco, a Albertito por los aportes técnicos invaluables a cambio de helado, a la gente que hizo la innumerable cantidad de Software Libre que usé en este trabajo por simplificarme enormemente la vida, y a la gente que me estoy olvidando por hacer eso que me estoy olvidando que hizo.

¡Es basura! ¡Es basura! ¡Es basura!

Jay Sherman

CAPÍTULO 1

Introducción

La recolección de basura es una técnica que data de fines de los años ‘50, cuando John McCarthy, creador de Lisp, agregó a dicho lenguaje la capacidad de administrar la memoria automáticamente (utilizando conteo de referencias), entre muchos otros conceptos revolucionarios para la época que recién fueron explotados masivamente en lenguajes dinámicos más modernos como Perl, Python, Ruby, etc. A partir de este momento, muchos lenguajes tomaron esta característica y hubo una gran cantidad de investigación al respecto, tomando particular importancia.

Nuevos algoritmos fueron desarrollados para atacar distintos problemas particulares y para mejorar el rendimiento, que ha sido una inquietud incesante en la investigación de recolectores de basura. Sin embargo el lenguaje más masivo que ha adoptado un recolector de basura (al menos en el ámbito empresarial) fue Java, con el cual la investigación sobre recolección de basura tomó un impulso extra. Probablemente el mayor desarrollo e investigación en cuanto a recolección de basura se siga dando para Java, acotando tal vez un poco el alcance de estos avances dado que ese lenguaje tiene características muy particulares (*tipado* estático, corre sobre una máquina virtual muy rica en cuanto a información de tipos, etc.) no disponibles en otros lenguajes. Sin embargo los lenguajes funcionales y/o con *tipado* dinámico siguieron teniendo un nivel de investigación y desarrollo importante, dado que fueron concebidos en su mayoría con la recolección de basura como parte del diseño.

Probablemente los lenguajes en los cuales es más difícil aplicar los avances que se desprendieron de Java o de las otras categorías de lenguajes con más avances en recolección de basura sean los de más bajo nivel, como C y C++, en los cuales se da lo inverso en cuanto a disponibilidad de información en tiempo de ejecución, sumado a la permisividad de estos lenguajes para realizar manipulación de memoria directamente y trabajar a muy bajo nivel. De la mano de estos lenguajes apareció otra veta de investigación en lo que se denominó recolectores de basura *conservativos*.

Una categoría de lenguaje que ha quedado prácticamente vacante es un término intermedio entre los lenguajes de muy alto nivel (como Java, Python, Haskell, etc.) y los de muy bajo nivel (como C y C++). El lenguaje de programación D está en esta categoría y, a pesar de haber sido diseñado con soporte de recolección de basura, al ser un lenguaje relativamente nuevo, ha tenido muy poco desarrollo en ese área. El lenguaje tiene todas las limitaciones de lenguajes de bajo nivel como C y C++, pero esas limitaciones suelen estar más aisladas, y provee un poco más de información que puede ser aprovechada por un recolector de la que suelen proveer los demás lenguajes de estas características. Esto presenta una oportunidad única en cuanto a investigación y desarrollo de recolectores que se ajusten a estas características.

1.1 Objetivo

El objetivos de esta tesis es mejorar el recolector de basura de el lenguaje D, investigando el estado del arte en recolección de basura y haciendo un análisis de viabilidad de los algoritmos principales para optar por el que mejor se ajuste a D. Una vez hecho esto se propone implementar una solución y verificar los resultados mediante pruebas experimentales.

Un aspecto importante del análisis y la solución propuesta por este trabajo es participar de la comunidad del lenguaje para poder desarrollar una mejora que sea aceptada y utilizada por dicha comunidad. Por lo tanto el algoritmo o mejora que "mejor se ajuste a D" estará supeditado en gran parte a los requerimientos más urgentes de sus usuarios.

1.2 Alcance

El alcance de este trabajo se limita a los siguientes puntos:

- Explorar los problemas del recolector de basura actual.
- Evaluar cuáles de estos problemas son de mayor importancia para la comunidad de usuarios de D.
- Analizar la viabilidad de algoritmos y optimizaciones para solucionar o minimizar el o los problemas de mayor importancia.
- Implementar una solución o mejora en base al análisis elaborado.
- Comparar mediante la utilización de un banco de pruebas (*benchmark*) la implementación con la actual y posiblemente con otras implementaciones relevantes a fin de cuantificarla.
- Proveer todos los elementos necesarios para que la solución pueda ser adoptada por el lenguaje.

1.3 Limitaciones

Dado que el lenguaje de programación D puede ser enlazado con código objeto C, y por lo tanto interactuar directamente con éste, podrán haber limitaciones en el recolector resultante con respecto a esto. En este trabajo se busca lograr un recolector que sea eficiente para casos en donde el código que interactúa con C esté bien aislado, por lo que estas porciones de código pueden quedar por fuera del recolector de basura o necesitar un manejo especial.

1.4 Organización

Este trabajo se encuentra dividido en 6 capítulos que se describen a continuación:

1. *Introducción*: breve descripción del problema a tratar, presentando los objetivos y alcances del trabajo.
2. *El lenguaje de programación D*: presenta las características principales del lenguaje, destacando aquellas de mayor relevancia para este trabajo.
3. *Recolección de basura*: presenta los algoritmos básicos de recolección de basura y describe el estado del arte.

4. *Recolección de basura en D*: explica los problemas particulares que presenta el lenguaje para la recolección de basura, describe el diseño e implementación del recolector actual, presenta sus principales deficiencias y analiza la viabilidad de diferentes soluciones.
5. *Solución adoptada*: propone una solución a los problemas principales encontrados y define un banco de pruebas para analizar los resultados de las modificaciones hechas.
6. *Conclusión*: describe las conclusiones alcanzadas en base a los resultados del trabajo, analizando puntos pendientes, mostrando trabajos relacionados y proponiendo trabajos futuros.

CAPÍTULO 2

El lenguaje de programación D

2.1 Historia

D es un lenguaje de programación relativamente joven. Nació en 1999 y el 2 de enero de 2007 salió su versión 1.0. Poco tiempo después se continúo el desarrollo del lenguaje en la versión 2.0, que pasó a ser considerada estable aproximadamente en junio de 2010 con el lanzamiento del libro "The D Programming Language" [ALX10], pero aún es un trabajo en progreso.

El lenguaje fue diseñado e implementado por Walter Bright, desarrollador principal de Zortech C++, uno de los primeros compiladores de C++ que compilaba a código nativo, y está fuertemente influenciado por éste. Sin embargo toma muchos conceptos de otros lenguajes de más alto nivel, como Java o incluso lenguajes dinámicos como Perl, Python y Ruby.

El origen del lenguaje está plasmado en su sitio web, en donde se cita [DWEB]:

> It seems to me that most of the "new" programming languages fall into one of two categories: Those from academia with radical new paradigms and those from large corporations with a focus on RAD and the web. Maybe it's time for a new language born out of practical experience implementing compilers.

Esto podría traducirse como:

> Parece que la mayoría de los lenguajes de programación "nuevos" caen en 2 categorías: aquellos académicos con nuevos paradigmas radicales y aquellos de grandes corporaciones con el foco en el desarrollo rápido y web. Tal vez es hora de que nazca un nuevo lenguaje de la experiencia práctica implementando compiladores.

La versión 1.0 fue más bien una etiqueta arbitraria que un indicador real de estabilidad y completitud. Luego de liberarse se siguieron agregando nuevas características al lenguaje hasta que se empezó el desarrollo en paralelo de la versión 2.0 al introducirse el concepto de inmutabilidad y funciones *puras* [1] (a mediados de 2007).

A partir de este momento la versión 1.0 quedó *teóricamente* congelada, introduciendo solo cambios que arreglen errores (*bug fixes*), agregando nuevas características solamente en la versión 2.0 del lenguaje. La realidad es que se hicieron cambios incompatibles a la versión 1.0 del lenguaje en reiteradas ocasiones, pero se fue tendiendo a cada vez introducir menos cambios incompatibles. Sin embargo al día de hoy

[1] Por funciones *puras* en D se entiende que no tienen efectos colaterales. Es decir, una función pura siempre que se llame con la misma entrada producirá el mismo resultado. Esto es análogo a como funcionan los lenguajes funcionales en general, abriendo la puerta a la programación de estilo funcional en D.

el compilador de referencia sigue teniendo algunas características presentes en la especificación del lenguaje sin implementar, por lo que todavía no hay una implementación completa de la versión 1.0 del lenguaje.

El lenguaje ha sido, hasta el desarrollo de la versión 2.0 al menos, un esfuerzo unipersonal de Walter Bright, dados sus problemas a la hora de delegar o aceptar contribuciones. Esto motivó a la comunidad de usuarios de D a crear bibliotecas base alternativas a la estándar (llamada Phobos) en las cuales se pudiera trabajar sin las trabas impuestas por el autor del lenguaje.

En este contexto nacen primero Mango y luego Ares. Mango fue creada por Kris Macleod Bell a principios de 2004 como una biblioteca que provee servicios básicos de entrada/salida (o *I/O* de *input/output* en inglés) de alto rendimiento. Siendo estos servicios algo básico lo más natural hubiera sido que se encuentren en la biblioteca estándar de D pero por las dificultades para contribuir a ésta, se desarrolla como una biblioteca separada. A mediados de 2004 Sean Kelly crea Ares , con las mismas motivaciones pero con la intención de crear una biblioteca base (conocida en inglés como *runtime*) que incluye los servicios básicos que necesita el lenguaje (información de tipos, manejo de excepciones e hilos, creación y manipulación de objetos, recolector de basura, etc.). Al poco tiempo de liberarse Ares, Mango empieza a utilizarla como biblioteca base.

Para comienzos de 2006, se empieza a trabajar en la combinación de ambas bibliotecas para lograr una biblioteca estándar alternativa con un alto grado de cohesión. Finalmente a principios de 2007, coincidiendo por casualidad con la aparición de D 1.0, se anuncia el resultado de este combinación bajo el nombre de Tango, proveyendo una alternativa completa y madura a la biblioteca estándar de D Phobos. A principios de 2008 los principales desarrolladores de Tango (Kris Bell, Sean Kelly, Lars Ivar Igesund y Michael Parker) publican el libro llamado Learn to Tango with D [BKIP08].

Esto por un lado fue un gran avance porque dio un impulso muy considerable al lenguaje pero por otro un gran retroceso, porque todavía al día de hoy D 1.0 tiene dos bibliotecas base, una estándar pero de peor calidad, menos mantenida y usada; y una alternativa de mayor calidad y apertura a la comunidad (pero no estándar). El peor problema es que ambas son **incompatibles**, por lo que un programa hecho con Tango no funciona con Phobos y viceversa (a menos que el programador haya invertido una cantidad de tiempo considerable en asegurarse que funcione con ambas).

Esto hace que la compatibilidad de programas y bibliotecas esté muy fragmentada entre las dos bibliotecas base. Si bien no parece que vaya a haber solución alguna a este problema para D 1.0, D 2.0 va en camino a solucionar este problema ya que utiliza DRuntime, un nuevo intento de Sean Kelly por proveer una biblioteca *runtime* bien organizada y mantenida, que es una adaptación de la biblioteca *runtime* de Tango a D 2.0. Sin embargo Tango no fue adaptada a D 2.0 todavía, y no hay muchas perspectivas de que sea portada en algún momento, por un lado porque en general la comunidad sigue fragmentada entre muchos usuarios de D 1.0 que no están contentos con los cambios introducidos en D 2.0, en su mayoría usuarios de Tango, y que no planean migrar a esa versión; y por otro porque el desarrollo de Phobos 2.0 se ha abierto mucho y tiene muchos colaboradores, por lo tanto la mayor parte de la gente que utiliza D 2.0 está contenta con el estado de Phobos 2.0.

2.2 Descripción general

D es un lenguaje de programación con sintaxis tipo C, multi-paradigma, compilado, con *tipado* fuerte y estático, buenas capacidades tanto de programación de bajo nivel (*system programming*) como de alto nivel; además es compatible de forma binaria con C (se puede enlazar código objeto C con código objeto D). Con estas características, D logra llenar un vacío importante que hay entre lo lenguajes de alto bajo nivel y los de alto nivel [BKIP08]. Si bien tiene herramientas de muy bajo nivel, que por lo tanto son muy propensas a errores, da una infinidad de mecanismos para evitar el uso de estas herramientas a

menos que sea realmente necesario. Además pone mucho énfasis en la programación confiable, para lo cual provee muchos mecanismos para detectar errores en los programas de forma temprana.

Si puede pensarse en C++ como un "mejor C", podría decirse que D es un "mejor C++", ya que el objetivo del lenguaje es muy similar a C++, pero implementa muchas características que jamás pudieron entrar en el estándar de C++ y lo hace de una forma mucho más limpia, ya que no debe lidiar con problemas de compatibilidad hacia atrás, y cuenta con la experiencia del camino recorrido por C++, pudiendo extraer de él los mejores conceptos pero evitando sus mayores problemas.

Una gran diferencia con C++ es que el análisis sintáctico (*parsing*) se puede realizar sin ningún tipo de análisis semántico, dado que a diferencia de éstos su gramática es libre de contexto (*context-free grammar*). Esto acelera y simplifica considerablemente el proceso de compilación [WBB10] [DWOV].

Otra gran diferencia es que D decide incluir recolección de basura como parte del lenguaje, mientras que en el comité de estandarización de C++ nunca se llegó a un consenso para su incorporación.

2.3 Características del lenguaje

A continuación se enumeran las principales características de D, agrupadas por unidades funcionales o paradigmas que soporta [DWLR]:

2.3.1 Programación genérica y meta-programación

La programación genérica se trata de la capacidad de poder desarrollar algoritmos y estructuras independientes de los tipos que manipulan (pero de forma segura o *type-safe*). Esto fue muy popularizado por C++ gracias a su soporte de plantillas (*templates*) y luego otros lenguajes como Java y C# lo siguieron. Sin embargo otros lenguajes proveen formas más avanzadas de programación genérica, gracias a sistemas de tipos más complejos (como Haskell).

La meta-programación se refiere en general a la capacidad de un lenguaje para permitir generar código dentro del mismo programa de forma automática. Esto permite evitar duplicación de código y fue también muy popularizado por el soporte de *templates* de C++, aunque muchos otros lenguajes tienen mejor soporte de meta-programación, en especial los lenguajes dinámicos (como Python).

D provee las siguientes herramientas para realizar programación genérica y meta-programación:

`if` estático (`static if`)
: Esta construcción es similar a la directiva del preprocesador de C/C++ `#if`, pero a diferencia de éste, el `static if` de D tiene acceso a todos los símbolos del compilador (constantes, tipos, variables, etc) [DWSI].

  Ejemplo:

```
static if ((void*).sizeof == 4)
   pragma(msg, "32 bits");
```

Inferencia de tipos básica implícita y explícita (mediante `typeof`)
: Si no se especifica un tipo al declarar una variable, se infiere a partir del tipo de su valor de inicialización [DWIN].

  Ejemplo:

```
static i = 5;       // i es int
const d = 6.0;     // d es double
auto s = "hola"; // s es string (que es un alias de char[])
```

Mediante el uso de `typeof` se puede solicitar el tipo de una expresión arbitraria [DWTO].

Ejemplo:

```
typeof(5 + 6.0) d; // d es double
```

Iteración sobre colecciones (`foreach`)

Cualquier tipo de colección (arreglos estáticos y dinámicos, arreglos asociativos, clases, estructuras o delegados) puede ser iterada mediante la sentencia `foreach` [DWFE].

Ejemplo:

```
int[] a = [ 1, 2, 3 ];
int total = 0;
foreach (i; a)
   total += i;
```

Templates

Tanto clases como funciones pueden ser generalizadas. Esto permite desarrollar algoritmos genéricos sin importar el tipo de los datos de entrada, siempre y cuando todos los tipos tengan una *interfaz* común. Esto también es conocido como *polimorfismo en tiempo de compilación*, y es la forma más básica de programación genérica [DWTP].

Ejemplo:

```
T sumar(T)(T x, T y) { return x + y; }
auto i = sumar!(int)(5, 6);     // i == 11
auto f = sumar!(float)(5, 6); // j == 11.0f
```

Además se pueden definir bloques de declaraciones generalizadas (esto no es posible en C++), permitiendo instanciar dicho bloque con parámetros particulares. Esto sirve como un mecanismo para la reutilización de código, ya que puede incluirse un mismo bloque en distintos lugares (por ejemplo clases). Un bloque generalizado puede verse como una especie de módulo.

Ejemplo:

```
template bloque(T, U) {
   T x;
   U foo(T y);
}

bloque!(int, float).x = 5;
float f = bloque!(int, float).foo(7);
```

La utilidad más prominente de los bloques generalizados se da al acompañarse de *mixins*.

Además las *templates* de D tienen las siguientes características destacables:

Instanciación implícita de funciones generalizadas

El lenguaje es capaz de deducir los parámetros siempre que no hayan ambigüedades.

Ejemplo:

```
auto i = sumar(5, 6);           // i == 11
auto f = sumar(5.0f, 6.0f); // f == 11.0f
```

Especialización explícita y parcial de *templates*

La especialización de *templates* consiste, al igual que en C++, en proveer una implementación especializada para un tipo de dato (o valor) de los parámetros. Especialización parcial se refiere a la capacidad de especializar un parámetro a través de un subtipo. Por ejemplo, se puede especializar un *template* para cualquier tipo de puntero, o para cualquier tipo de arreglo dinámico, sin necesidad de especificar el tipo al que apunta dicho puntero o el tipo almacenado por el arreglo.

Ejemplo de especialización:

```
T sumar(T)(T x, T y)       { return x + y; }
T sumar(T: int)(T x, T y) { return x + y + 1; }
auto i = sumar(5, 6);          // i == 12
auto f = sumar(5.0f, 6.0f) // f == 11.0f
```

Ejemplo de especialización parcial:

```
T sumar(T)(T x, T y)      { return x + y; }
T sumar(T: T*)(T x, T y) { return *x + *y; }
int x = 5, y = 6;
auto i = sumar(&x, &y); // i == 11
float v = 5.0f, w = 6.0f;
auto f = sumar(&v, &w); // f == 11.0f
```

Tipos, valores (incluyendo *strings*) y *templates* como parámetros

Este es otro bloque de construcción importantísimo para la programación genérica en D, ya que combinando *templates* que toman *strings* como parámetro en combinación con *string mixins* pueden hacerse toda clase de meta-programas.

Ejemplo:

```
template hash(string s, uint so_far=0) {
   static if (s.length == 0)
      const hash = so_far;
   else
      const hash = hash!(s[1 .. length], so_far * 11 + s[0]);
}
string s = hash!("hola"); // calculado en tiempo de compilación
```

Cantidad de parámetros variables para *templates*

Esta característica permite implementar tuplas y otros algoritmos que inherentemente deben tomar una cantidad variable de parámetros en tiempo de compilación.

Ejemplo:

```
double sumar(T...)(T t) {
   double res = 0.0;
   foreach (x; t)
      res += x;
   return res;
}
double d = sumar(1, 2.0, 3.0f, 4l); // d == 10.0
```

***CTFE* (*compile-time function execution*)**

Si una función cumple ciertas reglas básicas (como por ejemplo no tener efectos colaterales) puede ser ejecutada en tiempo de compilación en vez de tiempo de ejecución. Esto permite hacer algunos cálculos que no cambian de ejecución en ejecución al momento de compilar, mejorando el rendimiento o permitiendo formas avanzadas de meta-programación. Esta característica se vuelve particularmente útil al combinarse con *string mixins* [DWCF].

Ejemplo:

```
int factorial(int n) {
   if (n == 1)
      return 1;
   else
      return n * factorial(n - 1);
}
static int x = factorial(5); // calculado en tiempo de compilación
int x = factorial(5); // calculado en tiempo de ejecución
```

Esta característica es muy importante para evitar la duplicación de código.

Mixins*, incluyendo *string mixins

La palabra *mixin* tiene significados distintos en varios lenguajes de programación. En D *mixin* significa tomar una secuencia arbitraria de declaraciones e insertarla en el contexto (*scope*) actual. Esto puede realizarse a nivel global, en clases, estructuras o funciones. Esto sirve como un mecanismo para evitar duplicación de código que puede ser introducida por la falta de herencia múltiple [DWMT].

Ejemplo:

```
class A {
   mixin bloque!(int, float);
}
A a = new A;
a.x = 5;
float f = a.foo(a.x);

class B {
   mixin bloque!(long, double);
}
B b = new B;
b.x = 5l;
double d = a.foo(a.x);
```

String mixin se refiere a la capacidad de *incrustar* un *string* que contenga un fragmento de código en un programa como si este fragmento hubiera sido escrito en el código fuente directamente por el programador. Esto permite hacer manipulaciones arbitrariamente complejas en combinación con funciones ejecutadas en tiempo de compilación [DWME] [DWMX].

Ejemplo:

```
string generar_sumar(string var_x, string var_y) {
   return "return " ~ var_x ~ " + " ~ var_y ~ ";";
}
```

```
int sumar(int a, int b) {
   mixin(generar_sumar!("a", "b"));
}
```

Expresiones `is`
: Las *expresiones ‘‘is‘‘* permiten la compilación condicional basada en las características de un tipo [DWIE].

  Ejemplo:

```
T foo(T)(T x) {
   static if (is(T == class))
      return new T;
   else
      return T.init;
}
```

  Esto provee además una forma simple de reflexión en tiempo de compilación.

2.3.2 Programación de bajo nivel (*system programming*)

Por programación de bajo nivel nos referimos a la capacidad de un lenguaje de manipular el hardware directamente, o al menos la memoria. C es probablemente el lenguaje de bajo nivel más popular, seguido por C++.

D presenta muchas características de bajo nivel:

Compila a código de máquina nativo
: Los programas generados por D no son interpretados ni necesitan una máquina virtual como otros lenguajes de más alto nivel como Java, C#, Python, etc [DWOV].

***Assembly* empotrado**
: Provee acceso directo al *hardware* y la posibilidad de utilizar cualquier característica de éste que no esté disponible en el lenguaje.

  Una ventaja sobre C y C++ es que el lenguaje *assembly* utilizado dentro de D está especificado, por lo que se puede mantener la portabilidad entre compiladores incluso cuando se utiliza *assembly* (mientras que no se cambie de arquitectura, por supuesto) [DWIA].

`goto`
: Al igual que C y C++, D provee la flexibilidad del uso de `goto` [DWGT].

Compatibilidad con C
: Soporta todos los tipos de C y es ABI [2] compatible con éste. Esto permite enlazar archivos objeto estándar de C y D en un mismo programa. Además permite interoperar con C a través de `extern (C)` [DWCC].

  Ejemplo:

```
extern (C) printf(const char* format, ...);
printf("3 + 5 == %d\n", 3 + 5); // llama al printf de C
```

[2] Interfaz de Aplicación Binaria (del inglés *Application Binary Interface*).

Manejo de memoria explícito
Permite asignar estructuras en el *stack* o en el *heap*, haciendo uso de los servicios del sistema operativo o la biblioteca estándar de C [DWMM].

Objetos y arreglos *livianos*
Por objetos *livianos* se entiende no-polimórficos. Es decir, un agrupamiento de variables análogo al `struct` de C, sin tabla virtual ni otro tipo de *overhead*. Los arreglos *livianos* son arreglos estáticos como en C, cuyo tamaño es fijo, también sin ningún tipo de *overhead* como C. Además puede asignarse un arreglo dinámicamente usando `malloc()` y utilizar el operador `[]` para accederlo [DWST] [DWCL].

Esto también permite interoperar con C, ya que pueden definirse `structs` y arreglos que pueden ser intercambiados con dicho lenguaje sin problemas.

Ejemplo:

```
struct timeval {
   time_t      tv_sec;
   suseconds_t tv_usec;
}
extern (C) {
   void* malloc(size_t);
   size_t strlen(const char *);
   int gettimeofday(timeval *, void *);
}
char* s = cast(char*) malloc(2);
s[0] = 'C';
s[1] = '\0';
size_t l = strlen(s); // l == 1
timeval tv;
gettimeofday(&tv, null);
```

Rendimiento
La *Programación genérica y meta-programación* permite realizar muchas optimizaciones ya que se resuelve en tiempo de compilación y por lo tanto aumenta el rendimiento en la ejecución [DWTP].

Número de punto flotante de 80 bits
El tipo `real` de D tiene precisión de 80 bits si la plataforma lo soporta (por ejemplo en i386) [DWTY].

Control de alineación de miembros de una estructura
Mediante `align` se puede especificar la alineación a tener en una estructura [DWAL].

Ejemplo:

```
align (1)
struct paquete_de_red {
   char  tipo;
   short valor;
}
// paquete_de_red.sizeof == 3
```

2.3.3 Programación de alto nivel

Programación de alto nivel se refiere a construcciones más avanzadas que una sentencia para iterar; expresiones con una semántica más ricas que proveen de mayor expresividad al programador o le permiten focalizarse de mejor manera en los algoritmos independizándose del *hardware* o de como funciona una computadora. Es exactamente el opuesto a *Programación de bajo nivel (system programming)*.

En general estas características tienen como efecto secundario una mejora de la productividad de los programadores. D adopta herramientas de muchos lenguajes de alto nivel, como Java y Python, por ejemplo:

Manejo automático de memoria
: Al igual que C/C++ y prácticamente cualquier lenguaje imperativo maneja automáticamente el *stack*, pero a diferencia de la mayoría de los lenguajes de bajo nivel, D permite manejar el *heap* de manera automática también a través de un *recolector de basura* [DWGC].

Sistema de paquetes y módulos (similar a Java o Python)
: Un módulo es una unidad que agrupa clases, funciones y cualquier otra construcción de lenguaje. Un paquete es una agrupación de módulos. D asocia un módulo a un archivo fuente (y un archivo objeto cuando éste es compilado) y un paquete a un directorio. A diferencia de C/C++ no necesita de un preprocesador para incluir declaraciones de otros *módulos* (en C/C++ no existe el concepto de módulo, solo de unidades de compilación) [DWMO].

  Ejemplo:

  `a.d`:

```
module a;
void f() {}
```

  `b.d`:

```
module b;
void f() {}
```

  `c.d`:

```
module c;
import a;
import b: f;
a.f();
b.f();
f(); // ejecuta b.f()
```

Funciones y delegados
: Las funciones pueden ser sobrecargadas (funciones con el mismo nombre pero distinta cantidad o tipo de parámetros), pueden especificarse argumentos de entrada, salida o entrada/salida, argumentos por omisión o argumentos evaluados de forma perezosa (*lazy*). Además pueden tener una cantidad de argumentos variables pero manteniendo información de tipos (más seguro que C/C++) [DWFU].

  Los *delegados* son punteros a función con un contexto asociado. Este contexto puede ser un objeto (en cuyo caso la función es un método) o un *stack frame* (en cuyo caso la función es una función anidada).

Además de esto los delegados son ciudadanos de primera clase [3], disponiendo de forma literal (delegado anónimo), lo que permite construcciones de alto nivel muy conveniente. Los argumentos evaluados de forma perezosa no son más que un delegado que se ejecuta solo cuando es necesario.

Ejemplo:

```
bool buscar(T[] arreglo, T item, bool delegate(T x, T y) igual) {
   foreach (t, arreglo)
      if (igual(t, item))
         return true;
   return false;
}
struct Persona {
   string nombre;
}
Persona[] personas;
// llenas personas
Persona p;
p.nombre = "Carlos";
bool encontrado = buscar(personas, p,
                                  (Persona x, Persona y) {
                                     return x.nombre == y.nombre;
                                  }
                         );
```

Arreglos *dinámicos* y arreglos asociativos

Los arreglos *dinámicos* son arreglos de longitud variable manejados automáticamente por el lenguaje (análogos al `std::vector` de C++). Soportan concatenación (a través del operador ~), rebanado o *slicing* (a través del operador `[x..y]`) y chequeo de límites (*bound checking*) [DWAR].

Los arreglos asociativos (también conocidos como *hashes* o diccionarios) también son provistos por el lenguaje [DWAA].

Ambos son ciudadanos de primera clase, disponiendo de forma literal.

Ejemplo:

```
int[] primos = [ 2, 3, 5, 7, 11, 13, 17, 19 ];
primos ~= [ 23, 29 ];
auto menores_que_10 = primos[0..4]; // [ 2, 3, 5, 7 ]
int[string] agenda;
agenda["Pepe"] = 5555_1234;
```

Strings

Al igual que los delegados y arreglos dinámicos y asociativos, los *strings* son ciudadanos de primera clase, teniendo forma literal y siendo codificados en UTF-8/16/32. Son un caso particular de arreglo dinámico y es posible utilizarlos en sentencias `switch`/`case` [DWSR].

Ejemplo:

```
string s = "árbol";
```

[3] Por ciudadano de primera clase se entiende que se trata de un tipo soportado por completo por el lenguaje, disponiendo de expresiones literales anónimas, pudiendo ser almacenados en variables, estructuras de datos, teniendo una identidad intrínseca, más allá de un nombre dado, etc. En realidad los arreglos asociativos no pueden ser expresados como literales anónimos pero sí tienen una sintaxis especial soportada directamente por el lenguaje.

```
switch (s) {
   case "árbol":
      s = "tree";
   default:
      s = "";
}
```

`typedef` y `alias`
El primero define un nuevo tipo basado en otro. A diferencia de C/C++ el tipo original no puede ser implícitamente convertido al tipo nuevo (excepto valores literales), pero la conversión es válida en el otro sentido (similar a los `enum` en C++). Por el contrario, `alias` es análogo al `typedef` de C/C++ y simplemente es una forma de referirse al mismo tipo con un nombre distinto [DWDC].

Ejemplo:

```
typedef int tipo;
int foo(tipo x) {}
tipo t = 10;
int i = 10;
foo(t);
foo(i); // error, no compila
alias tipo un_alias;
un_alias a = t;
foo(a);
```

Documentación embebida
D provee un sistema de documentación embebida, análogo a lo que proveen Java o Python en menor medida. Hay comentarios especiales del código que pueden ser utilizados para documentarlo de forma tal que luego el compilador pueda extraer esa información para generar un documento [DWDO].

Números complejos
D soporta números complejos como ciudadanos de primera clase. Soporta forma literal de números imaginarios y complejos [DWTY].

Ejemplo:

```
ifloat im = 5.0i;
float   re = 1.0;
cfloat c   = re + im; // c == 1.0 + 5.0i
```

2.3.4 Programación orientada a objetos

La orientación a objetos es probablemente el paradigma más utilizado en la actualidad a la hora de diseñar e implementar un programa. D provee muchas herramientas para soportar este paradigma de forma confiable. Entre las características más salientes se encuentran:

Objetos *pesados*
Objetos polimórficos como los de cualquier lenguaje con orientación real a objetos. Estos objetos poseen una tabla virtual para despacho dinámico, todos los métodos son virtuales a menos que se indique lo contrario y tienen semántica de referencia [4]. Estos objetos tienen un *overhead* comparados a los objetos *livianos* pero aseguran una semántica segura para trabajar con orientación a

[4] Semántica de referencia significa que el tipo es tratado como si fuera un puntero. Nunca se hacen copias del objeto, siempre se pasa por referencia.

objetos, evitando problemas con los que se enfrenta C++ (como *slicing* [5]) debido a que permite semántica por valor [6] [DWCL].

D además soporta tipos de retorno covariantes para funciones virtuales. Esto significa que una función sobreescrita por una clase derivada puede retornar un tipo que sea derivado del tipo retornado por la función original sobreescrita [DWFU].

Ejemplo:

```
class A { }
class B : A { }

class Foo {
   A test() { return null; }
}

class Bar : Foo {
   B test() { return null; } // sobreescribe y es covariante con Foo.test()
}
```

Interfaces

D no soporta herencia múltiple pero sí interfaces. Una interfaz es básicamente una tabla virtual, una definición de métodos virtuales que debe proveer una clase. Las interfaces no proveen una implementación de dichos métodos, ni pueden tener atributos. Esto simplifica mucho el lenguaje y no se pierde flexibilidad porque puede conseguirse el mismo efecto de tener herencia múltiple a través de interfaces y *mixins* para proveer una implementación o atributos en común a varias clases que implementan la misma interfaz [DWIF].

Sobrecarga de operadores

La sobrecarga de operadores permite que un objeto tenga una sintaxis similar a un tipo de dato nativo. Esto es muy importante además para la programación genérica [DWOO].

Clases anidadas

Al igual que C (con respecto a `struct`) y C++, pueden anidarse clases dentro de clases. D sin embargo provee la posibilidad de acceder a atributos de la instancia exterior desde la anidada [DWNC].

Ejemplo:

```
class Exterior {
   int m;
   class Anidada {
      int foo() {
         return m; // ok, puede acceder a un miembro de Exterior
      }
   }
}
```

Esto tiene un pequeño *overhead* ya que la clase `Anidada` debe guardar un puntero a la clase `Exterior`. Si no se necesita este comportamiento es posible evitar este *overhead* utilizando `static`, en cuyo caso solo puede acceder a atributos estáticos de la clase `Exterior`.

[5] Este problema se da en C++ cuando se pasa una clase derivada a una función que acepta una clase base por valor como parámetro. Al realizarse una copia de la clase con el constructor de copia de la clase base, se pierden (o *rebanan*) los atributos de la clase derivada, y la información de tipos en tiempo de ejecución (*RTTI*).

[6] Semántica de valor significa que el tipo es tratado como si fuera un valor concreto. En general se pasa por valor y se hacen copias a menos que se utilice explícitamente un puntero.

Ejemplo:

```
class Exterior {
   int m;
   static int n;
   static class Anidada {
      int foo() {
         //return m; // error, miembro de Exterior
         return n; // ok,   miembro estático de Exterior
      }
   }
}
```

Propiedades (*properties*)

En D se refiere a funciones miembro que pueden ser tratadas sintácticamente como campos de esa clase/estructura [DWPR].

Ejemplo:

```
class Foo {
   int data() { return _data; } // propiedad de lectura
   int data(int value) { return _data = value; } // de escritura
   private int _data;
}
Foo f = new Foo;
f.data = 1;        // llama a f.data(1)
int i = f.data; // llama a f.data()
```

Además tipos nativos, clases, estructuras y expresiones tienen *properties* predefinidos, por ejemplo:

`sizeof`
: Tamaño ocupado en memoria (ejemplo: `int.sizeof` -> 4).

`init`
: Valor de inicialización por omisión (ejemplo: `float.init` -> *NaN* [7]).

`stringof`
: Representación textual del símbolo o expresión (ejemplo: `(1+2).stringof` -> `"1 + 2"`).

`mangleof`
: Representación textual del tipo *mutilado* [8] [DWAB].

`alignof`
: Alineación de una estructura o tipo.

Estos son solo los *properties* predefinidos para todos los tipos, pero hay una cantidad considerable de *properties* extra para cada tipo.

[7] Del inglés *Not A Number*, es un valor especial codificado según IEEE 754-2008 [IEEE754] que indica que estamos ante un valor inválido.

[8] *Name mangling* es el nombre dado comúnmente a una técnica necesaria para poder sobrecargar nombres de símbolos. Consiste en codificar los nombres de las funciones tomando como entrada el nombre de la función y la cantidad y tipo de parámetros, asegurando que dos funciones con el mismo nombre pero distintos parámetros (sobrecargada) tengan nombres distintos.

2.3.5 Programación confiable

Programación confiable se refiere a las capacidades o facilidades que provee el lenguaje para evitar fallas de manera temprana (o la capacidad de evitar que ciertas fallas puedan existir directamente). D presta particular atención a esto y provee las siguientes herramientas:

Excepciones
: D soporta excepciones de manera similar a Java: provee `try`, `catch` y `finally`. Esto permite que los errores difícilmente pasen silenciosamente sin ser detectados [DWEX].

`assert`
: Es una condición que debe cumplirse siempre en un programa, como un chequeo de integridad. Esto es muy utilizado en C/C++, donde `assert()` es una *macro* que solo se compila cuando la *macro* `NDEBUG` no está definida. Esto permite eliminar los chequeos de integridad del programa, que pueden ser costosos, para versiones que se suponen estables.

  D lleva este concepto más allá y hace al `assert` parte del lenguaje [DWCP]. Si una verificación no se cumple, lanza una excepción. El `assert` no es compilado cuando se utiliza una opción del compilador.

  Ejemplo:

```
File f = open("archivo");
assert (f.ok());
```

Diseño por contrato
: El diseño por contrato es un concepto introducido por el lenguaje Eiffel a mediados/finales de los '80. Se trata de incorporar en el lenguaje las herramientas para poder aplicar verificaciones formales a las interfaces de los programas.

  D implementa las siguientes formas de diseño por contrato (todas se ejecutan siempre y cuando no se compile en modo *release*, de manera de no sacrificar rendimiento cuando es necesario) [DWCP]:

  Pre y post condiciones
  : Ejemplo:

```
double raiz_cuadrada(double x)
in { // pre-condiciones
   assert (x >= 0.0);
}
out (resultado) { // post-condiciones
   assert (resultado >= 0.0);
   if (x < 1.0)
      assert (resultado < x);
   else if (x > 1.0)
      assert (resultado > x);
   else
      assert (resultado == 1);
}
body {
   // implementación
}
```

  Invariantes de representación
  : La invariante de representación es un método de una clase o estructura que es verificada

cuando se completa su construcción, antes de la destrucción, antes y después de ejecutar cualquier función miembro pública y cuando se lo requiere de forma explícita utilizando `assert`.

Ejemplo:

```
class Fecha {
      int dia;
      int hora;
      invariant() {
         assert(1 <= dia && dia <= 31);
         assert(0 <= hora && hora < 24);
      }
}
```

Pruebas unitarias

Es posible incluir pequeñas pruebas unitarias en el lenguaje. Éstas son ejecutadas (cuando no se compila en modo *release*) al comenzar el programa, antes de que la función `main()` [DWUT].

Ejemplo:

```
unittest {
   Fecha fecha;
   fecha.dia = 5;
   assert (fecha.dia == 5);
   assert (fecha);
}
```

Orden de construcción estática

A diferencia de C++, D garantiza el orden de inicialización de los módulos. Si bien en C++ no hay módulos si no unidades de compilación, es posible que se ejecute código antes del `main()` en C++, si hay, por ejemplo, instancias globales con un constructor definido. C++ no garantiza un orden de inicialización, lo que trae muchos problemas. En D se define el orden de inicialización y es el mismo orden en que el usuario importa los módulos [DWMO].

Inicialización garantizada

Todas las variables son inicializadas por el lenguaje (a menos que el usuario pida explícitamente que no lo sean) [DWTY] [DWVI]. Siempre que sea posible se elijen valores de inicialización que permitan saber al programador que la variable no fue inicializada explícitamente, de manera de poder detectar errores de manera temprana.

Ejemplo:

```
double d;       // inicializado a NaN
int x;          // inicializado a 0
Fecha f;        // inicializado a null
byte[5] a;      // inicializados todos los valores a 0
long l = void; // NO inicializado (explícitamente)
```

***RAII* (*Resource Adquisition Is Initialization*)**

Es una técnica muy utilizada en C++ que consiste en reservar recursos por medio de la construcción de un objeto y liberarlos cuando se libera éste. Al llamarse al destructor de manera automática cuando se sale del *scope*, se asegura que el recurso será liberado también.

Esta técnica es la base para desarrollar código seguro en cuanto a excepciones (*exception-safe*) [SUTT99].

En D no es tan común utilizar *RAII* dada la existencia del recolector de basura (en la mayoría de los casos el recurso a administrar es sencillamente memoria). Sin embargo en los casos en donde es necesario, puede utilizarse *RAII* mediante la utilización de la palabra reservada `scope`, que limita la vida de un objeto a un bloque de código [DWES].

Ejemplo:

```
class Archivo {
   this() { /* adquiere recurso */ }
   ~this() { /* libera recurso */ }
}
void f() {
   scope Archivo archivo = new Archivo;
   // uso de archivo
} // en este punto se llama al destructor de archivo
```

Guardias de bloque (*scope guards*)

Además de poder limitar la vida de una instancia a un *scope*, es posible especificar un bloque de código arbitrario a ejecutar al abandonar un *scope*, ya sea cuando se sale del *scope* normalmente o por una falla [DWES].

Ejemplo:

```
int f(Lock lock) {
   lock.lock();
   scope (exit)
      lock.unlock();              // ejecutado siempre que salga de f()
   auto trans = new Transaccion;
   scope (success)
      trans.commit();             // ejecutado si sale con "return"
   scope (failure)
      trans.rollback();           // ejecutado si sale por una excepción
   if (condicion)
      throw Exception("error");   // lock.unlock() y trans.rollback()
   else if (otra_condicion)
      return 5;                   // lock.unlock() y trans.commit()
   return 0;                      // lock.unlock() y trans.commit()
}
```

Esta es una nueva forma de poder escribir código *exception-safe*, aunque el programador debe tener un poco más de cuidado de especificar las acciones a ejecutar al finalizar el *scope*.

Primitivas de sincronización de hilos

La programación multi-hilo está directamente soportada por el lenguaje, y se provee una primitiva de sincronización al igual que Java. La palabra reservada `synchronized` puede aparecer como modificador de métodos (en cuyo caso se utiliza un *lock* por clase para sincronizar) o como una sentencia, en cuyo caso se crea un *lock* global por cada bloque `synchronized` a menos que se especifique sobre qué objeto realizar la sincronización [DWSY]. Por ejemplo:

```
class Foo {
   synchronized void bar() { /* cuerpo */ }
}
```

Es equivalente a:

```
class Foo {
   void bar() {
      synchronized (this) { /* cuerpo */ }
   }
}
```

2.4 Compiladores

Hay, hasta el momento, 3 compiladores de D de buena calidad: DMD, GDC y LDC.

DMD es el compilador de referencia, escrito por Walter Bright. El *front-end* [9] de este compilador ha sido liberado bajo licencia Artistic/GPL y es utilizado por los otros dos compiladores, por lo tanto en realidad hay solo un compilador disponible con 3 *back-ends* [10] diferentes.

Con DMD 1.041 se publicó el código fuente completo del compilador, pero con una licencia muy restrictiva para uso personal, por lo que el único efecto logrado por esto es que la gente pueda mandar parches o correcciones del compilador pero no lo convierte en Software Libre, siendo el único de los 3 compiladores que no tiene esta característica.

El compilador GDC es el *front-end* de DMD utilizando al compilador GCC como *back-end*. Fue un muy buen compilador pero estuvo abandonado por casi tres años. A mediados de este año recibió un nuevo impulso y de a poco se está poniendo al día con los *front-ends* actuales de DMD 1.0 y 2.0, aunque la versión 2.0 viene bastante más rezagada y todavía no es una alternativa viable a DMD.

LDC sufrió una suerte similar, es un compilador joven que utiliza como *back-end* a LLVM (una infraestructura modera para construir compiladores), nacido a mediados de 2007 como un proyecto personal y privado de Tomas Lindquist Olsen, que estuvo trabajando de forma privada en el proyecto hasta mediados de 2008, momento en que decide publicar el código mediante una licencia libre. Para ese entonces el compilador era todavía inestable y faltaban implementar varias cosas, pero el estado era lo suficientemente bueno como para captar varios colaboradores muy capaces, como Christian Kamm y Frits Van Bommel que rápidamente se convirtieron en parte fundamental del proyecto. El primer *release* (0.9) de una versión relativamente completa y estable fue a principios de 2009 que fue seguido por la versión 0.9.1 que como puntos más salientes agregó soporte para x86-64 y assembly embebido. El compilador tuvo un crecimiento excepcional pero estuvo muy inactivo por algún tiempo y, si bien sigue siendo mantenido, en general los nuevos *front-end* de DMD llevan tiempo de integrar y no está al día con el *back-end* de LLVM (por ejemplo desde que se actualizó para utilizar LLVM 2.7 que perdió la capacidad de generar símbolos de depuración).

Además de estos compiladores hay varios otros experimentales, pero ninguno de ellos de calidad suficiente todavía. Por ejemplo hay un compilador experimental que emite *CIL* (*Common Intermediate Language*), el *bytecode* de .NET, llamado DNet. También hay un *front-end* escrito en D, llamado Dil.

Originalmente, dado que GDC estaba siendo mantenido y que LDC no existía, este trabajo iba a ser realizado utilizando GDC como compilador, dado que al ser Software Libre podía ser modificado de ser necesario. Pero finalmente, dada la poca confiabilidad que presenta la continuidad del desarrollo de tanto GDC como LDC, y que el código de DMD está disponible en su totalidad (aunque no sea Software Libre por completo), se optó por utilizar este último, dado que es la implementación de referencia que fue más constantemente mantenida y desarrollada.

[9] *Front-end* es la parte del compilador encargada de hacer el análisis léxico, sintáctico y semántico del código fuente, generando una representación intermedia que luego el *back-end* convierte a código de máquina.

[10] El *back-end* es la parte del compilador encargada de convertir la representación intermedia generada por el *front-end* a código de máquina.

CAPÍTULO 3

Recolección de basura

3.1 Introducción

Recolección de basura se refiere a la recuperación automática de memoria del *heap* [1] una vez que el programa ha dejado de hacer referencia a ella (y por lo tanto, ha dejado de utilizarla).

A medida que el tiempo pasa, cada vez los programas son más complejos y es más compleja la administración de memoria. Uno de los aspectos más importantes de un recolector de basura es lograr un mayor nivel de abstracción y modularidad, dos conceptos claves en la ingeniería de software [JOLI96]. En particular, al diseñar o programar bibliotecas, de no haber un recolector de basura, **la administración de memoria pasa a ser parte de la interfaz**, lo que produce que los módulos tengan un mayor grado de acoplamiento.

Además hay una incontable cantidad de problemas asociados al manejo explícito de memoria que simplemente dejan de existir al utilizar un recolector de basura. Por ejemplo, los errores en el manejo de memoria (como *buffer overflows* [2] o *dangling pointers* [3]) son la causa más frecuente de problemas de seguridad [BEZO06].

La recolección de basura nació junto a Lisp a finales de 1950 y en los siguientes años estuvo asociada principalmente a lenguajes funcionales, pero en la actualidad está presente en prácticamente todos los lenguajes de programación, de alto o bajo nivel, aunque sea de forma opcional. En los últimos 10 años tuvo un gran avance, por la adopción en lenguajes de desarrollo rápido utilizados mucho en el sector empresarial, en especial Java, que fue una plataforma de facto para la investigación y desarrollo de recolectores de basura (aunque no se limitaron a este lenguaje las investigaciones).

En las primeras implementaciones de recolectores de basura la penalización en el rendimiento del programa se volvía prohibitiva para muchas aplicaciones. Es por esto que hubo bastante resistencia a la utilización de recolectores de basura, pero el avance en la investigación fue haciendo que cada vez sea

[1] *Heap* es un área de memoria que se caracteriza por ser dinámica (a diferencia del área de memoria estática que está disponible durante toda la ejecución de un programa). Un programa puede reservar memoria en tiempo de ejecución según sea necesario y liberarla cuando ya no la necesita. A diferencia del *stack*, la duración de la *reserva* no está atada a un bloque de código.

[2] Un *buffer overflow* (*desbordamiento de memoria* en castellano) se produce cuando se copia un dato a un área de memoria que no es lo suficientemente grande para contenerlo. Esto puede producir que el programa sea abortado por una violación de segmento, o peor, sobreescribir un área de memoria válida, en cuyo caso los resultados son impredecibles.

[3] Un *dangling pointer* (*puntero colgante* en castellano) es un puntero que apunta a un área de memoria inválida. Ya sea porque el elemento apuntado no es el mismo tipo o porque la memoria ya ha sido liberada. Al ser desreferenciado, los resultados son impredecibles, el programa podría abortarse por una violación de segmento o podrían pasar peores cosas si el área de memoria fue re-asignada para almacenar otro objeto.

una alternativa más viable al manejo manual de memoria, incluso para aplicaciones con altos requerimientos de rendimiento. En la actualidad un programa que utiliza un recolector moderno puede ser comparable en rendimiento con uno que utiliza un esquema manual. En particular, si el programa fue diseñado con el recolector de basura en mente en ciertas circunstancias puede ser incluso más eficiente que uno que hace manejo explícito de la memoria. Muchos recolectores mejoran la localidad de referencia [4], haciendo que el programa tenga un mejor comportamiento con el caché y la memoria virtual.

El recolector de basura debe tener un comportamiento correcto y predecible para que sea útil, si el programador no puede confiar en el recolector de basura, éste se vuelve más un problema que una solución, porque introduce nuevos puntos de falla en los programas, y lo que es peor, puntos de falla no controlados por el programador, volviendo mucho más difícil la búsqueda de errores.

3.1.1 Conceptos básicos

Los programas pueden hacer uso principalmente de 4 áreas de memoria:

Registros
: Se trata de la memoria más básica de una computadora. Es el área de memoria en la que puede operar realmente el procesador, es extremadamente escasa y generalmente su uso es administrado por el lenguaje de programación (o compilador más específicamente). Excepto en situaciones muy particulares, realizando tareas de muy bajo nivel, un programador nunca manipula los registros explícitamente.

Área de memoria estática
: Es la forma de memoria más simple que un programador utiliza explícitamente. En general las variables globales se almacenan en este área, que es parte inherente del programa y está disponible durante toda su ejecución, por lo tanto nunca cambia su capacidad en tiempo de ejecución. Es la forma más básica de administrar memoria, pero tiene una limitación fundamental: **el tamaño de la memoria tiene que ser conocido en tiempo de compilación**. Los primeros lenguajes de programación solo contaban con este tipo de memoria (además de los registros del procesador).

***Stack* (pila)**
: Los primeros lenguajes de programación que hicieron uso de una pila aparecieron en el año 1958 (Algol-58 y Atlas Autocode) y fueron los primeros en introducir estructura de bloques, almacenando las variables locales a estos bloques utilizando una pila [JOLI96]. Esto permite utilizar recursividad y tener un esquema simple de memoria dinámica. Sin embargo este esquema es muy limitado porque el orden de reserva y liberación de memoria tiene que estar bien establecido. Una celda [5] asignada antes que otra nunca puede ser liberada antes que aquella.

Heap
: A diferencia del *stack*, el *heap* provee un área de memoria que puede ser obtenida dinámicamente pero sin limitaciones de orden. Es el tipo de memoria más flexible y por lo tanto el más complejo de administrar; razón por la cual existen los recolectores de basura.

La recolección de basura impone algunas restricciones sobre la manera de utilizar el *heap*. Debido a que un recolector de basura debe ser capaz de determinar el grafo de conectividad de la memoria en uso, es

[4] Localidad de referencia es la medida en que los accesos sucesivos de memoria cercana espacialmente son cercanos también en el tiempo. Por ejemplo, un programa que lee todos los elementos de una matriz contigua de una vez o que utiliza la misma variable repetidamente tiene buena localidad referencia. Una buena localidad de referencia interactúa bien con la memoria virtual y caché, ya que reduce el conjunto de trabajo (o *working set*) y mejora la probabildad de éxito (*hit rate*).

[5] En general en la literatura se nombra a una porción de memoria asignada individualmente *celda*, *nodo* u *objeto* indistintamente. En este trabajo se utilizará la misma nomenclatura (haciendo mención explícita cuando alguno de estos términos se refiera a otra cosa, como al nodo de una lista o a un objeto en el sentido de programación orientada a objetos).

necesario que el programa siempre tenga alguna referencia a las celdas activas en los registros, memoria estática o *stack* (normalmente denominado *root set*).

Esto implica que una celda sea considerada basura si y sólo si no puede ser alcanzada a través del grafo de conectividad que se comienza a recorrer desde el *root set*. Por lo tanto, una celda está *viva* si y sólo si su dirección de memoria está almacenada en una celda *raíz* (parte del *root set*) o si está almacenada en otra celda *viva* del *heap*.

Cabe aclarar que esta es una definición conceptual, asumiendo que el programa siempre limpia una dirección de memoria almacenada en el *root set* o una celda del *heap* cuando la celda a la que apunta no va a ser utilizada nuevamente. Esto no es siempre cierto y los *falsos positivos* que esto produce se conoce como un tipo de pérdida de memoria (que es posible incluso al utilizar un recolector de basura) llamada pérdida de memoria *lógica*. Esto puede no ser evitable (incluso cuando el programador no cometa errores) en lenguajes de programación que requieran un recolector de basura conservativo.

Por último, siendo que el recolector de basura es parte del programa de forma indirecta, es común ver en la literatura que se diferencia entre dos partes del programa, el recolector de basura y el programa en sí. A la primera se la suele denominar simplemente *recolector* y a la segunda *mutator*, dado que es la única que modifica (o *muta*) el grafo de conectividad.

3.1.2 Recorrido del grafo de conectividad

El problema de encontrar las celdas *vivas* de un programa se reduce a recorrer un grafo dirigido. El grafo se define como:

$$G = (V, A)$$

Donde V es el conjunto de vértices, dado por las celdas de memoria y A es un conjunto de pares ordenados (aristas), dado por la relación $M \to N$ (es decir, los punteros).

El grafo comienza a recorrerse desde el *root set* y todos los vértices que fueron visitados componen el *live set*; el resto de los vértices son *basura*.

Más formalmente, Definimos:

Camino

Es una secuencia de vértices tal que cada uno de los vértices tiene una arista al próximo vértice en la secuencia. Todo camino finito tiene un *vértice inicial* y un *vértice final* (llamados en conjunto *vértices terminales*). Cualquier vértice no terminal es denominado *vértice interior*.

$$\underset{v_1 \to v_N}{C} = \left\{ v_1, \ldots, v_N \in V \Big/ \underset{i \in [1, N-1]}{\forall v_i} \exists (v_i \to v_{i+1}) \in A \right\}$$

Un camino cuyos *vértices terminales* coinciden, es decir $v_1 = v_N$, es denominado **Ciclo**. Cabe notar que los *vértices terminales* de un ciclo son completamente arbitrarios, ya que cualquier *vértice interior* puede ser un *vértice terminal*.

Conexión

Decimos que M está *conectado* a N si y sólo si existe un camino de M a N.

$$M \mapsto N \iff \exists \underset{M \to N}{C} \in G$$

Live set

Es el conjunto de celdas *vivas* está dado por todos los vértices (v) del grafo para los cuales existe una raíz en el *root set* que esté conectada a él.

$$Live\ set = \left\{ v \in V \Big/ \left(\exists r \in Root\ set / r \mapsto v \right) \right\}$$

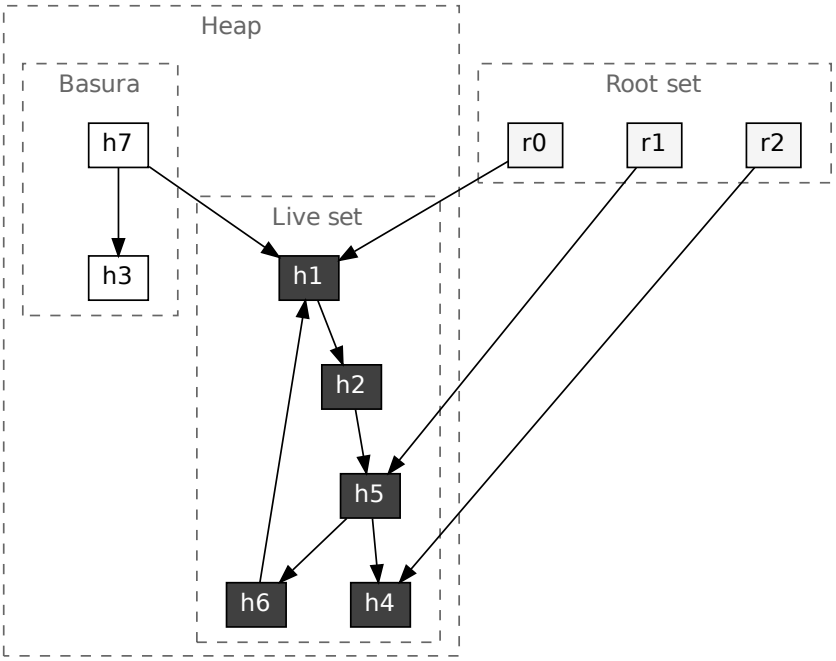


Figura 3.1: Distintas partes de la memoria, incluyendo relación entre *basura*, *live set*, *heap* y *root set*.

Basura

La basura, o celdas *muertas*, quedan determinadas entonces por todas las celdas del *heap* que no son parte del *live set*.

$$Basura = V - Live\ set$$

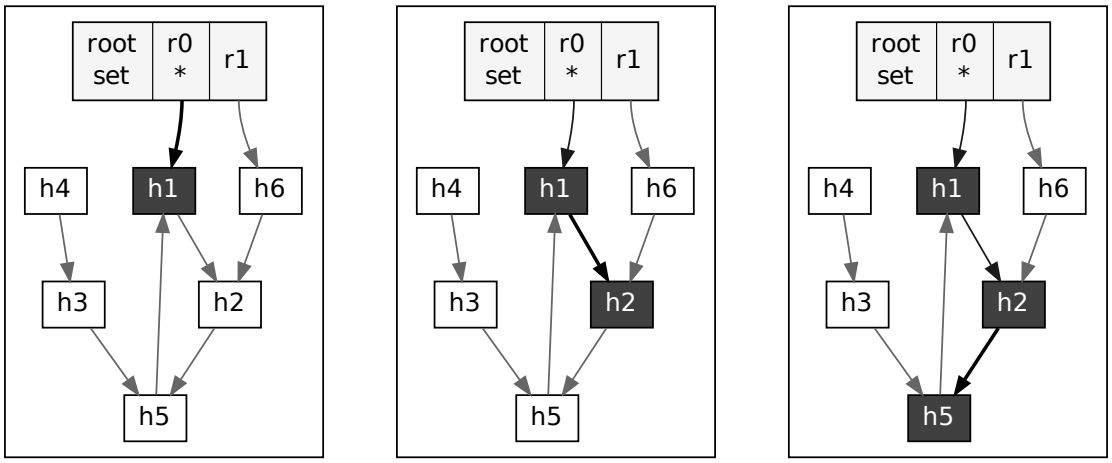
El *Live set* y la *Basura* conforman una partición del *heap* (ver figura 3.1).

Al proceso de visitar los vértices *conectados* desde el *root set* se lo denomina *marcado*, *fase de marcado* o *mark phase* en inglés, debido a que es necesario marcar los vértices para evitar visitar dos veces el mismo nodo en casos en los que el grafo contenga ciclos. De forma similar a la búsqueda, que puede realizarse *primero a lo ancho* (*breadth-first*) o *primero a lo alto* (*depth-first*) del grafo, el marcado de un grafo también puede realizarse de ambas maneras. Cada una podrá o no tener efectos en el rendimiento, en particular dependiendo de la aplicación puede convenir uno u otro método para lograr una mejor localidad de referencia.

Un algoritmo simple (recursivo) de marcado *primero a lo alto* puede ser el siguiente (asumiendo que partimos con todos los vértices sin marcar) [6]:

```
function mark(v) is
   if not v.marked
      v.marked = true
      foreach (src, dst) in v.edges
         mark(dst)

function mark_phase() is
```

[6] Para presentar los algoritmos se utiliza una forma simple de pseudo-código. El pseudo-código se escribe en inglés para que pueda ser más fácilmente contrastado con la literatura, que está en inglés. Para diferenciar posiciones de memoria y punteros de las celdas en sí, se usa la misma sintaxis que C, `r*` denota una referencia o puntero y `*r` denota "objeto al que apunta `r`". Se sobreentiende que `r = o` siempre toma la dirección de memoria de `o`.



(a) Se comienza a marcar el grafo por la raíz r0.

(b) Luego de marcar el nodo `h1`, se procede al `h2`.

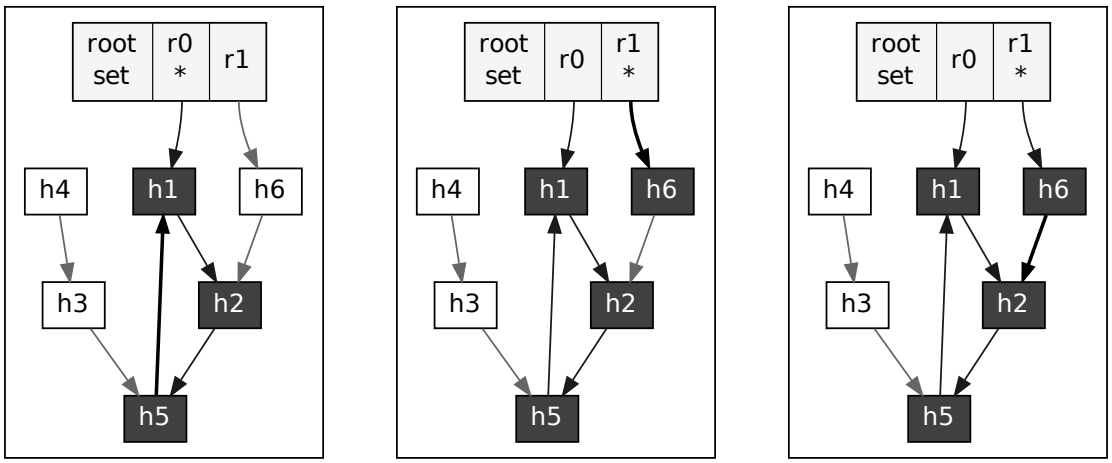
(c) Luego sigue el nodo h5.

Figura 3.2: Ejemplo de marcado del grafo de conectividad (parte 1).

```
foreach r in root_set
   mark(r)
```

Una vez concluido el marcado, sabemos que todos los vértices con la marca son parte del *live set* y que todos los vértices no marcados son *basura*. Esto es conocido también como **abstracción bicolor**, dado que en la literatura se habla muchas veces de *colorear* las celdas. En general, una celda sin marcar es de color blanco y una marcada de color negro.

Puede observarse un ejemplo del algoritmo en la figura 3.2, en la cual se marca el sub-grafo apuntando por `r0`. Luego se marca el sub-grafo al que apunta `r1` (ver figura 3.3), concluyendo con el marcado del grafo completo, dejando sin marcar solamente las celdas *basura* (en blanco).



(a) El nodo h5 tiene una arista al h1, pero el h1 ya fue visitado, por lo tanto no se visita nuevamente.

(b) Se concluye el marcado del sub-grafo al que conecta r0, se procede a marcar el sub-grafo al que conecta r1, marcando al nodo h6.

(c) El nodo h6 tiene una arista al h2, pero éste ya fue marcado por lo que no se vuelve a visitar. No hay más raíces, se finaliza el marcado del grafo.

Figura 3.3: Ejemplo de marcado del grafo de conectividad (parte 2).

3.1.3 Abstracción tricolor

Muchos algoritmos utilizan tres colores para realizar el marcado. El tercer color, gris generalmente, indica que una celda debe ser visitada. Esto permite algoritmos *concurrentes* e *incrementales*, además de otro tipo de optimizaciones. Entonces, lo que plantea esta abstracción es una nueva partición del heap al momento de marcar, esta vez son tres porciones: blanca, gris y negra.

Al principio todas las celdas se pintan de blanco, excepto el *root set* que se pinta de gris. Luego se van obteniendo celdas del conjunto de las grises y se las pinta de negro, pintando sus hijas directas de gris.

Una vez que no hay más celdas grises, tenemos la garantía de que las celdas negras serán el *live set* y las celdas blancas *basura*. Esto se debe a que siempre se mantiene esta invariante: **ninguna celda negra apunta directamente a una celda blanca** (considerando como atómica la operación de pintar una celda de negro y a sus hijas directas de gris). Las celdas blancas siempre son apuntadas por celdas blancas o grises. Entonces, siempre que el conjunto de celdas grises sea vacío, no habrán celdas negras conectadas a blancas, siendo las celdas blancas *basura*.

El algoritmo básico para marcar con tres colores es el siguiente (asumiendo que todas las celdas parten pintadas de blanco, es decir, el conjunto blanco contiene todas las celdas de memoria y los conjuntos negro y gris están vacíos):

```
function mark_phase() is
   foreach r in root_set
      gray_set.add(r)
   while not gray_set.empty()
      v = gray_set.pop()
      black_set.add(v)
      foreach (src, dst) in v.edges
         if dst in white_set
            white_set.remove(dst)
            gray_set.add(dst)
```

Si bien este algoritmo no es recursivo, tiene un requerimiento de espacio $O(|Live\ set|)$. Un ejemplo donde se aprecia esto a simple vista es cuando el *Live set* resulta una lista simplemente enlazada, en cuyo caso el *gray_set* deberá almacenar todos los nodos del *Live set*.

3.1.4 Servicios

En general todos los algoritmos de recolección de basura utilizan servicios de una capa inferior [7] y proveen servicios a una capa superior [8].

A continuación se presentan las primitivas en común que utilizan todos los recolectores a lo largo de este documento.

Servicios utilizados por el recolector son los siguientes:

$alloc() \rightarrow cell$
: Obtiene una nueva celda de memoria. El mecanismo por el cual se obtiene la celda es indistinto para esta sección, puede ser de una lista libre, puede ser de un administrador de memoria de más bajo nivel provisto por el sistema operativo o la biblioteca estándar de C (`malloc()`), etc.

[7] En general estos servicios están provistos directamente por el sistema operativo pero también pueden estar dados por un administrador de memoria de bajo nivel (o *low level allocator* en inglés).

[8] En general estos servicios son utilizados directamente por el lenguaje de programación, pero pueden ser utilizados directamente por el usuario del lenguaje si éste interatúa con el recolector, ya sea por algún requerimiento particular o porque el lenguaje no tiene soporte diercto de recolección de basura y el recolector está implementado como una biblioteca de usuario.

Cómo organizar la memoria es un área de investigación completa y si bien está estrechamente relacionada con la recolección de basura, en este trabajo no se prestará particular atención a este aspecto (salvo casos donde el recolector impone una cierta organización de memoria en el *low level allocator*). Por simplicidad también asumiremos (a menos que se indique lo contrario) que las celdas son de tamaño fijo. Esta restricción normalmente puede ser fácilmente relajada (en los recolectores que la tienen).

$free(cell)$
: Libera una celda que ya no va a ser utilizada. La celda liberada debe haber sido obtenida mediante `alloc()`.

Y los servicios básicos proporcionados por el recolector son los siguientes:

$new() \rightarrow cell$
: Obtiene una celda de memoria para ser utilizada por el programa.

$update(ref, cell)$
: Notifica al recolector que la referencia ref ahora apunta a $cell$. Visto más formalmente, sería análogo a decir que hubo un cambio en la conectividad del grafo: la arista $src \rightarrow old$ cambia por $src \rightarrow new$ (donde src es la celda que contiene la referencia ref, old es la celda a la que apunta la referencia ref y new es el argumento $cell$). Si $cell$ es `null`, sería análogo a informar que se elimina la arista $src \rightarrow old$.

$del(cell)$
: Este servicio, según el algoritmo, puede ser utilizado para informar un cambio en la conectividad del grafo, la eliminación de una arista (análogo a $update(ref, null)$ pero sin proporcionar información sobre la arista a eliminar). Esto es generalmente útil solo en *conteo de referencias*. Para otros recolectores puede significar que el usuario asegura que no hay más referencias a esta celda, es decir, análogo a eliminar el conjunto de aristas $\{(v, w) \in A, v \in Live\ set, w \in Live\ set / w = cell\}$.

$collect()$
: Este servicio indica al recolector que debe hacer un análisis del grafo de conectividad en busca de *basura*. Generalmente este servicio es invocado por el propio recolector cuando no hay más celdas reciclables.

No todos los servicios son implementados por todos los recolectores, pero son lo suficientemente comunes como para describirlos de forma general en esta sección. Algunos son principalmente ideados para uso interno del recolector, aunque en ciertas circunstancias pueden ser utilizados por el usuario también.

3.2 Algoritmos clásicos

En la literatura se encuentran normalmente referencias a tres algoritmos clásicos, que son utilizados generalmente como bloques básicos para construir recolectores más complejos. Se presentan las versiones históricas más simples a fin de facilitar la comprensión conceptual.

3.2.1 Conteo de referencias

Se trata del algoritmo más antiguo de todos, implementado por primera vez por John McCarthy para Lisp a finales de 1950. Se trata de un método *directo* e *incremental* por naturaleza, ya que distribuye la carga de la recolección de basura durante toda la ejecución del programa, cada vez que el *mutator* cambia la conectividad de algún nodo del grafo de conectividad.

El método consiste en tener un contador asociado a cada celda que contenga la cantidad de celdas **vivas** que apuntan a ésta. Es decir, es la cardinalidad del conjunto de aristas que tienen por destino a la celda. Formalmente podemos definir el contador $rc(v)$ (de *reference counter* en inglés) de la siguiente manera:

$$rc(v) = \left| \left\{ (v_1, v_2) \in A / v_1 \in Live\ set \cup Root\ set \wedge v_2 = v \right\} \right|$$

El *mutator* entonces debe actualizar este contador cada vez que el grafo de conectividad cambia, es decir, cada vez que se agrega, modifica o elimina una arista del grafo (o visto de una forma más cercana al código, cada vez que se agrega, modifica o elimina un puntero).

Esta invariante es fundamental para el conteo de referencias, porque se asume que si el contador es 0 entonces el *mutator* no tiene ninguna referencia a la celda y por lo tanto es *basura*:

$$rc(v) = 0 \Rightarrow v \in Basura$$

Para mantener esta invariante el *mutator*, cada vez que cambia un puntero debe decrementar en 1 el contador de la celda a la que apuntaba antiguamente e incrementar en 1 el contador de la celda a la que apunta luego de la modificación. Esto asegura que la invariante se mantenga durante toda la ejecución del programa. Si al momento de decrementar un contador éste queda en 0, la celda asociada debe liberarse de forma de poder ser reciclada. Esto implica que si esta celda almacena punteros, los contadores de las celdas apuntadas deben ser decrementados también, porque solo deben almacenarse en el contador las aristas del *live set* para mantener la invariante. De esto puede resultar que otros contadores de referencia queden en 0 y más celdas sean liberadas. Por lo tanto, teóricamente la complejidad de eliminar una referencia puede ser $O(|Live\ set|)$ en el peor caso.

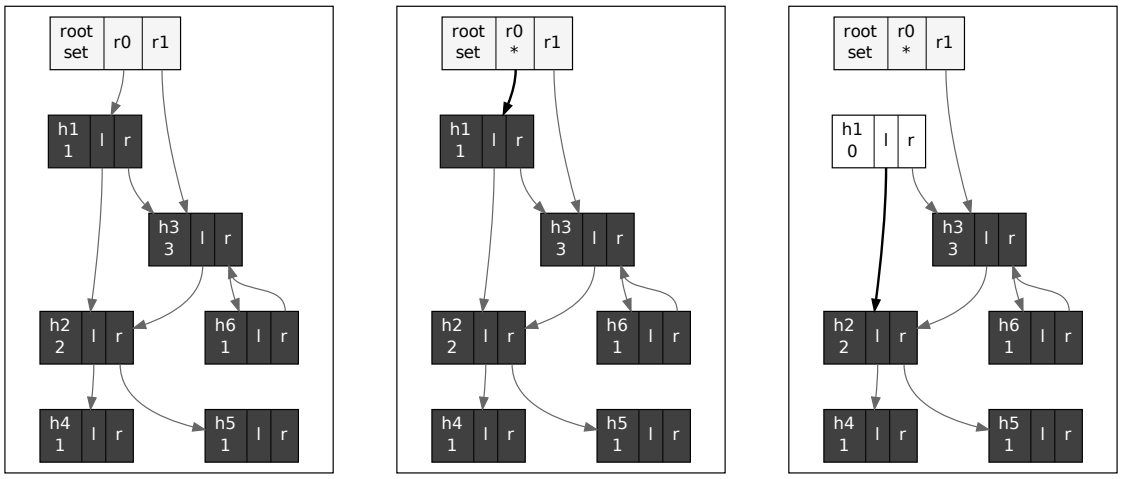
Las primitivas implementadas para este tipo de recolector son las siguientes (acompañadas de una implementación básica):

```
function new() is
   cell = alloc()
   if cell is null
      throw out_of_memory
   cell.rc = 1
   return cell

function del(cell) is
   cell.rc = cell.rc - 1
   if cell.rc is 0
      foreach child* in cell.children
         del(*child)
      free(cell)

function update(ref*, cell) is
   cell.rc = cell.rc + 1
   del(*ref)
   *ref = cell
```

Ciclos

El conteo de referencias tiene, sin embargo, un problema fundamental: **falla con estructuras cíclicas**. Esto significa que siempre que haya un ciclo en el grafo de conectividad, hay una pérdida de memoria potencial en el programa.

Cuando esto sucede, las celdas que participan del ciclo tienen siempre su contador mayor que 0, sin embargo puede suceder que ningún elemento del *root set* apunte a una celda dentro del ciclo, por lo



(a) Estado inicial del grafo de conectividad.

(b) Al ejecutarse `update(r0, null)`, se comienza por visitar la celda `h1`.

(c) Se decrementa el contador de `h1` quedando en 0 (pasa a ser *basura*). Se elimina primero `h1.l` y luego `h1.r`.

Figura 3.4: Eliminación de la referencia `r0` → `h1` (parte 1).

tanto el ciclo es *basura* (al igual que cualquier otra celda para la cual hayan referencias desde el ciclo pero que no tenga otras referencias externas) y sin embargo los contadores no son 0. Los ciclos, por lo tanto, violan la invariante del conteo de referencia.

Hay formas de solucionar esto, pero siempre recaen en un esquema que va por fuera del conteo de referencias puro. En general los métodos para solucionar esto son variados y van desde realizar un marcado del sub-grafo para detectar ciclos y liberarlos hasta tener otro recolector completo de *emergencia*; pasando por tratar los ciclos como un todo para contar las referencias al ciclo completo en vez de a cada celda en particular.

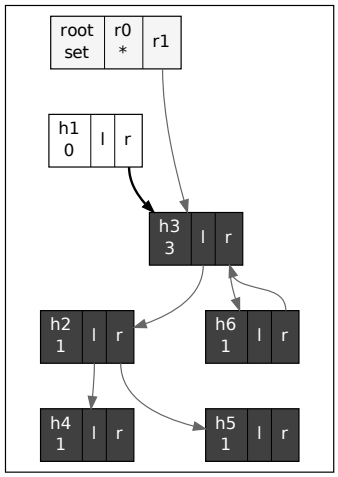
Incluso con este problema, el conteo de referencia sin ningún tipo de solución en cuanto a la detección y recolección de ciclos fue utilizado en muchos lenguajes de programación sin que su necesidad sea tan evidente. Por ejemplo Python agregó recolección de ciclos en la versión 2.0 [NAS00] (liberada en octubre de 2000) y PHP recién agrega detección de ciclos en la versión 5.3 [PHP530].

Ejemplo

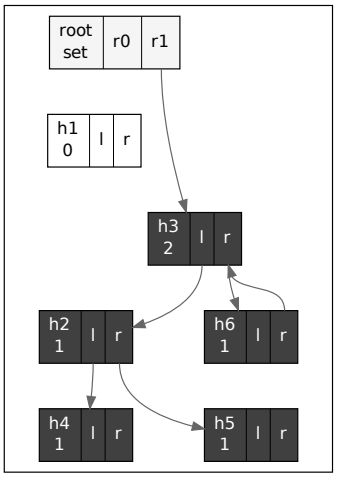
A continuación se presenta un ejemplo gráfico para facilitar la comprensión del algoritmo. Por simplicidad se asumen celdas de tamaño fijo con dos punteros, `left` (`l`) y `right` (`r`) y se muestra el contador de referencias abajo del nombre de cada celda. Se parte con una pequeña estructura ya construida y se muestra como opera el algoritmo al eliminar o cambiar una referencia (cambios en la conectividad del grafo). En un comienzo todas las celdas son accesibles desde el *root set* por lo tanto son todas parte del *live set*.

Se comienza por eliminar la referencia de `r0` a `h1`, que determina que `h1` se convirtió en *basura* (ver figura 3.4). Esto conduce al decremento del contador de `h2` y `h3` que permanecen en el *live set* ya que sus contadores siguen siendo mayores a 0 (ver figura 3.5 en la página siguiente).

Luego se cambia una referencia (en vez de eliminarse) realizándose la operación `update(h3.l, h5)`. Para esto primero se incrementa el contador de referencias de `h5` para evitar confundirlo accidentalmente con *basura* si se elimina alguna celda que apuntaba a ésta. Luego se procede a decrementar el contador de `h2` que queda en 0, transformándose en *basura* (ver figura 3.6 en la página siguiente).

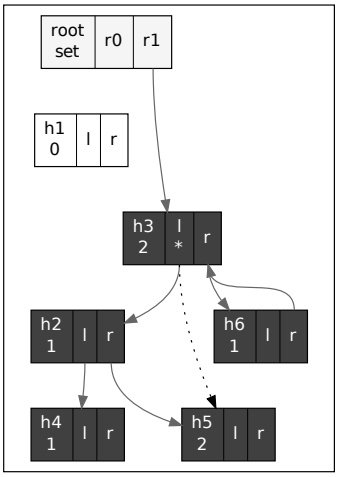


(a) Se decrementa el contador de `h2` pero no queda en 0 (permanece en el *live set*).

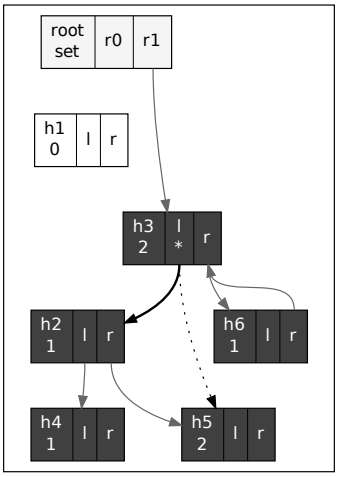


(b) El contador de `h3` tampoco queda en 0, sigue en el *live set*.

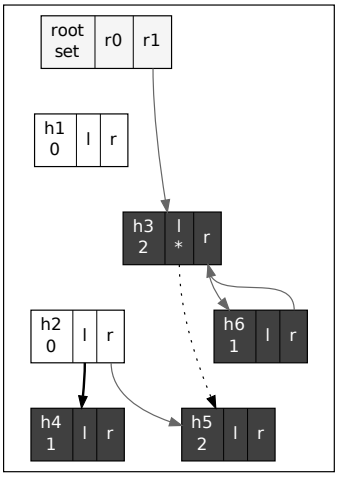
Figura 3.5: Eliminación de la referencia `r0` → `h1` (parte 2).



(a) Comienza `update(h3.l, h5)`, se incrementa el contador de `h5`.

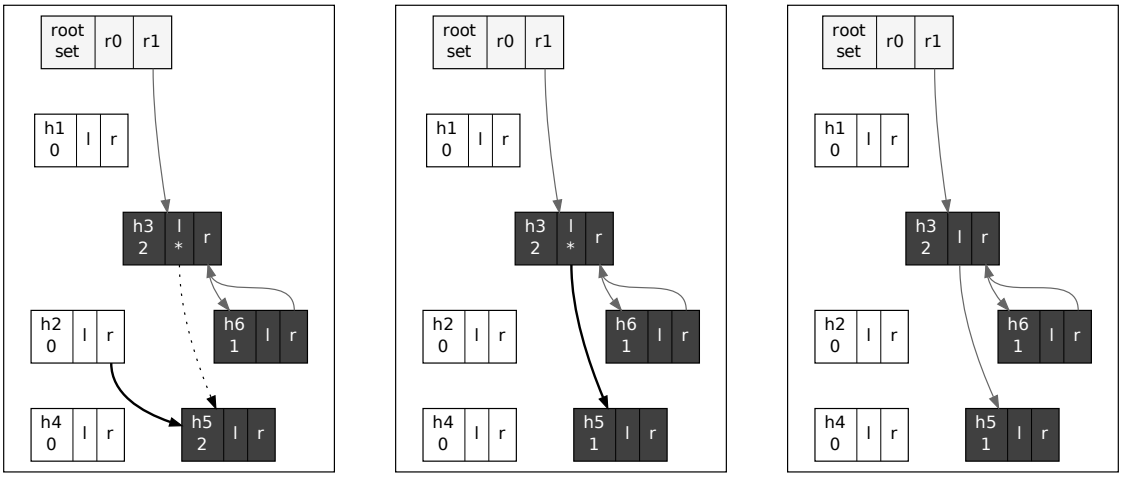


(b) Luego se procede a visitar la antigua referencia de `h3.l` (`h2`).



(c) Se decrementa el contador de `h2` y queda en 0 (pasa a ser *basura*). Se eliminan las referencias a las hijas.

Figura 3.6: Cambio en la referencia `h3.l` → `h2` a `h3.l` → `h5` (parte 1).



(a) Se decrementa el contador de `h4` quedando en 0, pasa a ser *basura*. Se continúa con `h5`.

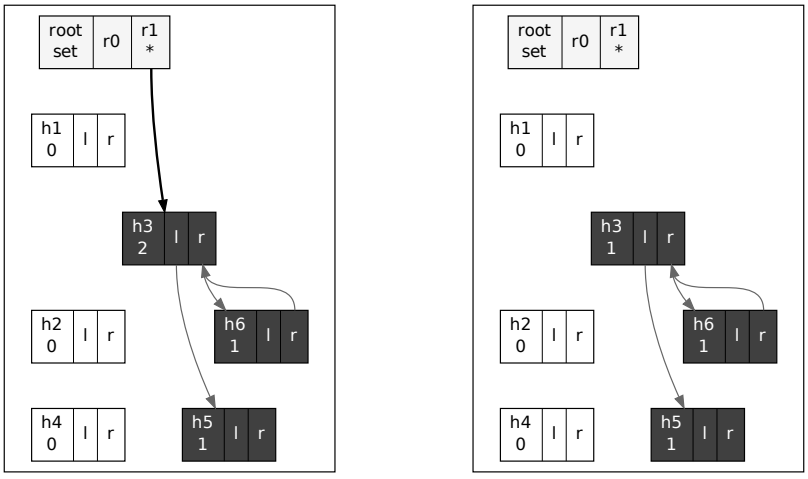
(b) Se decrementa el contador de `h5` pero sigue siendo mayor que 0.

(c) Se termina por actualizar la referencia de `h3.l` para que apunte a `h5`.

Figura 3.7: Cambio en la referencia `h3.l` → `h2` a `h3.l` → `h5` (parte 2).

Lo mismo pasa cuando se desciende a `h4`, pero al descender a `h5` y decrementar el contador, éste sigue siendo mayor que 0 (pues `h3` va a apuntar a `h5`) así que permanece en el *live set*. Finalmente se termina de actualizar la referencia `h3.l` para que apunte a `h5` (ver figura 3.7).

Finalmente se presenta lo que sucede cuando se elimina la última referencia a un ciclo (en este caso un ciclo simple de 2 celdas: `h3` y `h6`). Se elimina la única referencia externa al ciclo (`r1`), por lo que se visita la celda `h3` decrementando su contador de referencias, pero éste continúa siendo mayor que 0 porque la celda `h6` (parte del ciclo) la referencia. Por lo tanto el ciclo, y todas las celdas a las que apunta que no tienen otras referencias externas y por lo tanto deberían ser *basura* también (`h5`), no pueden ser recicladas y su memoria es perdida (ver figura 3.8).



(a) El ejecutarse `update(r1, null)` se visita la celda `h3`.

(b) Se decrementa el contador de `h3` pero sigue siendo mayor que 0 por el ciclo.

Figura 3.8: Eliminación de la referencia `r1` → `h3` (pérdida de memoria debido a un ciclo).

3.2.2 Marcado y barrido

Este algoritmo es el más parecido a la teoría sobre recolección de basura. Consiste en realizar la recolección en 2 fases: marcado y barrido. La primera fase consiste en el proceso de marcar el grafo de conectividad del *heap* para descubrir qué celdas son alcanzables desde el *root set*, tal y como se describió en *Recorrido del grafo de conectividad*.

Una vez marcadas todas las celdas, se sabe que las celdas *blancas* son *basura*, por lo tanto el paso que queda es el *barrido* de estas celdas, liberándolas. Esto se efectúa recorriendo todo el *heap*. Por lo tanto cada recolección es $O(|Heap|)$, a diferencia del conteo de referencia que dijimos que en el peor caso es $O(|Live\ set|)$. Sin embargo el conteo de referencias se ejecuta **cada vez que se actualiza una referencia** mientras que la recolección en el marcado y barrido se realiza típicamente solo cuando el *mutator* pide una celda pero no hay ninguna libre. Esto hace que la constante del conteo de referencias sea típicamente varios órdenes de magnitud mayores que en el marcado y barrido.

A continuación se presentan los servicios básicos de este algoritmo:

```
function new() is
   cell = alloc()
   if cell is null
      collect()
   cell = alloc()
   if cell is null
      throw out_of_memory
   return cell

function collect() is
   mark_phase()
   sweep_phase()

function sweep_phase() is
   foreach cell in heap
      if cell.marked
         cell.marked = false
      else
         free(cell)
```

El algoritmo `mark_sweep()` es exactamente igual al presentado en *Recorrido del grafo de conectividad*. Es preciso notar que la fase de barrido (`sweep_phase()`) debe tener una comunicación extra con el *low level allocator* para poder obtener todas las celdas de memoria que existen en el *heap*.

A diferencia del conteo de referencias, este algoritmo es *indirecto* y *no incremental*, ya que se realiza un recorrido de todo el *heap* de forma espaciada a través de la ejecución del programa. En general el *mutator* sufre pausas considerablemente mayores (en promedio) que con el conteo de referencias, lo que puede ser problemático para aplicaciones con requerimientos rígidos de tiempo, como aplicaciones *real-time*. Debido a la percepción de las pausas grandes, este tipo de colectores se conocen como *stop-the-world* (o *detener el mundo*).

Una ventaja fundamental sobre el conteo de referencias es la posibilidad de reclamar estructuras cíclicas sin consideraciones especiales. Podemos observar como esto es posible analizando el ejemplo en las figuras 3.2 y 3.3 en la página 27. Si se eliminaran las referencias $r0 \rightarrow h1$ y $h6 \rightarrow h2$, la fase de marcado consistiría solamente en marcar la celda $h6$, pues es la única alcanzable desde el *root set*. Todas las demás celdas permanecerían blancas y por lo tanto pueden ser liberadas sin inconvenientes en la fase de barrido, que recorre el *heap* linealmente.

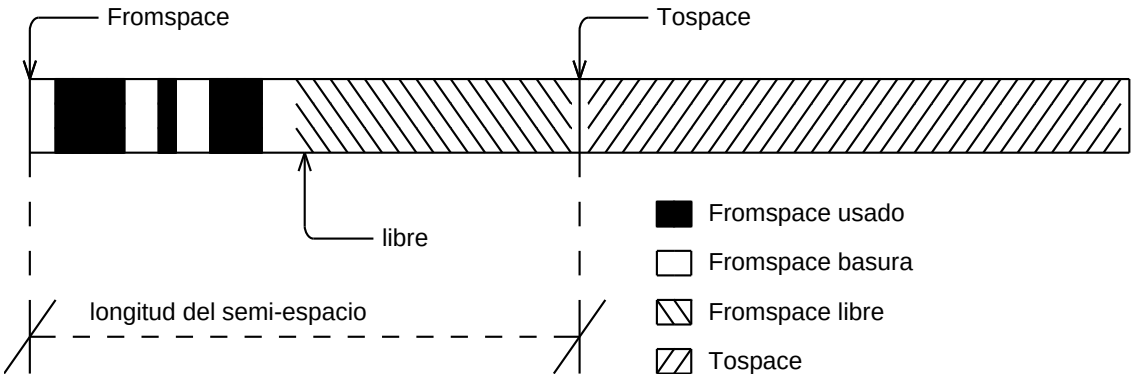


Figura 3.9: Estructura del *heap* de un recolector con copia de semi-espacios.

3.2.3 Copia de semi-espacio

Este algoritmo consiste en hacer una partición del *heap* en 2 mitades o *semi-espacios*, llamados usualmente *Fromspace* y *Tospace*. El primero se utiliza para asignar nuevas celdas de forma lineal, asumiendo un *heap* contiguo, incrementando un puntero (ver figura 3.9). Esto se conoce como *pointer bump allocation* y es, probablemente, la forma más eficiente de asignar memoria (tan eficiente como asignar memoria en el *stack*). Esto permite además evitar el problema de la *fragmentación* de memoria [9] que normalmente afectan a los otros algoritmos clásicos (o sus *low level allocators*).

La segunda mitad (*Tospace*) permanece inutilizada hasta que se agota el espacio en el *Fromspace*; en ese momento comienza el proceso de recolección de basura que consiste en recorrer el grafo de conectividad, copiando las celdas *vivas* del *Fromspace* al *Tospace* de manera contigua, como si estuvieran asignando por primera vez. Como la posición en memoria de las celdas cambia al ser movidas, es necesario actualizar la dirección de memoria de todas las celdas *vivas*. Para esto se almacena una dirección de memoria de re-dirección, *forwarding address*, en las celdas que mueven. La *forwarding address* sirve a su vez de marca, para no recorrer una celda dos veces (como se explica en *Recorrido del grafo de conectividad*). Cuando se encuentra una celda que ya fue movida, simplemente se actualiza la referencia por la cual se llegó a esa celda para que apunte a la nueva dirección, almacenada en la *forwarding address*. Una vez finalizado este proceso, el *Fromspace* y *Tospace* invierten roles y se prosigue de la misma manera (todo lo que quedó en el viejo *Fromspace* es *basura* por definición, por lo que se convierte el *Tospace*).

A continuación se presenta una implementación sencilla de los servicios provistos por este tipo de recolectores. Cabe destacar que este tipo de recolectores deben estar íntimamente relacionados con el *low level allocator*, ya que la organización del *heap* y la forma de asignar memoria es parte fundamental de este algoritmo. Se asume que ya hay dos áreas de memoria del mismo tamaño destinadas al *Fromspace* y *Tospace*, y la existencia de 4 variables globales: `fromspace` (que apunta a la base del *Fromspace*), `tospace` (que apunta a la base del *Tospace*), `spacesize` (que contiene el tamaño de un semi-espacio) y `free` (que apunta al lugar del *Fromspace* donde comienza la memoria libre). También vale aclarar que este algoritmo soporta inherentemente celdas de tamaño variable, por lo que los servicios `alloc()` y `new()` [10] reciben como parámetro el tamaño de la celda a asignar:

```
function alloc(size) is
   if free + size > fromspace + spacesize
```

[9] La *fragmentación* de memoria sucede cuando se asignan objetos de distintos tamaños y luego libera alguno intermedio, produciendo *huecos*. Estos *huecos* quedan inutilizables hasta que se quiera asignar un nuevo objeto de tamaño igual al *hueco* (o menor). Si esto no sucede y se acumulan muchos *huecos* se dice que la memoria está *fragmentada*.

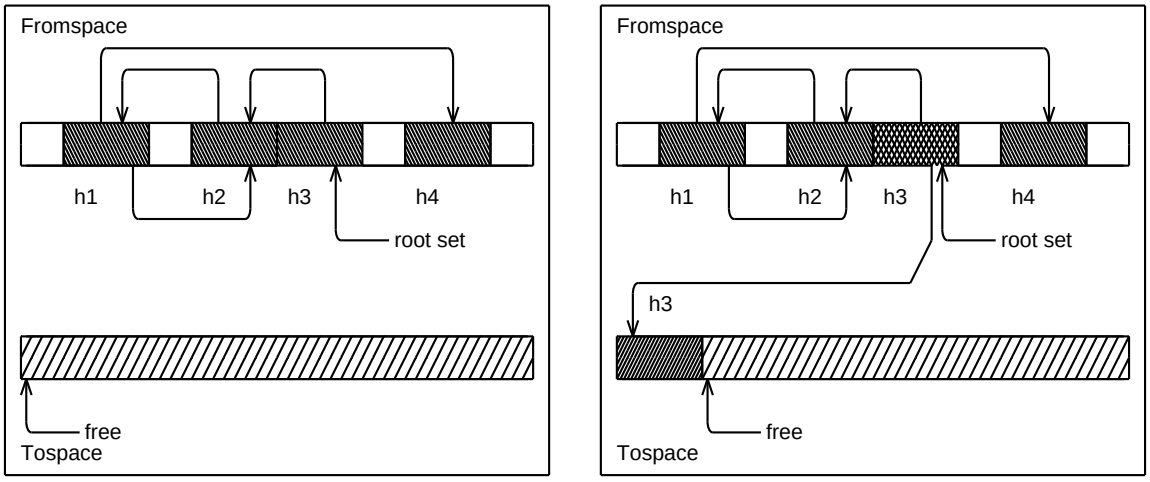
[10] Notar que `new()` es igual que en el marcado y barrido con la salvedad de que en este caso toma como parámetro el tamaño de la celda.

```
      return null
   else
      cell = free
      free = free + size
      return cell

function new(size) is
   cell = alloc(size)
   if cell is null
      collect()
   cell = alloc(size)
   if cell is null
      throw out_of_memory
   return cell

function collect() is
   free = tospace
   foreach r in root_set
      r = copy(r)
   fromspace, tospace = tospace, fromspace

function copy(cell) is
   if cell.forwarding_address is null
      cell.forwarding_address = free
      cell.copy_to(free)
      free = free + cell.size
      foreach child in cell
         child = copy(child)
      return cell.forwarding_address
   else
      return cell.forwarding_address
```

Esta técnica tiene nombres variados en inglés: *semi-space*, *two-space* o simplemente *copying collector*. En este documento se denomina "copia de semi-espacio" porque los otros nombres son demasiado generales y pueden describir, por ejemplo, algoritmos donde no hay copia de celdas o donde no hay 2 semi-espacios (como se verá en *Estado del arte*).

Al igual que el *Marcado y barrido* este algoritmo es *indirecto*, *no incremental* y *stop-the-world*. Las diferencias con los esquemas vistos hasta ahora son evidentes. La principal ventaja sobre el marcado y barrido (que requiere una pasada sobre el *live set*, el marcado, y otra sobre el *heap* entero, el barrido) es que este método requiere una sola pasada y sobre las celdas vivas del *heap* solamente. La principal desventaja es copia memoria, lo que puede ser particularmente costoso, además de requerir, como mínimo, el doble de memoria de lo que el *mutator* realmente necesita. Esto puede traer en particular problemas con la memoria virtual y el caché, por la pobre localidad de referencia.

Por lo tanto los recolectores de este tipo pueden ser convenientes por sobre el marcado y barrido cuando se espera que el *live set* sea muy pequeño luego de una recolección. En estos casos el trabajo realizado por este tipo de recolectores puede ser considerablemente menor que el del marcado y barrido. Y por el contrario, si el *working set* es pequeño, al ser *compactado* en memoria puede mejorar la localidad de referencia (si el *working set* es grande se corre el riesgo de que la localidad de referencia empeore al moverse las celdas).



(a) Estructura inicial del *heap*. El *Fromspace* está complete y se inicial la recolección.

(b) Se sigue la referencia del *root set*, copiando `h3` al *Tospace* y dejando una *forwarding address*.

Figura 3.10: Ejemplo de recolección con copia de semi-espacios (parte 1).

Ejemplo

A continuación se presenta un sencillo ejemplo del algoritmo. Se parte de una estructura simple con 4 celdas en el *Fromspace* (que incluye un pequeño ciclo para mostrar que este algoritmo tampoco tiene inconvenientes para recolectarlos). Asumimos que ya no queda lugar en el *Fromspace* por lo que comienza la ejecución de `collect()`. Se comienza por el *root set* que apunta a `h3`, por lo tanto ésta es movida al *Tospace* primero, dejando una *forwarding address* a la nueva ubicación (ver figura 3.10).

A continuación se copian las *hijas* de `h3`, en este caso sólo `h2`, que se ubica en el *Tospace* a continuación de `h3`, dejando nuevamente su `forwarding address` en la celda original. Al proceder recursivamente, se procede a copiar `h1` al *Tospace*, dejando una vez más la *forwarding address* en la celda original y procediendo con las hijas. Aquí podemos observar que al seguirse la referencia $h1 \rightarrow h2$, como `h2` ya había sido visitada, solamente se actualiza la referencia apuntando a la nueva ubicación de `h2` pero no se vuelve a copiar la celda (ver figura 3.11 en la página siguiente).

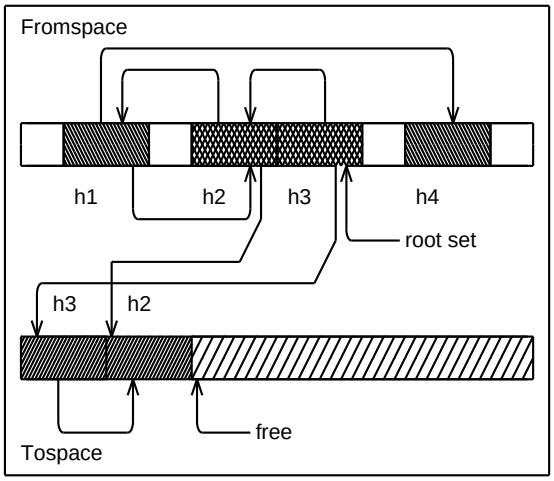
Se termina de copiar recursivamente las hijas de `h1` al copiar `h4` que resulta la última celda (sin hijas). Finalmente se invierten los roles de los semi-espacios y se actualiza la referencia del *root set* para que apunte a la nueva ubicación de `h3`, como se muestra en la figura 3.12 en la página siguiente.

3.3 Estado del arte

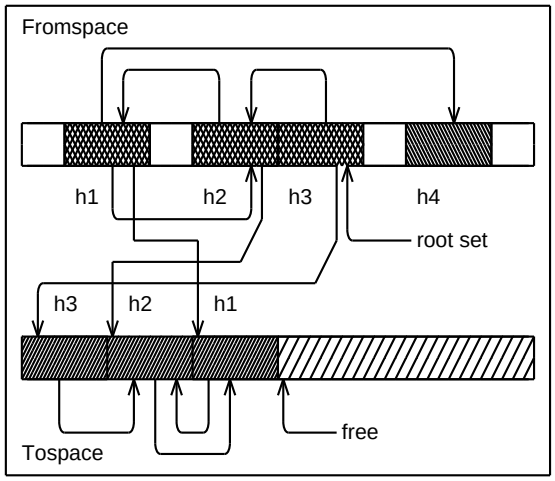
La manera en que la investigación sobre recolección de basura ha crecido es realmente sorprendente. Hay, al menos, 2995 publicaciones sobre recolección de basura registradas al momento de escribir este documento [GCBIB]. Esto hace que el análisis del estado del arte sea particularmente complejo y laborioso.

Analizar todas las publicaciones existentes es algo excede los objetivos de este trabajo, por lo tanto se analizó solo una porción significativa, utilizando como punto de partida a [JOLI96].

De este análisis se observó que la gran mayoría de los algoritmos son combinaciones de diferentes características básicas; por lo tanto se intentó aislar estas características que son utilizadas como bloques de construcción para algoritmos complejos. Ésta tampoco resultó ser una tarea sencilla debido a que

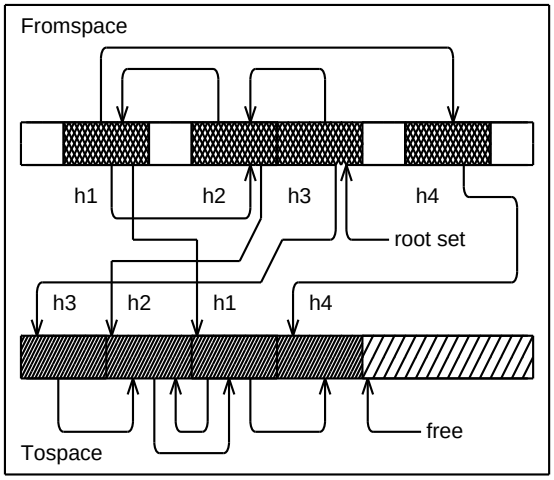


(a) Se sigue $h3 \rightarrow h2$, copiando `h2` al *Tospace* y dejando una *forwarding address*.

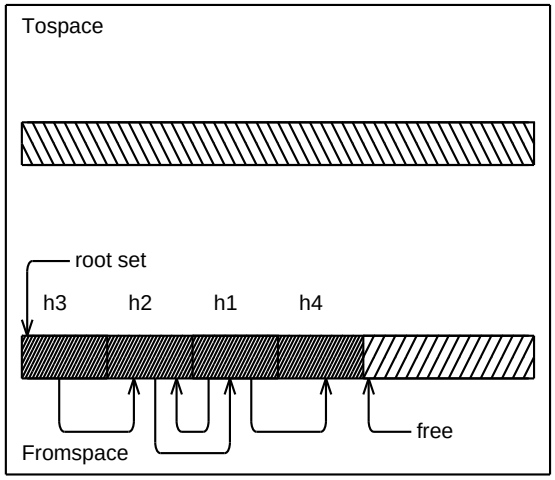


(b) Se sigue $h2 \rightarrow h1$, copiando `h1`. Luego $h1 \rightarrow h2$ pero `h2` no se copia, sólo se actualiza la referencia con la *forwarding address*.

Figura 3.11: Ejemplo de recolección con copia de semi-espacios (parte 2).



(a) Se sigue $h1 \rightarrow h4$ copiando *h4'* al *Tospace* y dejando una *forwarding address*.



(b) Se finaliza la recolección, se intercambian los roles de los semi-espacios y se actualiza la referencia del *root set*.

Figura 3.12: Ejemplo de recolección con copia de semi-espacios (parte 3).

muchos de estos bloques de construcción básicos están interrelacionados y encontrar una división clara para obtener características realmente atómicas no es trivial.

La construcción de recolectores más complejos se ve alimentada también por la existencia de recolectores *híbridos*; es decir, recolectores que utilizan más de un algoritmo dependiendo de alguna característica de la memoria a administrar. No es poco común observar recolectores que utilizan un algoritmo diferente para celdas que sobreviven varias recolecciones que para las que mueren rápidamente, o que usan diferentes algoritmos para objetos pequeños y grandes, o que se comporten de forma conservativa para ciertas celdas y sean precisos para otras.

De todas estas combinaciones resulta el escenario tan fértil para la investigación sobre recolección de basura.

A continuación se presentan las principales clases de algoritmos y características básicas encontradas durante la investigación del estado del arte. La separación de clases y aislamiento de características no es siempre trivial, ya que hay ciertas clases de recolectores que están interrelacionadas (o ciertas características pueden estar presentes sólo en recolectores de una clase en particular).

3.3.1 Recolección directa / indirecta

Generalmente se llama recolección **directa** a aquella en la cual el compilador o lenguaje instrumenta al *mutator* de forma tal que la información sobre el grafo de conectividad se mantenga activamente cada vez que hay un cambio en él. Normalmente se utiliza un contador de referencia en cada celda para este propósito, permitiendo almacenar en todo momento la cantidad de nodos que apuntan a ésta (ver *Conteo de referencias*). Esto permite reclamar una celda instantáneamente cuando el *mutator* deja de hacer referencia a ella. Este tipo de recolectores son inherentemente *incrementales*.

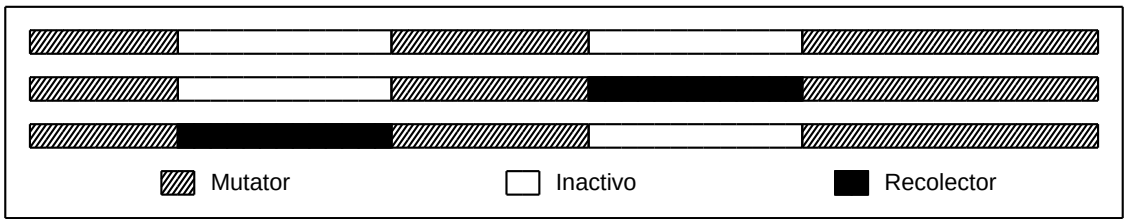
Por el contrario, los recolectores **indirectos** normalmente no interfieren con el *mutator* en cada actualización del grafo de conectividad (exceptuando algunos *recolectores incrementales* que a veces necesitan instrumentar el *mutator* pero no para mantener el estado del grafo de conectividad completo). La recolección se dispara usualmente cuando el *mutator* requiere asignar memoria pero no hay más memoria libre conocida disponible y el recolector se encarga de generar la información de conectividad desde cero para determinar qué celdas son *basura*. Prácticamente todos los recolectores menos el *conteo de referencias* están dentro de esta categoría (como por ejemplo, el *marcado y barrido* y *copia de semi-espacio*).

Otros ejemplos de recolectores modernos *directos* son el recolector de basura de Python [NAS00] y [LINS05] (aunque ambos tiene un algoritmo *indirecto* para recuperar ciclos).

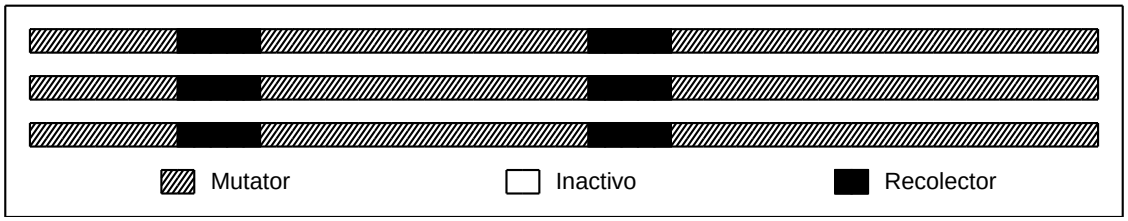
3.3.2 Recolección incremental

Recolección incremental es aquella que se realiza de forma intercalada con el *mutator*. En general el propósito es disminuir el tiempo de las pausas causadas por el recolector (aunque generalmente el resultado es un mayor costo total de recolección en términos de tiempo).

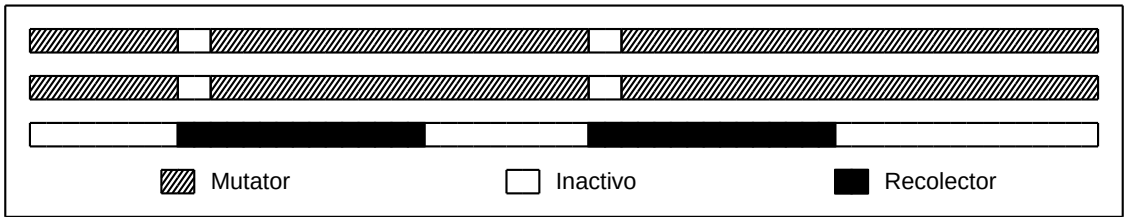
De los algoritmos clásicos el único que es incremental en su forma más básica es el *conteo de referencias*. Otros recolectores pueden hacerse incrementales de diversas maneras, pero en general consta de hacer parte del trabajo de escanear el grafo de conectividad cada vez que el *mutator* asigna memoria. En general para hacer esto es también necesario instrumentar al *mutator* de forma tal que informe al recolector cada vez que cambia el grafo de conectividad, para que éste pueda marcar al sub-grafo afectado por el cambio como *desactualizado* y así re-escanearlo nuevamente en la próxima iteración. Para realizar esto en recolectores *indirectos* se utiliza la *abstracción tricolor*; cuando el *mutator* cambia una referencia, se marca *gris* la celda que la contiene, de modo que el recolector vuelva a visitarla.



(a) *Stop-the-world*.



(b) Paralelo.



(c) Concurrente.

Figura 3.13: Distintos tipos de recolectores según el comportamiento en ambientes multi-hilo.

En general el rendimiento de los recolectores incrementales disminuye considerablemente cuando el *mutator* actualiza muy seguido el grafo de conectividad, porque debe re-escanear sub-grafos que ya había escaneado una y otra vez. A esto se debe también que en general el tiempo de procesamiento total de un recolector incremental sea mayor que uno no incremental, aunque el tiempo de pausa de una recolección sea menor.

Ejemplos de recolectores que se encuentran dentro de esta categoría son [BOEH91], [LINS05],

3.3.3 Recolección concurrente / paralela / *stop-the-world*

Los recolectores concurrentes son aquellos que pueden correr en paralelo con el *mutator*. Por el contrario, aquellos que pausan el *mutator* para realizar la recolección son usualmente denominados *stop-the-world* (*detener el mundo*), haciendo referencia a que pausan todos los hilos del *mutator* para poder escanear el grafo de conectividad de forma consistente. Hay una tercera clase de colectores que si bien son *stop-the-world*, utilizan todos los hilos disponibles para realizar la recolección (ver figura 3.13).

Para lograr que un recolector sea concurrente generalmente el mecanismo es similar al necesario para hacer un *recolector incremental*: hay que instrumentar al *mutator* para que informe al recolector cuando se realiza algún cambio en el grafo de conectividad, de forma tal que pueda volver a escanear el sub-grafo afectado por el cambio.

Esto también trae como consecuencia el incremento en el tiempo total que consume el recolector, debido a la necesidad de re-escanear sub-grafos que han sido modificados, además de la sincronización necesaria entre *mutator* y recolector.

¿Cuál es la idea entonces de un recolector concurrente? Una vez más, al igual que los recolectores incrementales, el principal objetivo es disminuir las largas pausas provocadas por la recolección de basura. Sin embargo, este tipo de algoritmos además permite hacer un mejor aprovechamiento de las arquitecturas *multi-core* [11] que cada vez son más comunes, ya que el *mutator* y el recolector pueden estar corriendo realmente en paralelo, cada uno en un procesador distinto. Algunos recolectores van más allá y permiten incluso paralelizar la recolección de basura en varios hilos ([HUEL98], [LINS05]). Otros ejemplos de recolectores concurrentes (aunque no ofrece paralelización del procesamiento del recolector en varios hilos) son [BOEH91], [RODR97].

Todos los *algoritmos clásicos* que se han citado son del tipo *stop-the-world*.

3.3.4 Lista de libres / *pointer bump allocation*

Esta clasificación se refiere principalmente a la forma en que se organiza el *heap*, íntimamente relacionado al *low level allocator*. Si bien se ha dicho que en este trabajo no se prestará particular atención a este aspecto, en ciertos algoritmos es tan relevante que tampoco es sensato pasarlo por alto completamente.

En términos generales, hay dos formas fundamentales de organizar el *heap*, manteniendo una lista de libres o realizando *pointer bump allocation*, como se explicó en *Copia de semi-espacio*. La primera forma consiste, a grandes rasgos, en separar el *heap* en celdas (que pueden agruparse según tamaño) y enlazarlas en una lista de libres. Al solicitarse una nueva celda simplemente se la desenlaza de la lista de libres. Por otro lado, cuando el recolector detecta una celda *muerta*, la vuelve a enlazar en la lista de libres. Este es un esquema simple pero con limitaciones, entre las principales, el costo de asignar puede ser alto si hay muchos tamaños distintos de celda y soportar tamaño de celda variable puede ser complejo o acarrear muchas otras ineficiencias. El *marcado y barrido* en general usa este esquema, al igual que el *conteo de referencias*.

Otro forma de organizar el *heap* es utilizándolo como una especie de *stack* en el cual para asignar simplemente se incrementa un puntero. Este esquema es simple y eficiente, si el recolector puede mover celdas (ver *Movimiento de celdas*); de otra manera asignar puede ser muy costoso si hay que buscar un *hueco* en el heap (es decir, deja de reducirse a incrementar un puntero). El clásico ejemplo de esta familia es el algoritmo visto en *Copia de semi-espacio*.

Sin embargo, entre estos dos extremos, hay todo tipo de híbridos. Existen recolectores basados en *regiones*, que se encuentran en un punto intermedio. Dentro de una región se utiliza un esquema de *pointer bump allocation* pero las regiones en sí se administran como una lista de libres (como por ejemplo [BLAC08]). Otra variación (más común) de este esquema son los *two level allocators* que asignan páginas completas (similar a las regiones) y dentro de cada página se asignan las celdas. Ambas, páginas y celdas, se administran como listas de libres (ejemplos que utilizan este esquema son [BOEHWD] y el *recolector actual de D*).

3.3.5 Movimiento de celdas

Otra característica muy importante del recolector de basura es si mueve las celdas o no. En general el movimiento de celdas viene de la mano del esquema de *pointer bump allocation*, ya que *compacta* todas las celdas *vivas* al comienzo del *heap* luego de una recolección, permitiendo este esquema para asignar nuevas celdas, pero puede utilizarse en esquemas híbridos como recolectores basados en *regiones* (por ejemplo [BLAC08]).

[11] Una arquitectura *multi-core* es aquella que combina dos o más núcleos (*cores*) independientes que trabajan a la misma frecuencia, pero dentro de un solo circuito integrado o procesador.

Además los recolectores con movimiento de celdas deben ser *precisos*, porque es necesario tener la información completa de los tipos para saber cuando actualizar los punteros (de otra manera se podría escribir un dato de una celda que no era un puntero). Para que un recolector pueda mover celdas, aunque sea parcialmente, en recolectores *semi-precisos* se utiliza un método conocido como *pinning* (que significa algo como *pinchar con un alfiler*); una celda es *pinned* (*pinchada*) cuando hay alguna referencia no-precisa a ella, y por lo tanto no puede ser movida (porque no se puede estar seguro si es posible actualizar dicha referencia).

La ventaja principal de los colectores con movimiento es la posibilidad de utilizar *pointer bump allocation* y que es sencillo implementar recolectores *generacionales* sobre estos.

De los algoritmos clásicos sólo la *copia de semi-espacios* mueve celdas, el *conteo de referencias* y *marcado y barrido* no lo hacen. Además hay otro algoritmo bastante básico que mueve celdas, el **marcado y compactado**. Éste no tiene 2 semi-espacios, directamente mueve las celdas compactándolas al comienzo del *heap*. El algoritmo es un poco más complejo que la *copia de semi-espacios* pero suele proveer una mayor localidad de referencia y no *desperdicia* un semi-espacio que está inutilizado salvo en el momento de la recolección. Por ejemplo para Mono, que antes usaba un recolector conservativo sin movimiento ([BOEHWD]) se está implementando un recolector de este tipo [MOLAWE] [MOLA06].

3.3.6 Recolectores conservativos versus precisos

Los recolectores *conservativos* son aquellos que tienen la capacidad de poder lidiar con un *root set* o celdas que no tengan información de tipos asociada. Esto significa que el recolector no sabe donde hay punteros (o referencias) en una celda o raiz dada. Es decir, una ubicación particular puede ser un puntero o no. Esto trae una variada cantidad de problemas, como retención de celdas que en realidad son *basura* simplemente porque hay algún dato que coincide con la dirección de memoria en la que está almacenada esa celda *basura* [12]. Además los recolectores puramente conservativos no puede mover celdas (ver *Movimiento de celdas*), dado que no pueden actualizar los supuestos punteros por la posibilidad de que sean *falsos positivos*.

Sin embargo hay ciertas circunstancias que hacen que no quede más remedio que el recolector sea conservativo, por ejemplo cuando se utiliza un recolector de basura para un lenguaje que no ha sido pensado para tener uno (como C o C++).

Por el contrario, los recolectores que poseen a su disposición información completa sobre el tipo de la celda, y por ende información sobre cuales de sus campos son realmente punteros, son denominados *precisos*. Estos recolectores no están sujetos a estas limitaciones y por lo tanto son potencialmente más eficientes en cuanto a tiempo y espacio. Los lenguajes que fueron diseñados para tener un recolector de basura (y en especial aquellos que son de relativo alto nivel) en general disponen de recolectores precisos.

Hay casos donde se posee información de tipos para algunas celdas solamente, o más comúnmente se posee información de tipos de celdas que se encuentran en el *heap* pero no para el *stack* y registros (por ejemplo [MOLA06]). En estos casos se puede adoptar un esquema híbrido y tratar algunas referencias de forma conservativa y otras de forma precisa, de manera de mitigar, aunque sea de forma parcial, los efectos adversos de los recolectores conservativos. Estos recolectores son conocidos como *semi-precisos*. Los recolectores semi-precisos pueden mover celdas si utilizan un mecanismo de *pinning* (ver *Movimiento de celdas*).

El ejemplo de recolector conservativo por excelencia es el recolector Boehm-Demers-Wiser ([BOEH88], [BOEH91], [BOEH93], [BOEHWD]) aunque puede comportarse de forma semi-precisa si el usuario se encarga de darle la información de tipos (en cuyo caso el recolector deja de ser transparente para el

[12] Esto es lo que se conoce como un *falso positivo*, algo que aparenta ser un puntero pero en realidad no lo es.

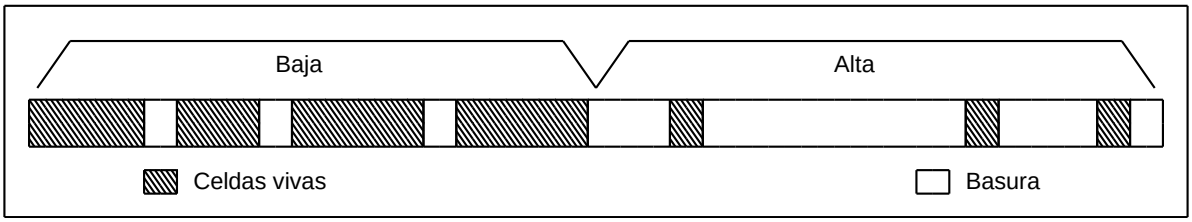


Figura 3.14: Concentración de basura en distintas particiones del *heap*.

usuario). Otros ejemplos de recolectores con cierto grado de precisión son el *recolector actual de D* y [BLAC08].

3.3.7 Recolección por particiones / generacional

Otra forma de reducir la cantidad de pausas y la cantidad de trabajo realizado por el recolector en general es dividiendo el *heap* en particiones de manera tal de recolectar solo las partes donde más probabilidad de encontrar *basura* haya.

Entonces, si el recolector tiene algún mecanismo para identificar zonas de alta concentración de *basura* puede hacer la recolección solo en ese área donde el trabajo va a ser mejor recompensado (ver figura 3.14).

Sin embargo encontrar zonas de alta concentración no es trivial. La forma más divulgada de encontrar estas zonas es dividiendo el *heap* en una partición utilizada para almacenar celdas *jóvenes* y otra para celdas *viejas*. Una celda *vieja* es aquella que ha *sobrevivido* una cantidad *N* de recolecciones, mientras que el resto se consideran *jóvenes* (las celdas *nacen* jóvenes). Los recolectores que utilizan este tipo de partición son ampliamente conocido como recolectores **generacionales**. La *hipótesis generacional* dice que el área de celdas jóvenes tiene una mayor probabilidad de ser un área de alta concentración de basura [JOLI96]. Basándose en esto, los recolectores generacionales primero intentan recuperar espacio del área de celdas jóvenes y luego, de ser necesario, del área de celdas viejas. Es posible tener varias generaciones e ir subiendo de generación a generación a medida que es necesario. Sin embargo en general no se obtienen buenos resultados una vez que se superan las 3 particiones. La complejidad que trae este método es que para recolectar la generación joven es necesario tomar las referencias de la generación vieja a la joven como parte del *root set* (de otra forma podrían tomarse celdas como *basura* que todavía son utilizadas por las celdas viejas). Revisar toda la generación vieja no es una opción porque sería prácticamente lo mismo que realizar una recolección del *heap* completo. La solución está entonces, una vez más, en instrumentar el *mutator* para que avise al recolector cuando cambia una referencia de la generación vieja a la joven (no es necesario vigilar las referencias en sentido inverso ya que cuando se recolecta la generación vieja se hace una recolección del *heap* completo).

Sin embargo, a pesar de ser este el esquema más difundido para dividir el *heap* y realizar una recolección parcial sobre un área de alta concentración de basura, no es la única. Otros recolectores proponen hacer un análisis estático del código revisando la conectividad entre los objetos según sus tipos (esto es posible solo en lenguajes con *tipado* estático), de manera tal de separar en distintas áreas grupos de tipos que no pueden tener referencias entre sí [HIRZ03]. Este análisis hace que sea innecesario instrumentar el *mutator* para reportar al recolector cambios de referencias inter-particiones, sencillamente porque queda demostrado que no existe dicho tipo de referencias. Esto quita una de las principales ineficiencias y complejidades del esquema generacional.

CAPÍTULO 4

Recolección de basura en D

D propone un nuevo desafío en cuanto al diseño de un recolector de basura, debido a la gran cantidad de características que tiene y paradigmas que soporta.

D ya cuenta con un recolector que hace lo necesario para funcionar de forma aceptable, pero su diseño e implementación son relativamente sencillos comparados con el *estado del arte* de la recolección de basura en general. Además la implementación actual presenta una serie de problemas que se evidencia en las quejas que regularmente la comunidad de usuarios de D menciona en el grupo de noticias.

En esta sección se analizarán las necesidades particulares de D con respecto a la recolección de basura. También se analiza el diseño e implementación del recolector actual, presentando sus fortalezas y debilidades. Finalmente se analiza la viabilidad de los diferentes algoritmos vistos en *Estado del arte*.

4.1 Características y necesidades particulares de D

En esta sección se hará un recorrido por las características y necesidades particulares que tiene D como lenguaje con respecto a la recolección de basura.

4.1.1 Programación de bajo nivel (*system programming*)

Sin dudas las características de D que lo hacen más complejo a la hora de implementar un recolector de basura son sus capacidades de programación de bajo nivel (ver *Programación de bajo nivel (system programming)*).

Al proveer acceso a *assembly*, permitir estructuras de tipo *union* y ser compatible con C/C++, el recolector de basura tiene muchas restricciones. Por ejemplo debe tratar de forma conservativa los registros y el *stack*, ya que es la única forma de interactuar de forma segura con C/C++ y *assembly*.

Además debe poder interactuar con manejo de memoria explícito, ya sea omitiendo por completo el *heap* del recolector o liberando explícitamente memoria de éste. Esta característica es muy inusual en un recolector, a excepción de recolectores conservativos diseñados para C/C++ que tienen las mismas (o más) limitaciones.

La posibilidad de controlar la alineación de memoria es otra complicación sobre el recolector de basura, incluso aunque éste sea conservativo. Dado que tratar la memoria de forma conservativa byte a byte sería impracticable (tanto por la cantidad de *falsos positivos* que esto provocaría como por el impacto en el rendimiento por el exceso de posibles punteros a revisar, además de lo ineficiente que es operar sobre

memoria no alineada), en general el recolector asume que el usuario nunca va a tener la única referencia a un objeto en una estructura no alineada al tamaño de palabra.

4.1.2 Programación de alto nivel

Las características de programación de alto nivel también impone dificultades o restricciones al recolector de basura (ver *Programación de alto nivel*). Por ejemplo el soporte de rebanado (*slicing*) de arreglos hace que el recolector deba soportar punteros *interiores* [1] (esto también es necesario porque en general en D o en cualquier lenguaje de bajo nivel se puede tener un puntero a cualquier parte de una celda).

Los arreglos dinámicos y asociativos en particular dependen fuertemente del recolector de basura, en particular cuando se agregan elementos (o se concatenan dos arreglos).

Dado que los *strings* son arreglos dinámicos y que el lenguaje provee un buen soporte de arreglos dinámicos y asociativos y *slicing*, es de esperarse que el recolector deba comportarse de forma correcta y eficiente ante las operaciones más típicas de estas estructuras que dependan de él.

4.1.3 Información de tipos

Hasta aquí D comparte todas las restricciones con respecto a la recolección de basura con los lenguajes de bajo nivel que no tienen ningún soporte para recolectar basura. Sin embargo, a diferencia de éstos, D tiene una información de tipos más rica. Al momento de asignar memoria D puede proveer cierta información sobre el objeto a asignar (como si puede contener punteros o no) que puede ser utilizada por el recolector para realizar una recolección más precisa (ver *Recolectores conservativos versus precisos*).

En general esta información no es suficiente como para implementar un recolector completamente preciso (no al menos sin agregar un mejor soporte de reflexión al lenguaje) pero puede ser de ayuda considerable para el recolector.

4.1.4 Orientación a objetos y finalización

D soporta el paradigma de orientación a objetos, donde es común permitir que un objeto, al ser destruido, realice alguna tarea de finalización (a través de una función miembro llamada *destructor*, o `~this()` en D). Esto significa que el recolector, al encontrar que no hay más referencias a un objeto, debe ejecutar el destructor.

La especificación dice [DWDE]:

> The garbage collector is not guaranteed to run the destructor for all unreferenced objects. Furthermore, the order in which the garbage collector calls destructors for unreference objects is not specified. This means that when the garbage collector calls a destructor for an object of a class that has members that are references to garbage collected objects, those references may no longer be valid. This means that destructors cannot reference sub objects.

Afortunadamente el orden de finalización no está definido, ya que esto sería extremadamente difícil de proveer por un recolector (si no imposible). Esto significa que si bien se ejecutan los destructores de los objetos que dejan de ser alcanzables desde el *root set*, no se define en que orden se hace, y por lo tanto un objeto no puede acceder a sus atributos que sean referencias a otros objetos en un destructor.

[1] Los punteros *interiores* son aquellos que en vez de apuntar al inicio de una celda, apuntan a una dirección arbitraria dentro de ella. Esto no es posible en muchos lenguajes de programación, como por ejemplo Java, lo que simplifica la recolección de basura.

Esta restricción en realidad se ve relaja con el soporte de *RAII*. Si se utiliza la palabra clave `scope` al crear una serie de objetos, estos serán destruidos determinísticamente al finalizar el *scope* actual en el orden inverso al que fueron creados y, por lo tanto, un usuario podría hacer uso de los atributos que sean referencias a otros objetos creados con `scope` si el orden en que fueron creados (y por lo tanto en que serán destruidos) se lo permite.

Sin embargo no hay forma actualmente de saber dentro de un destructor si éste fue llamado determinísticamente o no, por lo tanto es virtualmente imposible hacer uso de esta distinción, a menos que una clase sea declarada para ser creada solamente utilizando la palabra reservada `scope`.

Cabe aclarar que, estrictamente hablando y según la especificación de D, el recolector no debe garantizar la finalización de objetos bajo ninguna circunstancia, es decir, el recolector podría no llamar a ningún destructor. Sin embargo esto es probablemente una vaguedad en la redacción y dadas las garantías que provee la implementación actual la comunidad de D cuenta con ellas.

4.2 Recolector de basura actual de D

Como paso básico fundamental para poder mejorar el recolector de basura de D, primero hay que entender la implementación actual, de forma de conocer sus puntos fuertes, problemas y limitaciones.

Como se mencionó en la sección *El lenguaje de programación D*, hay dos bibliotecas base para soportar el lenguaje (*runtimes*): Phobos y Tango. La primera es la biblioteca estándar de D, la segunda un proyecto más abierto y dinámico que surgió como alternativa a Phobos dado que estaba muy descuidada y que era muy difícil impulsar cambios en ella. Ahora Phobos tiene el agravante de estar *congelada* en su versión 1 (solo se realizan correcciones de errores).

Dado que Tango está mejor organizada, su desarrollo es más abierto (aceptan cambios y mejoras) y que hay una mayor disponibilidad de programas y bibliotecas escritos para Tango, en este trabajo se decide tomar esta biblioteca *runtime* como base para el análisis y mejoras propuestas, a pesar de ser Phobos la estándar. De todas formas el recolector de basura de Tango es prácticamente el mismo que el de Phobos, por lo tanto éste análisis en particular es válido para cualquiera de las dos.

El recolector actual es un recolector *indirecto*, *no incremental* que realiza un *marcado y barrido* relativamente básico. A diferencia del algoritmo clásico presentado éste realiza un marcado no recursivo. La fase de marcado es *stop-the-world* mientras que la fase de barrido corre en paralelo con el *mutator*, excepto el hilo que disparó la recolección que es quien efectúa el barrido (además los hilos que intenten asignar nueva memoria o interactuar con el recolector de cualquier otra forma se bloquean hasta que la fase de barrido concluya). El marcado es casi totalmente *conservativo*; si bien posee alguna información de tipos (distingue entre celdas que pueden tener punteros y celdas que definitivamente no los tienen, pero no dispone de información sobre qué campos de las celdas son punteros y cuales no). Además no tiene soporte alguno de *recolección particionada*.

Si bien el recolector es bastante básico, posee una *organización de memoria* relativamente moderna (utiliza una *lista de libres* con un *two level allocator*) y algunas optimizaciones particulares para amortiguar casos patológicos.

4.2.1 Organización del *heap*

La memoria del *heap* está organizada en *pools*. Un *pool* es una región de *páginas* contiguas. Una página es, en general, la unidad mínima de memoria que maneja un sistema operativo con soporte de memoria virtual. Cada página dentro de un *pool* sirve a su vez como contenedora de bloques (llamados *bin* en la *implementación*) de tamaño fijo. Todos los bloques pertenecientes a la misma página tienen el mismo

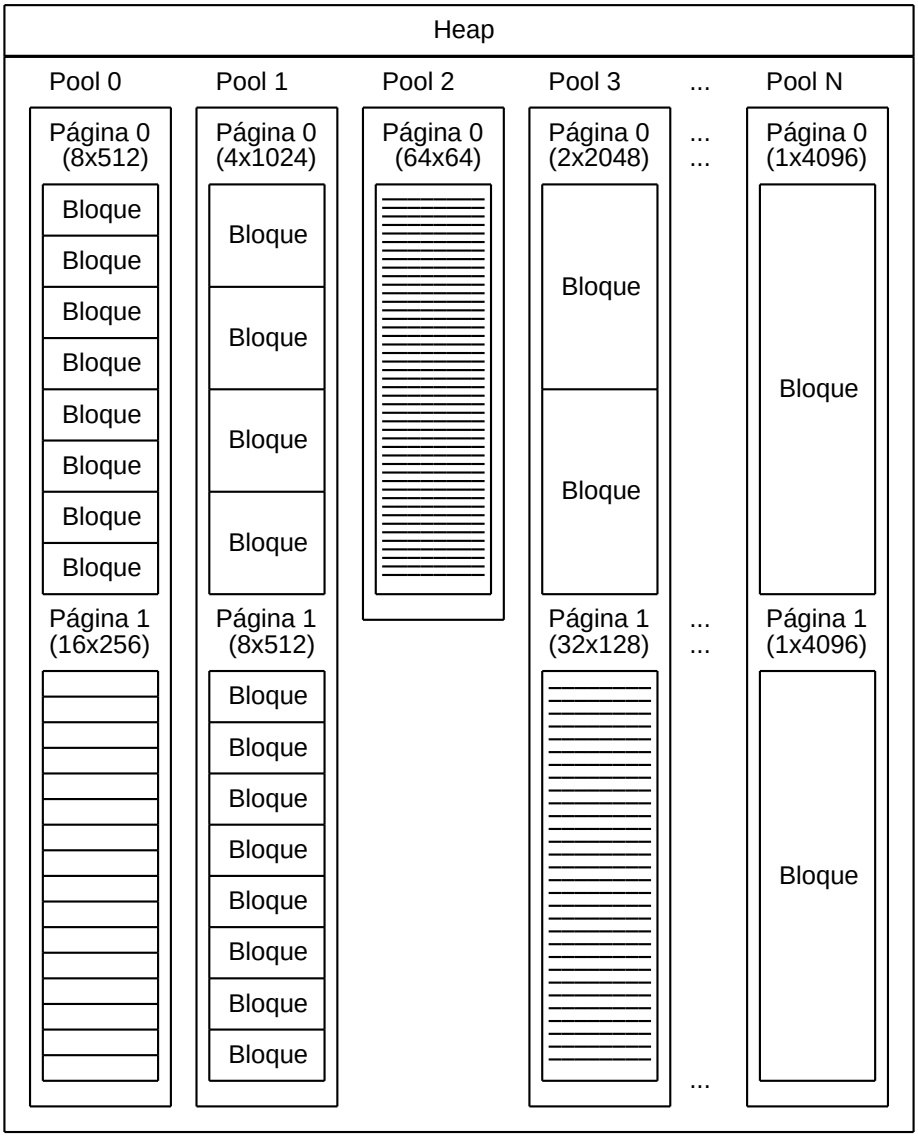


Figura 4.1: Organización del *heap*. En este ejemplo todos los *pools* tienen 2 páginas excepto el *pool* 2 que tiene una sola. El tamaño de bloque que almacena cada página varía entre 64 bytes (página 0 del *pool* 2) hasta 4096 (ambas páginas del *pool* N) que es una página completa.

tamaño de bloque (ver figura 4.1). Los tamaños de bloque posibles son potencias de 2 desde 16 bytes hasta 4096 (el tamaño típico de una página), es decir: 16, 32, 64, 128, 256, 512, 1024, 2048 y 4096 [2]. Todos los objetos, arreglos o celdas en general se ubican en estos bloques (en uno del tamaño más pequeño que haya que sea suficientemente grande como para almacenar dicho objeto). En caso de que un objeto sea mayor a una página, se utilizan la menor cantidad de páginas contiguas de un pool que tengan espacio suficiente para almacenar dicho objeto.

Cada página de un *pool* puede tener asignado un tamaño de bloque específico o puede estar libre. A su vez, cada bloque puede estar ocupado por una celda o estar libre. Los bloques libres de un tamaño específico (a excepción de aquellos bloques que ocupen una página entera) además forman parte de una *lista de libres* (ver figura 4.2 en la página siguiente). Esto permite asignar objetos relativamente pequeños de forma bastante eficiente.

[2] Además existe otro tamaño de bloque especial que se utiliza para indicar la continuación de un objeto grande (que ocupan más de una página).

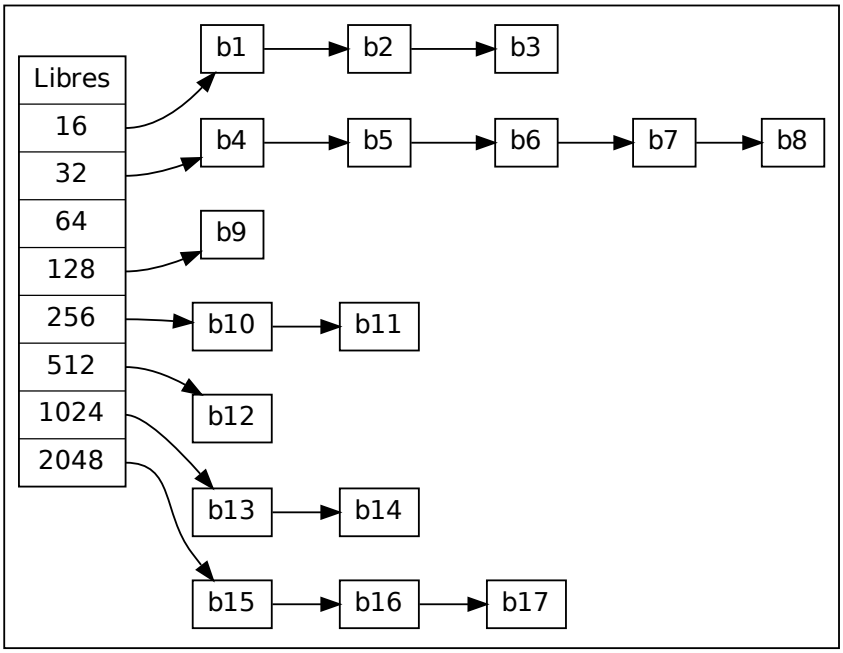


Figura 4.2: Ejemplo de listas de libres.

Atributos de *pool*

Cada *pool* tiene la siguiente información asociada:

number_of_pages
: Cantidad de páginas que tiene. Esta cantidad es fija en toda la vida de un *pool*.

pages
: Bloque de memoria contiguo de tamaño `PAGE_SIZE * number_of_pages` (siendo `PAGE_SIZE` el tamaño de página, que normalmente son 4096 bytes).

Atributos de página

Cada página dentro de un *pool* tiene un único atributo asociado: *block_size*. Se trata del tamaño de los bloques que almacena esta página.

Una página siempre almacena bloques del mismo tamaño, que pueden ser 16, 32, 64, 128, 256, 512, 1024, 2048 o 4096 (llamado con el nombre especial `PAGE`). Además hay dos tamaños de bloque simbólicos que tienen un significado especial:

`FREE`
: Indica que la página está completamente libre y disponible para albergar cualquier tamaño de bloque que sea necesario (pero una vez que se le asignó un nuevo tamaño de bloque ya no puede ser cambiado hasta que la página vuelva a liberarse por completo).

`CONTINUATION`
: Indica que esta página es la continuación de un objeto grande (es decir, que ocupa dos o más páginas). Luego se presentan más detalles sobre objetos grandes.

Las páginas con estos tamaños de bloque especiales conceptualmente no contienen bloques.

Atributos de bloque

Cada bloque tiene asociados varios atributos:

mark
: Utilizado en la fase de *marcado*, indica que un nodo ya fue visitado (serían las celdas *negras* en la *abstracción tricolor*).

scan
: Utilizado también en la fase de *marcado*, indica que una celda visitada todavía tiene *hijas* sin marcar (serían las celdas *grises* en la *abstracción tricolor*).

free
: Indica que el bloque está libre (no está siendo utilizado por ningún objeto *vivo*). Esto es necesario solo por la forma en la que realiza el *marcado* y *barrido* en el *algoritmo actual* (las celdas con este atributo son tomadas como *basura* aunque estén marcadas con *mark*).

final
: Indica que el bloque contiene un objeto que tiene un destructor (que debe ser llamado cuando la celda pasa de *viva* a *basura*).

noscan
: Indica que el bloque contiene un objeto que no tiene punteros y por lo tanto no debe ser escaneado (no tiene *hijas*).

Objetos grandes

El recolector de basura actual de D trata de forma diferente a los objetos grandes. Todo objeto grande empieza en un bloque con tamaño `PAGE` y (opcionalmente) continúa en los bloques contiguos subsiguientes que tengan el tamaño de bloque `CONTINUATION` (si el objeto ocupa más que una página). El fin de un objeto grande queda marcado por el fin del *pool* o una página con tamaño de bloque distinto a `CONTINUATION` (lo que suceda primero).

Cuando un objeto grande se convierte en *basura*, todas sus páginas se liberan por completo, siendo marcadas con tamaño `FREE` para que puedan ser almacenado en ellas otros objetos grandes o incluso nuevos bloques de un tamaño determinado.

4.2.2 Algoritmos del recolector

A continuación se explica como provee el recolector actual de D los servicios básicos que debe proveer cualquier recolector, como se presentó en la sección *Servicios*.

Cabe aclarar que se presenta una versión simplificada del algoritmo, o más precisamente, de la implementación del algoritmo, ya que no se exponen en esta sección muchas optimizaciones que harían muy compleja la tarea de explicar como funciona conceptualmente. En la siguiente sección, *Detalles de implementación*, se darán más detalles sobre las optimizaciones importantes y diferencias con el algoritmo aquí presentado, junto con detalles sobre como se implementa la organización del *heap* que se explicó en la sección anterior.

Recolección

A grandes rasgos el algoritmo de recolección puede resumirse de las dos fases básicas de cualquier algoritmo de *marcado y barrido*:

```
function collect() is
   mark_phase()
   sweep_phase()
```

Fase de marcado

Esta fase consiste de varios pasos, que pueden describirse con el siguiente algoritmo:

```
function mark_phase() is
   global more_to_scan = false
   stop_the_world()
   clear_mark_scan_bits()
   mark_free_lists()
   mark_static_data()
   push_registers_into_stack(thread_self)
   thread_self.stack.end = get_stack_top()
   mark_stacks()
   pop_registers_from_stack(thread_self)
   mark_user_roots()
   mark_heap()
   start_the_world()
```

La variable **global** `more_to_scan` indica al algoritmo iterativo cuando debe finalizar: la función `mark_range()` (que veremos más adelante) lo pone en `true` cuando una nueva celda debe ser visitada, por lo tanto la iteración se interrumpe cuando no hay más celdas por visitar.

Las funciones `stop_the_world()` y `start_the_world()` pausan y reanudan todos los hilos respectivamente (salvo el actual). Al pausar los hilos además se apilan los registros del procesador en el *stack* y se guarda la posición actual del *stack* para que la fase de marcado pueda recorrerlos [3]:

```
function stop_the_world() is
   foreach thread in threads
      if thread is thread_self
         continue
      thread.pause()
      push_registers_into_stack(thread)
      thread.stack.end = get_stack_top()

function start_the_world() is
   foreach thread in reversed(threads)
      if thread is thread_self
         continue
      pop_registers_from_stack(thread)
      thread.resume()
```

La función `clear_mark_scan_bits()` se encarga de restablecer todos los atributos *mark* y *scan* de cada bloque del *heap*:

```
function clear_mark_scan_bits() is
   foreach pool in heap
```

[3] El procedimiento para apilar y desapilar los registros en el *stack* se realiza en realidad utilizando las señales `SIGUSR1` y `SIGUSR2` (respectivamente). Es el manejador de la señal el que en realidad apila y desapila los registros y guarda el puntero al *stack*. Se omiten los detalles para simplificar la explicación del algoritmo.

```
      foreach page in pool
         foreach block in page
            block.mark = false
            block.scan = false
```

La función `mark_free_lists()` por su parte se encarga de activar el bit *mark* de todos los bloques de las listas de libres de manera de que la fase de marcado (que es iterativa y realiza varias pasadas sobre **todo** el *heap*, incluyendo las celdas libres) no visite las celdas libres perdiendo tiempo sin sentido y potencialmente manteniendo *vivas* celdas que en realidad son *basura* (*falsos positivos*):

```
function mark_free_lists() is
   foreach free_list in free_lists
      foreach block in free_list
         block.mark = true
         block.free = true
```

Notar que los bloques libres quedan entonces marcados aunque sean *basura* por definición. Para evitar que en la etapa de barrido se tomen estos bloques como celdas vivas, a todos los bloques en la lista de libres también se los marca con el bit *free*, así el barrido puede tomar como *basura* estos bloques aunque estén marcados.

El *root set* está compuesto por el área de memoria estática (variables globales), los *stacks* de todos los hilos y los registros del procesador. Primero se marca el área de memoria estática de manera *conservativa* (es decir, tomando cada *word* como si fuera un puntero):

```
function mark_static_data() is
   mark_range(static_data.begin, static_data.end)
```

Para poder tomar los registros como parte del *root set* primero se apilan en el *stack* a través de la función:

```
function push_registers_into_stack(thread) is
   foreach register in thread.registers
      push(register)
```

Y luego, al reiniciar los hilos cuando se termina de marcar, se descartan sacándolos de la pila (no es necesario ni correcto restablecer los valores ya que podrían tener nuevos valores):

```
function pop_registers_from_stack(thread) is
   foreach register in reverse(thread.registers)
      pop()
```

Una vez hecho esto, basta marcar (de forma conservativa) los *stacks* de todos los threads para terminar de marcar el *root set*:

```
function mark_stacks() is
   foreach thread in threads
      mark_range(thread.stack.begin, thread.stack.end)
```

Dado que D soporta manejo de memoria manual al mismo tiempo que memoria automática, es posible que existan celdas de memoria que no estén en el *root set* convencional ni en el *heap* del recolector. Para evitar que se libere alguna celda a la cual todavía existen referencias desde memoria administrada por el usuario, éste debe informarle al recolector sobre la existencia de estas nuevas raíces. Es por esto que para concluir el marcado del *root set* completo se procede a marcar las raíces definidas por el usuario:

```
function mark_user_roots() is
   foreach root_range in user_roots
      mark_range(root_range.begin, root_range.end)
```

El algoritmo de marcado no es recursivo sino iterativo por lo tanto al marcar una celda (o bloque) no se siguen sus *hijas*, solo se activa el bit de *scan* (a menos que la celda no contenga punteros, es decir, tenga el bit *noscan*):

```
function mark_range(begin, end) is
   pointer = begin
   while pointer < end
      [pool, page, block] = find_block(pointer)
      if block is not null and block.mark is false
         block.mark = true
         if block.noscan is false
            block.scan = true
            global more_to_scan = true
      pointer++
```

Por lo tanto en este punto, tenemos todas las celdas inmediatamente alcanzables desde el *root set* marcadas y con el bit *scan* activado si la celda puede contener punteros. Por lo tanto solo resta marcar (nuevamente de forma conservativa) iterativamente todo el *heap* hasta que no hayan más celdas para visitar (con el bit *scan* activo):

```
function mark_heap() is
   while global more_to_scan
      global more_to_scan = false
      foreach pool in heap
         foreach page in pool
            if page.block_size <= PAGE // saltea FREE y CONTINUATION
               foreach block in page
                  if block.scan is true
                     block.scan = false
                     if page.block_size is PAGE // objeto grande
                        begin = cast(byte*) page
                        end = find_big_object_end(pool, page)
                        mark_range(begin, end)
                     else // objeto pequeño
                        mark_range(block.begin, block.end)
```

Aquí puede verse, con un poco de esfuerzo, la utilización de la *abstracción tricolor*: todas las celdas alcanzables desde el *root set* son pintadas de *gris* (tienen los bits *mark* y *scan* activados), excepto aquellas celdas atómicas (es decir, que se sabe que no tienen punteros) que son marcadas directamente de *negro*. Luego se van obteniendo celdas del conjunto de las *grises*, se las pinta de *negro* (es decir, se desactiva el bit *scan*) y se pintan todas sus *hijas* de *gris* (o *negro* directamente si no tienen punteros). Este procedimiento se repite mientras el conjunto de celdas *grises* no sea vacío (es decir, que `more_to_scan` sea `true`).

A continuación se presenta la implementación de las funciones suplementarias utilizadas en la fase de marcado:

```
function find_big_object_end(pool, page) is
   pool_end = cast(byte*) pool.pages + (PAGE_SIZE * pool.number_of_pages)
   do
```

```
      page = cast(byte*) page + PAGE_SIZE
   while page.block_size is CONTINUATION and page < pool_end
   return page

function find_block(pointer) is
   foreach pool in heap
      foreach page in pool
         if page.block_size is PAGE
            big_object_start = cast(byte*) page
            big_object_end = find_big_object_end(pool, page)
            if big_object_start <= pointer < big_object_end
               return [pool, page, big_object_start]
         else if page.block_size < PAGE
            foreach block in page
               block_start = cast(byte*) block
               block_end = block_start + page.block_size
               if block_start <= pointer < block_end
                  return [pool, page, block_start]
   return [null, null, null]
```

Cabe destacar que la función `find_block()` devuelve el *pool*, la página y el comienzo del bloque al que apunta el puntero, es decir, soporta punteros *interiores*.

Fase de barrido

Esta fase es considerablemente más sencilla que el marcado; el algoritmo puede dividirse en dos pasos básicos:

```
function sweep_phase() is
   sweep()
   rebuild_free_lists()
```

El barrido se realiza con una pasada por sobre todo el *heap* de la siguiente manera:

```
function sweep() is
   foreach pool in heap
      foreach page in pool
         if page.block_size <= PAGE // saltea FREE y CONTINUATION
            foreach block in page
               if block.mark is false
                  if block.final is true
                     finalize(block)
                  block.free = true
                  block.final = false
                  block.noscan = false
                  if page.block_size is PAGE // objeto grande
                     free_big_object(pool, page)
```

Como se observa, se recorre todo el *heap* en busca de bloques y páginas libres. Los bloques libres son marcados con el atributo `free` y las páginas libres son marcadas con el tamaño de bloque simbólico `FREE`. Para los objetos grandes se marcan todas las páginas que utilizaban como `FREE`:

```
function free_big_object(pool, page) is
   pool_end = cast(byte*) pool.pages + (PAGE_SIZE * pool.number_of_pages)
```

```
   do
      page.block_size = FREE
      page = cast(byte*) page + PAGE_SIZE
   while page < pool_end and page.block_size is CONTINUATION
```

Además, los bloques que tienen en atributo `final` son finalizados llamando a la función `finalize()`. Esta función es un servicio que provee la biblioteca *runtime* y en última instancia llama al destructor del objeto almacenado en el bloque a liberar.

Una vez marcados todos los bloques y páginas con `free`, se procede a reconstruir las listas de libres. Como parte de este proceso se buscan las páginas que tengan todos los bloques libres para marcar la página completa como libre (de manera que pueda utilizarse para albergar otro tamaño de bloque u objetos grandes de ser necesario):

```
function rebuild_free_lists() is
   foreach free_list in free_lists
      free_list.clear()
   foreach pool in heap
      foreach page in pool
         if page.block_size < PAGE // objetos pequeños
            if is_page_free(page)
               page.block_size = FREE
            else
               foreach block in page
                  if block.free is true
                     free_lists[page.block_size].link(block)
```

Esta reorganización de listas libres además mejoran la localidad de referencia y previenen la fragmentación. La localidad de referencia se ve mejorada debido a que asignaciones de memoria próximas en el tiempo serán también próximas en espacio porque pertenecerán a la misma página (al menos si las asignaciones son todas del mismo tamaño). La fragmentación se minimiza por el mismo efecto, primero se asignarán todos los bloques de la misma página.

A continuación se presenta la implementación de una de las funciones suplementarias de la fase de barrido:

```
function is_page_free(page) is
   foreach block in page
      if block.free is false
         return false
   return true
```

Las demás funciones suplementarias pertenecen a la manipulación de listas libres que no son más que operaciones sobre una lista simplemente enlazada. En la sección *Detalles de implementación* se verá con más detalles como las implementa el recolector actual.

Asignación de memoria

La asignación de memoria del recolector es relativamente compleja, excepto cuando se asigna un objeto pequeño y ya existe algún bloque con el tamaño preciso en la lista de libres. Para el resto de los casos la cantidad de trabajo que debe hacer el recolector para asignar la memoria es considerable.

El algoritmo de asignación de memoria se puede resumir así:

```
function new(size, attrs) is
   block_size = find_block_size(size)
   if block_size < PAGE
      block = new_small(block_size)
   else
      block = new_big(size)
   if block is null
      throw out_of_memory
   if final in attrs
      block.final = true
   if noscan in attrs
      block.noscan = true
   return cast(void*) block
```

La función `find_block_size()` sencillamente busca el tamaño de bloque se mejor se ajuste al tamaño solicitado (es decir, el bloque más pequeño lo suficientemente grande como para poder almacenar el tamaño solicitado). Una vez más el algoritmo distingue objetos grandes de pequeños. Los pequeños se asignan de las siguiente manera:

```
function new_small(block_size) is
   block = find_block_with_size(block_size)
   if block is null
      collect()
      block = find_block_with_size(block_size)
      if block is null
         new_pool()
         block = find_block_with_size(block_size)
   return block
```

Se intenta reiteradas veces conseguir un bloque del tamaño correcto libre, realizando diferentes acciones si no se tiene éxito. Primero se intenta hacer una *recolección* y si no se puede encontrar suficiente espacio luego de ella se intenta crear un nuevo *pool* de memoria pidiendo memoria al *low level allocator* (el sistema operativo generalmente).

Para intentar buscar un bloque de memoria libre se realiza lo siguiente:

```
function find_block_with_size(block_size) is
   block = free_lists[block_size].pop_first()
   if block is null
      assign_page(block_size)
      block = free_lists[block_size].pop_first()
   return block
```

Donde `pop_first()` retorna `null` si la lista estaba vacía. Si no se puede obtener un bloque de la lista de libres correspondiente, se busca asignar una página libre al tamaño de bloque deseado de forma de *alimentar* la lista de libres con dicho tamaño:

```
function assign_page(block_size) is
   foreach pool in heap
      foreach page in pool
         if page.block_size is FREE
            page.block_size = block_size
            foreach block in page
               free_lists[page.block_size].link(block)
```

Cuando todo ello falla, el último recurso consiste en pedir memoria al sistema operativo, creando un nuevo *pool*:

```
function new_pool(number_of_pages = 1) is
   pool = alloc(pool.sizeof)
   if pool is null
      return null
   pool.number_of_pages = number_of_pages
   pool.pages = alloc(number_of_pages * PAGE_SIZE)
   if pool.pages is null
      free(pool)
      return null
   heap.add(pool)
   foreach page in pool
      page.block_size = FREE
   return pool
```

Se recuerda que la función `alloc()` es un *servicio* provisto por el *low level allocator* y en la implementación actual de D en general es el sistema operativo (aunque opcionalmente puede utilizarse la biblioteca estándar de C, que a su vez utiliza el sistema operativo).

Cualquier error en estas funciones es propagado y en última instancia, cuando todo falla, la función `new()` termina lanzando una excepción indicando que se agotó la memoria.

Si el tamaño de bloque necesario para cumplir con la asignación de memoria es de una o más páginas, entonces se utiliza otro algoritmo para alocar un objeto grande:

```
function new_big(size) is
   number_of_pages = ceil(size / PAGE_SIZE)
   pages = find_pages(number_of_pages)
   if pages is null
      collect()
      pages = find_pages(number_of_pages)
      if pages is null
         minimize()
         pool = new_pool(number_of_pages)
         if pool is null
            return null
         pages = assign_pages(pool, number_of_pages)
   pages[0].block_size = PAGE
   foreach page in pages[1..end]
      page.block_size = CONTINUATION
   return pages[0]
```

De forma similar a la asignación de objetos pequeños, se intenta encontrar una serie de páginas contiguas, dentro de un mismo *pool*, suficientes para almacenar el tamaño requerido y si esto falla, se realizan diferentes pasos y se vuelve a intentar. Puede observarse que, a diferencia de la asignación de objetos pequeños, si luego de la recolección no se pudo encontrar lugar suficiente, se trata de minimizar el uso de memoria física utilizando la siguiente función, que devuelve al *low level allocator* los *pools* completamente libres:

```
function minimize() is
   foreach pool in heap
      all_free = true
      foreach page in pool
         if page.block_size is not FREE
```

```
            all_free = false
            break
      if all_free is true
         free(pool.pages)
         free(pool)
         heap.remove(pool)
```

Volviendo a la función `new_big()`, para hallar una serie de páginas contiguas se utiliza el siguiente algoritmo:

```
function find_pages(number_of_pages) is
   foreach pool in heap
      pages = assign_pages(pool, number_of_pages)
      if pages
         return pages
   return null
```

Como se dijo, las páginas deben estar contenidas en un mismo *pool* (para tener la garantía de que sean contiguas), por lo tanto se busca *pool* por *pool* dicha cantidad de páginas libres consecutivas a través del siguiente algoritmo:

```
function assign_pages(pool, number_of_pages) is
   pages_found = 0
   first_page = null
   foreach page in pool
      if page.block_size is FREE
         if pages_found is 0
            pages_found = 1
            first_page = page
         else
            pages_found = pages_found + 1
         if pages_found is number_of_pages
            return [first_page .. page]
      else
         pages_found = 0
         first_page = null
   return null
```

Una vez más, cuando todo ello falla (incluso luego de una recolección), se intenta alocar un nuevo *pool*, esta vez con una cantidad de páginas suficientes como para almacenar el objeto grande y si esto falla el error se propaga hasta la función `new()` que lanza una excepción.

Liberación de memoria

La liberación de la memoria asignada puede hacerse explícitamente. Esto saltea el mecanismo de recolección, y es utilizado para dar soporte a manejo explícito de memoria asignada en el *heap* del recolector. En general el usuario no debe utilizar liberación explícita, pero puede ser útil en casos muy particulares:

```
function delete(pointer) is
   [pool, page, block_start] = find_block(pointer)
   if block is not null
      block.free = true
      block.final = false
      block.noscan = false
```

```
if page.block_size is PAGE // objeto grande
   free_big_object(pool, page)
else // objeto pequeño
   free_lists[page.block_size].link(block)
```

Como se puede observar, si el objeto es pequeño se enlaza a la lista de libres correspondiente y si es grande se liberan todas las páginas asociadas a éste, de forma similar a la *fase de barrido*. A diferencia de ésta, no se finaliza el objeto (es decir, no se llama a su destructor).

Finalización

Al finalizar el programa, el recolector es finalizado también y lo que realiza actualmente, además de liberar la memoria propia del recolector, es realizar una recolección. Es decir, si hay objetos que son todavía alcanzables desde el *root set*, esos objetos no son finalizados (y por lo tanto sus destructores no son ejecutados).

Como se ha visto, esto es perfectamente válido ya que D no garantiza que los objetos sean finalizados.

4.2.3 Detalles de implementación

Hay varias diferencias a nivel de implementación entre lo que se presentó en las secciones anteriores y como está escrito realmente el recolector actual. Con los conceptos e ideas principales ya explicadas, se procede a ahondar con más detalle en como está construido el recolector y algunas de sus optimizaciones principales.

Vale aclarar que el recolector de basura actual está implementado en D.

Estructuras de datos del recolector

El recolector está principalmente contenido en la estructura llamada `Gcx`. Dicha estructura tiene los siguientes atributos (divididos en categorías para facilitar la comprensión):

Raíces definidas por el usuario

***roots* (*nroots*, *rootdim*)**
: Arreglo variable de punteros simples que son tomados como raíces provistas por el usuario.

***ranges* (*nranges*, *rangedim*)**
: Arreglo variable de rangos de memoria que deben ser revisados (de forma conservativa) como raíces provistas por el usuario. Un rango es una estructura con dos punteros: `pbot` y `ptop`. Toda la memoria entre estos dos punteros se toma, palabra por palabra, como una raíz del recolector.

Estado interno del recolector

anychanges
: Variable que indica si en la fase de marcado se encontraron nuevas celdas con punteros que deban ser visitados. Otra forma de verlo es como un indicador de si el conjunto de celdas *grises* está vacío luego de una iteración de marcado (utilizando la *abstracción tricolor*). Es análoga a la variable `more_to_scan` presentada en *Fase de marcado*.

***inited* (sic)**
: Indica si el recolector fue inicializado.

stackBottom
: Puntero a la base del *stack* (asumiendo que el stack crece hacia arriba). Se utiliza para saber por donde comenzar a visitar el *stack* de forma conservativa, tomándolo con una raíz del recolector.

***Pools* (*pooltable*, *npools*)**
: Arreglo variable de punteros a estructuras `Pool` (ver más adelante). Este arreglo se mantiene siempre ordenado de menor a mayor según la dirección de memoria de la primera página que almacena.

bucket
: Listas de libres. Es un arreglo de estructuras `List` utilizadas para guardar la listas de libres de todos los tamaños de bloques posibles (ver más adelante).

Atributos que cambian el comportamiento

noStack
: Indica que no debe tomarse al *stack* como raíz del recolector. Esto es muy poco seguro y no debería ser utilizado nunca, salvo casos extremadamente excepcionales.

log
: Indica si se debe guardar un registro de la actividad del recolector. Es utilizado principalmente para depuración.

disabled
: Indica que no se deben realizar recolecciones implícitamente. Si al tratar de asignar memoria no se puede hallar celdas libres en el *heap* del recolector, se pide más memoria al sistema operativo sin correr una recolección para intentar recuperar espacio. Esto es particularmente útil para secciones de un programa donde el rendimiento es crítico y no se pueden tolerar grandes pausas como las que puede provocar el recolector.

Optimizaciones

p_cache*, *size_cache
: Caché del tamaño de bloque para un puntero dado. Obtener el tamaño de un bloque es una tarea costosa y común. Para evitarla en casos donde se calcula de forma sucesiva el tamaño del mismo bloque (como puede ocurrir al concatenar arreglos dinámicos) se guarda en un caché (de un solo elemento) el último valor calculado.

minAddr*, *maxAddr
: Punteros al principio y fin del *heap*. Pueden haber *huecos* entre estos dos punteros que no pertenezcan al *heap* pero siempre se cumple que si un puntero apunta al *heap* debe estar en este rango. Esto es útil para hacer un cálculo rápido para descartar punteros que fueron tomados de forma conservativa y en realidad no apuntan al *heap* (ver la función `find_block()` en *Fase de marcado*).

Pools

La primera diferencia es como está organizado el *heap*. Si bien la explicación presentada en la sección *Organización del heap* es correcta, la forma en la que está implementado no es tan *naïve* como los algoritmos presentados en *Algoritmos del recolector* sugieren.

El recolector guarda un arreglo variable de estructuras `Pool`. Cabe destacar que para implementar el recolector no se pueden utilizar los arreglos dinámicos de D (ver sección *Programación de alto nivel*) dado que éstos utilizan de forma implícita el recolector de basura, por lo tanto todos los arreglos variables

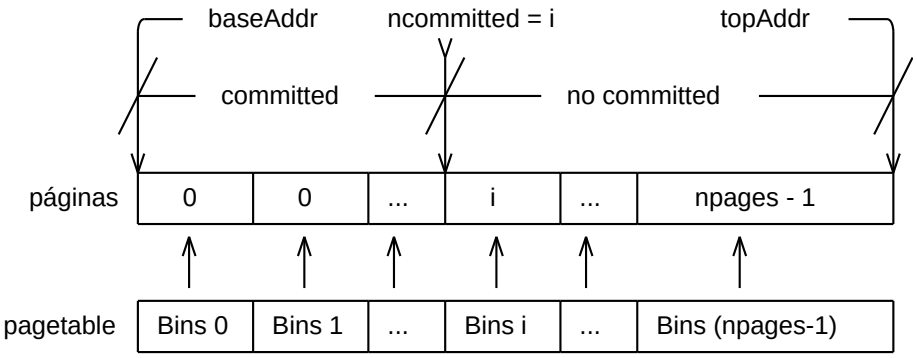


Figura 4.3: Vista gráfica de la estructura de un *pool* de memoria.

del recolector se implementan utilizando las funciones de C `malloc(3)`, `realloc(3)` y `free(3)` directamente.

La estructura `Pool` está compuesta por los siguientes atributos (ver figura 4.3):

baseAddr* y *topAddr
: Punteros al comienzo y fin de la memoria que almacena todas las páginas del *pool* (*baseAddr* es análogo al atributo *pages* utilizado en las secciones anteriores para mayor claridad).

mark*, *scan*, *freebits*, *finals*, *noscan
: Conjuntos de bits (*bitsets*) para almacenar los indicadores descriptos en *Organización del heap* para todos los bloques de todas las páginas del *pool*. *freebits* es análogo a *free* y *finals* a *final* en los atributos descriptos en las secciones anteriores.

npages
: Cantidad de páginas que contiene este *pool* (fue nombrado *number_of_pages* en las secciones anteriores para mayor claridad).

ncommitted
: Cantidad de páginas *encomendadas* al sistema operativo (*committed* en inglés). Este atributo no se mencionó anteriormente porque el manejo de páginas encomendadas le agrega una complejidad bastante notable al recolector y es solo una optimización para un sistema operativo en particular (Microsoft Windows).

pagetable
: Arreglo de indicadores de tamaño de bloque de cada página de este *pool*. Los indicadores válidos son `B_16` a `B_2048` (pasando por los valores posibles de bloque mencionados anteriormente, todos con el prefijo "`B_`"), `B_PAGE`, `B_PAGEPLUS` (análogo a `CONTINUATION`), `B_UNCOMMITTED` (valor que tienen las páginas que no fueron encomendadas aún) y `B_FREE`.

Como se observa, además de la información particular del *pool* se almacena toda la información de páginas y bloques enteramente en el *pool* también. Esto simplifica el manejo de lo que es memoria *pura* del *heap*, ya que queda una gran porción continua de memoria sin estar intercalada con meta-información del recolector.

Para poder acceder a los bits de un bloque en particular, se utiliza la siguiente cuenta para calcular el índice en el *bitset*:

$$index(p) = \frac{p - baseAddr}{16}$$

Donde `p` es la dirección de memoria del bloque. Esto significa que, sin importar cual es el tamaño de bloque de las páginas del *pool*, el *pool* siempre reserva suficientes bits como para que todas las páginas

puedan tener tamaño de bloque de 16 bytes. Esto puede ser desperdiciar bastante espacio si no predomina un tamaño de bloque pequeño.

Listas de libres

Las listas de libres se almacenan en el recolector como un arreglo de estructuras `List`, que se compone solamente de un atributo `List* next` (es decir, un puntero al siguiente). Entonces cada elemento de ese arreglo es un puntero al primer elemento de la lista en particular.

La implementación utiliza a los bloques de memoria como nodos directamente. Como los bloques siempre pueden almacenar una palabra (el bloque de menor tamaño es de 16 bytes y una palabra ocupa comúnmente entre 4 y 8 bytes según se trabaje sobre arquitecturas de 32 o 64 bits respectivamente), se almacena el puntero al siguiente en la primera palabra del bloque.

Algoritmos

Los algoritmos en la implementación real son considerablemente menos modulares que los presentados en la sección *Algoritmos del recolector*. Por ejemplo, la función `collect()` es una gran función de 300 líneas de código fuente.

A continuación se resumen las funciones principales, separadas en categorías para facilitar la comprensión. Los siguientes son métodos de la estructura `Gcx`:

Inicialización y terminación

initialize()
: Inicializa las estructuras internas del recolector para que pueda ser utilizado. Esta función la llama la biblioteca *runtime* antes de que el programa comience a correr.

Dtor()
: Libera todas las estructuras que utiliza el recolector.

Manipulación de raíces definidas por el usuario

addRoot(p), removeRoot(p), rootIter(dg)
: Agrega, remueve e itera sobre las raíces simples definidas por el usuario.

addRange(pbot, ptop), remove range(pbot), rangeIter(dg)
: Agrega, remueve e itera sobre los rangos de raíces definidas por el usuario.

Manipulación de bits indicadores

getBits(pool, biti)
: Obtiene los indicadores especificados para el bloque de índice `biti` en el *pool* `pool`.

setBits(pool, biti, mask)
: Establece los indicadores especificados en `mask` para el bloque de índice `biti` en el *pool* `pool`.

clrBits(pool, biti, mask)
: Limpia los indicadores especificados en `mask` para el bloque de índice `biti` en el *pool* `pool`.

Cada bloque (*bin* en la terminología de la implementación del recolector) tiene ciertos indicadores asociados. Algunos de ellos pueden ser manipulados (indirectamente) por el usuario utilizando las funciones mencionadas arriba.

El parámetro `mask` debe ser una máscara de bits que puede estar compuesta por la conjunción de los siguientes valores:

FINALIZE
El objeto almacenado en el bloque tiene un destructor (indicador *finals*).

NO_SCAN
El objeto almacenado en el bloque no contiene punteros (indicador *noscan*).

NO_MOVE
El objeto almacenado en el bloque no debe ser movido [4].

Búsquedas

findPool(p)
Busca el *pool* al que pertenece el objeto apuntado por `p`.

findBase(p)
Busca la dirección base (el inicio) del bloque apuntado por `p` (`find_block()` según la sección *Fase de marcado*).

findSize(p)
Busca el tamaño del bloque apuntado por `p`.

getInfo(p)
Obtiene información sobre el bloque apuntado por `p`. Dicha información se retorna en una estructura `BlkInfo` que contiene los siguientes atributos: `base` (dirección del inicio del bloque), `size` (tamaño del bloque) y `attr` (atributos o indicadores del bloque, los que se pueden obtener con `getBits()`).

findBin(size)
Calcula el tamaño de bloque más pequeño que pueda contener un objeto de tamaño `size` (`find_block_size()` según lo visto en *Asignación de memoria*).

Asignación de memoria

reserve(size)
Reserva un nuevo *pool* de al menos `size` bytes. El algoritmo nunca crea un *pool* con menos de 256 páginas (es decir, 1 MiB).

minimize()
Minimiza el uso de la memoria retornando *pools* sin páginas usadas al sistema operativo.

newPool(n)
Reserva un nuevo *pool* con al menos `n` páginas. Junto con `Pool.initialize()` es análoga a `new_pool()`, solo que esta función siempre incrementa el número de páginas a, al menos, 256 páginas (es decir, los *pools* son siempre mayores a 1 MiB). Si la cantidad de páginas pedidas supera 256, se incrementa el número de páginas en un 50 % como para que sirva para futuras asignaciones también. Además a medida que la cantidad de *pools* crece, también trata de obtener cada vez más memoria. Si ya había un *pool*, el 2do tendrá como mínimo 2 MiB, el 3ro 3 MiB y así sucesivamente hasta 8 MiB. A partir de ahí siempre crea *pools* de 8 MiB o la cantidad pedida, si ésta es mayor.

Pool.initialize(n_pages)
Inicializa un nuevo *pool* de memoria. Junto con `newPool()` es análoga a `new_pool()`.

[4] Si bien el recolector actual no tiene la capacidad de mover objetos, la interfaz del recolector hacer que sea posible una implementación que lo haga, ya que a través de este indicador se pueden fijar objetos apuntados desde algún segmento no conservativo (objeto *pinned*).

Mientras `newPool()` es la encargada de calcular la cantidad de páginas y crear el objeto *pool*, esta función es la que pide la memoria al sistema operativo. Además inicializa los conjuntos de bits: `mark`, `scan`, `freebits`, `noscan`. `finals` se inicializa de forma perezosa, cuando se intenta asignar el atributo `FINALIZE` a un bloque, se inicializa el conjunto de bits `finals` de todo el *pool*.

allocPage(bin)
Asigna a una página libre el tamaño de bloque `bin` y enlaza los nuevos bloques libres a la lista de libres correspondiente (análogo a `assign_page()`).

allocPages(n)
Busca `n` cantidad de páginas consecutivas libres (análoga a `find_pages(n)`).

malloc(size, bits)
Asigna memoria para un objeto de tamaño `size` bytes. Análoga al algoritmo `new(size, attr)` presentado, excepto que introduce además un caché para no recalcular el tamaño de bloque necesario si se realizan múltiples asignaciones consecutivas de objetos del mismo tamaño y que la asignación de objetos pequeños no está separada en una función aparte.

bigAlloc(size)
Asigna un objeto grande (análogo a `new_big()`). La implementación es mucho más compleja que la presentada en `new_big()`, pero la semántica es la misma. La única diferencia es que esta función aprovecha que `fullcollectshell()` / `fullcollect()` retornan la cantidad de páginas liberadas en la recolección por lo que puede optimizar levemente el caso en que no se liberaron suficientes páginas para asignar el objeto grande y pasar directamente a crear un nuevo *pool*.

free(p)
Libera la memoria apuntada por `p` (análoga a `delete()` de la sección anterior).

Recordar que la `pooltable` siempre se mantiene ordenada según la dirección de la primera página.

Recolección

mark(pbot, ptop)
Marca un rango de memoria. Este método es análogo al `mark_range()` presentado en la sección *Fase de marcado*.

fullcollectshell()
Guarda los registros del procesador asignado al hilo actual en su *stack* y llama a `fullcollect()`. El resto de los hilos son pausados y sus registros apilados por la función del *runtime* `thread_suspendAll()` (y restablecidos y reiniciados por `thread_resumeAll()`.

fullcollect(stackTop)
Realiza la recolección de basura. Es análoga a `collect()` pero es considerablemente menos modular, todos los pasos se hacen directamente en esta función: marcado del *root set*, marcado iterativo del *heap*, barrido y reconstrucción de la lista de libres. Además devuelve la cantidad de páginas que se liberaron en la recolección, lo que permite optimizar levemente la función `bigAlloc()`.

Finalización

El recolector actual, por omisión, solamente efectúa una recolección al finalizar. Por lo tanto, no se ejecutan los destructores de todos aquellos objetos que son alcanzables desde el *root set* en ese momento.

Existe la opción de no realizar una recolección al finalizar el recolector, pero no de finalizar *todos* los objetos (alcanzables o no desde el *root set*). Si bien la especificación de D permite este comportamiento (de hecho la especificación de D es tan vaga que permite un recolector que no llame jamás a ningún destructor), para el usuario puede ser una garantía muy débil y proveer finalización asegurada puede ser muy deseable.

Memoria *encomendada*

El algoritmo actual divide un *pool* en dos áreas: memoria *encomendada* (*committed* en inglés) y *no-encomendada*. Esto se debe a que originalmente el compilador de D DMD solo funcionaba en Microsoft Windows y este sistema operativo puede asignar memoria en dos niveles. En principio se puede asignar al proceso un espacio de memoria (*address space*) pero sin asignarle la memoria virtual correspondiente. En un paso posterior se puede *encomendar* la memoria (es decir, asignar realmente la memoria virtual).

Para aprovechar esta característica el recolector diferencia estos dos niveles. Sin embargo, esta diferenciación introduce una gran complejidad (que se omitió en las secciones anteriores para facilitar la comprensión), y convierte lo que es una ventaja en un sistema operativo en una desventaja para todos los demás (ya que los cálculos extra se realizan pero sin ningún sentido). De hecho hay sistemas operativos, como Linux, que realizan este trabajo automáticamente (la memoria no es asignada realmente al programa hasta que el programa no haga uso de ella; a esta capacidad se la denomina *overcommit*).

Como se vio en la figura 4.3 en la página 61, las páginas de un *pool* se dividen en *committed* y *uncommitted*. Siempre que el recolector recorre un *pool* en busca de una página o bloque, lo hace hasta la memoria *committed*, porque la *uncommitted* es como si jamás se hubiera pedido al sistema operativo a efectos prácticos. Además, al buscar páginas libres, si no se encuentran entre las *encomendadas* se intenta primero *encomendar* páginas nuevas antes de crear un nuevo *pool*.

Sincronización

Si bien el recolector no es paralelo ni concurrente (ver *Estado del arte*), soporta múltiples *mutator*s. La forma de implementarlo es la más simple. Todas las operaciones sobre el recolector que se llaman externamente están sincronizadas utilizando un *lock* global (excepto cuando hay un solo hilo *mutator*, en cuyo caso se omite la sincronización). Esto afecta también a la asignación de memoria y cualquier otro servicio provisto por el recolector.

4.2.4 Características destacadas

Si bien el recolector en términos generales no se aleja mucho de un *marcado y barrido clásico*, tiene algunas mejoras por sobre el algoritmo más básicos que vale la pena destacar:

Organización del *heap*

El *heap* está organizado de una forma que, si bien no emplea las técnicas más modernas que pueden observarse en el estado del arte (como *regiones*), es relativamente sofisticada. El esquema de *pools* y bloques permite disminuir considerablemente los problemas de *fragmentación* de memoria y evita búsquedas de *huecos* que pueden ser costosas (como *best-fit* [5]) o desperdiciar mucho espacio (como

[5] Las búsquedas de tipo *best-fit* son aquellas donde se busca el *hueco* en el *heap* (es decir, una región contínua de memoria libre) que mejor se ajuste al tamaño del objeto a asignar. Es decir, el *hueco* más pequeño lo suficientemente grande como para almacenarlo.

first-fit [6]), logrando un buen equilibrio entre velocidad y espacio desperdiciado.

Fase de marcado iterativa

A diferencia del algoritmo clásico recursivo, el algoritmo del recolector actual es iterativo. El algoritmo recursivo tiene un problema fundamental: se puede llegar a un desbordamiento de pila (o *stack overflow*). La cantidad de recursiones necesarias es, en el peor caso, $O(|Live\ set|)$ (por ejemplo, si todas las celdas del *heap* formaran una lista simplemente enlazada). Hay muchas técnicas para lidiar con este problema, algunas que podrían aplicarse a D y otras que no (como *pointer reversal*) [JOLI96]. El recolector actual, sin embargo, cambia complejidad en espacio por complejidad en tiempo, utilizando un algoritmo iterativo que es constante ($O(1)$) en espacio, pero que requiere varias pasada sobre el *heap* en vez de una (la cantidad de pasadas en el peor caso es $O(|Live\ set|)$, al igual que la profundidad del algoritmo recursivo, pero cada pasada se realiza sobre todo el *heap*).

Conjuntos de bits para indicadores

El algoritmo clásico propone almacenar en la propia celda la marca (para la fase de marcado) y otros indicadores. El algoritmo del recolector actual utiliza conjuntos de bits. Esto trae dos ventajas principales:

- Permite minimizar el espacio requerido, ya que de otra forma en general se desperdicia una palabra entera como cabecera de celda para guardar este tipo de información.
- Mejora la localidad de referencia, ya que los indicadores se escriben de forma muy compacta y en una región de memoria contigua que generalmente puede entrar en el cache o en pocas páginas de memoria acelerando considerablemente la fase de marcado.

Herramientas para depuración

El recolector provee algunas opciones para simplificar el diagnóstico y depuración de problemas, tanto del mismo recolector como del programa del usuario.

Las opciones más importantes son:

MEMSTOMP
: Su función es escribir un patrón determinado de bits en todos los bytes de un bloque de memoria según se haya:

  - Pedido un bloque menor a una página (`0xF0`).
  - Pedido un bloque mayor a una página (`0xF1`).
  - Dejado de usar debido a un pedido de achicamiento de un bloque (`0xF2`).
  - Pedido más páginas debido a un pedido de agrandamiento de un bloque (`0xF0`).
  - Liberado intencionalmente por el usuario (`0xF2`).
  - Barrido (`0xF3`).

  Esto permite al diagnosticar un problema saber, por ejemplo, si un determinado área de memoria fue recolectada recientemente, o liberada por el usuario, o recién adquirida, etc. con tan solo ver si un patrón de bits determinado está presente. Por supuesto puede existir *falsos positivos* pero su probabilidad es lo suficientemente baja como para que sea útil en la práctica.

[6] Las búsquedas de tipo *first-fit* son aquellas donde se busca el primer *hueco* en el *heap* (es decir, una región contínua de memoria libre) que sea lo suficientemente grande como para almacenar el objeto a asignar.

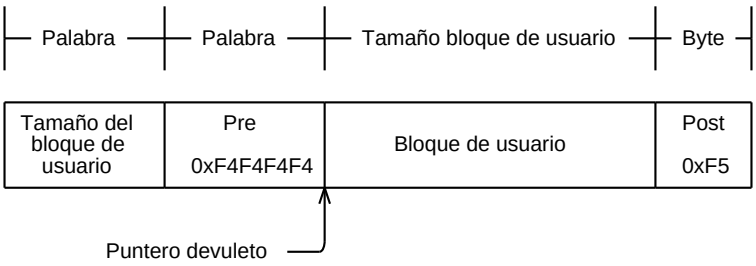


Figura 4.4: Esquema de un bloque cuando está activada la opción `SENTINEL`.

`SENTINEL`
: Su función detectar errores producidos por escribir más allá (o antes) del área de memoria solicitada. Está implementado reservando un poco más de memoria de la que pide el usuario y devolviendo un puntero a un bloque ubicado dentro del bloque real reservado (en vez de al inicio). Escribiendo un patrón de bits en los extremos del bloque real (ver figura 4.4) se puede verificar, en distintas situaciones (como por ejemplo al barrer el bloque), que esas guardas con los patrones de bits estén intactas (en caso contrario se ha escrito por fuera de los límites del bloque solicitado). Esto permite detectar de forma temprana errores tanto en el recolector como en el programa del usuario.

Ambas opciones son seleccionables sólo en tiempo de compilación del recolector, por lo que su utilidad real, al menos para el usuario, se ve severamente reducida.

4.2.5 Problemas y limitaciones

A continuación se presentan los principales problemas encontrados en la implementación actual del recolector de basura de D. Estos problemas surgen principalmente de la observación del código y de aproximadamente tres años de participación y observación del grupo de noticias, de donde se obtuvieron los principales problemas percibidos por la comunidad que utiliza el lenguaje.

Complejidad del código y documentación

El análisis del código fue muy complicado debido a la falta de documentación y desorganización del código. Además se nota que el recolector ha sido escrito en una fase muy temprana y que a ido evolucionando a partir de ello de forma descuidada y sin ser rescrito nunca para aprovechar las nuevas características que el lenguaje fue incorporando (por ejemplo *templates*).

Estos dos problemas (código complicado y falta de documentación) producen un efecto de círculo vicioso, porque provocan que sea complejo entender el recolector actual y en consecuencia sea muy complicado escribir documentación o mejorarlo. Esto a su vez provoca que, al no disponer de una implementación de referencia sencilla, sea muy difícil implementar un recolector nuevo.

Este es, probablemente, la raíz de todos los demás problemas del recolector actual. Para ilustrar la dimensión del problema se presenta la implementación real de la función `bigAlloc()`:

```
/**
 * Allocate a chunk of memory that is larger than a page.
 * Return null if out of memory.
 */
void *bigAlloc(size_t size)
{
    Pool*  pool;
```

```
    size_t npages;
    size_t n;
    size_t pn;
    size_t freedpages;
    void*  p;
    int    state;

    npages = (size + PAGESIZE - 1) / PAGESIZE;

    for (state = 0; ; )
    {
        // This code could use some refinement when repeatedly
        // allocating very large arrays.

        for (n = 0; n < npools; n++)
        {
            pool = pooltable[n];
            pn = pool.allocPages(npages);
            if (pn != OPFAIL)
                goto L1;
        }

        // Failed
        switch (state)
        {
        case 0:
            if (disabled)
            {   state = 1;
                continue;
            }
            // Try collecting
            freedpages = fullcollectshell();
            if (freedpages >= npools * ((POOLSIZE / PAGESIZE) / 4))
            {   state = 1;
                continue;
            }
            // Release empty pools to prevent bloat
            minimize();
            // Allocate new pool
            pool = newPool(npages);
            if (!pool)
            {   state = 2;
                continue;
            }
            pn = pool.allocPages(npages);
            assert(pn != OPFAIL);
            goto L1;
        case 1:
            // Release empty pools to prevent bloat
            minimize();
            // Allocate new pool
            pool = newPool(npages);
            if (!pool)
                goto Lnomemory;
            pn = pool.allocPages(npages);
            assert(pn != OPFAIL);
            goto L1;
```

```
        case 2:
            goto Lnomemory;
        default:
            assert(false);
        }
    }

  L1:
    pool.pagetable[pn] = B_PAGE;
    if (npages > 1)
        cstring.memset(&pool.pagetable[pn + 1], B_PAGEPLUS, npages - 1);
    p = pool.baseAddr + pn * PAGESIZE;
    cstring.memset(cast(char *)p + size, 0, npages * PAGESIZE - size);
    debug (MEMSTOMP) cstring.memset(p, 0xF1, size);
    //debug(PRINTF) printf("\tp = %x\n", p);
    return p;

  Lnomemory:
    return null; // let mallocNoSync handle the error
}
```

Se recuerda que la semántica de dicha función es la misma que la de la función `new_big()` presentada en *Asignación de memoria*.

Además, como se comentó en la sección anterior, los algoritmos en la implementación real son considerablemente menos modulares que los presentados en la sección *Algoritmos del recolector*. Por ejemplo, la función `fullcollect()` tiene 300 líneas de código fuente.

Memoria *encomendada*

Como se comentó en la sección anterior, diferenciar entre memoria *encomendada* de memoria *no-encomendada* es complejo y levemente costoso (en particular para sistemas operativos que no hacen esta distinción, al menos explícitamente, donde no hay ningún beneficio en realizar esta distinción).

Incluso para Microsoft Windows, la ventaja de realizar esta distinción debería ser comprobada.

Precisión

Este fue históricamente uno de los problemas principales del recolector de D [NGD46407] [NGD35364]. Sin embargo, desde que, en la versión 1.001, se ha incorporado la capacidad de marcar un bloque como de datos puros (no contiene punteros, el atributo `NO_SCAN`) [NGA6842], la gravedad de esos problemas ha disminuido considerablemente, aunque siguieron reportándose problemas más esporádicamente [NGD54084] [NGL13744].

De todas maneras queda mucho lugar para mejoras, y es un tema recurrente en el grupo de noticias de D y se han discutido formas de poder hacer que, al menos el *heap* sea preciso [NGD44607] [NGD29291]. Además se mostró un interés general por tener un recolector más preciso [NGD87831], pero no han habido avances al respecto hasta hace muy poco tiempo.

Otra forma de minimizar los efectos de la falta de precisión que se ha sugerido reiteradamente en el grupo es teniendo la posibilidad de indicar cuando no pueden haber punteros interiores a un bloque [NGD89394] [NGD71869]. Esto puede ser de gran utilidad para objetos grandes y en particular para mejorar la implementación de de arreglos asociativos.

Referencias débiles

Si bien el recolector de Tango tiene un soporte limitado de *referencias débiles* [7], el de Phobos no dispone de ningún soporte (por lo tanto no está contemplado oficialmente el lenguaje). Sin embargo hay una demanda apreciable [NGD86840] [NGD13301] [NGL8264] [NGD69761] [NGD74624] [NGD88065].

Para cubrir esta demanda, se han implementado soluciones como biblioteca para suplir la inexistencia de una implementación oficial [NGA9103] (la implementación de Tango es otro ejemplo).

Sin embargo éstas son en general poco robustas, extremadamente dependientes de la implementación del recolector y, en general, presentan problemas muy sutiles [NGD88065]. Por esta razón se ha discutido la posibilidad de incluir la implementación de *referencias débiles* como parte del lenguaje [NGD88559].

Concurrencia

El soporte actual de concurrencia, en todos sus aspectos, es muy primitivo. El recolector apenas soporta múltiples *mutators* pero con un nivel de sincronización excesivo.

Se ha sugerido en el pasado el uso de *pools* y listas de libres específicos de hilos, de manera de disminuir la contención, al menos para la asignación de memoria [NGD75952] [NGD87831].

Además se ha mostrado un interés por tener un nivel de concurrencia aún mayor en el recolector, para aumentar la eficiencia en ambientes *multi-core* en general pero en particular para evitar grandes pausas en programas con requerimientos de tiempo real, históricamente una de las principales críticas al lenguaje [NGD87831] [NGL3937] [NGD22968] [NGA15246] [NGD5622] [NGD2547] [NGD18354].

Finalización

El recolector actual no garantiza la finalización de objetos. En particular los objetos no son finalizados (es decir, no se llama a sus destructores) si aún alcanzables desde el *root set* cuando el programa termina. Cabe destacar que esto puede darse porque hay una referencia real desde el *root set* (en cuyo caso queda bajo el control del usuario) pero también, dado que el *root set* se visita de forma conservativa, se puede deber a un *falso positivo*, en cuyo caso la omisión de la finalización queda por completo fuera del control del usuario (y lo que es aún peor, el usuario no puede ser siquiera notificado de esta anomalía).

Si bien la especificación de D no requiere esta capacidad, no hay mayores problemas para implementar un recolector que dé este tipo de garantías [NGD88298].

Además los objetos pueden ser finalizados tanto determinísticamente (utilizando `delete` o `scope`; ver secciones *Programación de bajo nivel (system programming)* y *Programación confiable*) como no determinísticamente (cuando son finalizados por el recolector). En el primer caso se puede, por ejemplo, acceder sus atributos u otra memoria que se conozca *viva*, mientras que en el segundo no. Sin embargo un destructor no puede hacer uso de esta distinción, haciendo que la finalización determinística tenga a fines prácticos las mismas restricciones que la finalización no determinística. Es por esto que se ha sugerido permitir al destructor distinguir estos dos tipos de finalización [NGD89302].

Eficiencia

El rendimiento en general del recolector es una de las críticas frecuentes. Si bien hay muchos problemas que han sido resueltos, en especial por la inclusión de un mínimo grado de precisión en la versión 1.001,

[7] Una referencia débil (o *weak reference* en inglés) es aquella que que no protege al objeto referenciado de ser reciclado por el recolector.

en la actualidad se siguen encontrando en el grupo de noticias críticas respecto a esto [NGD43991] [NGD67673] [NGD63541] [NGD90977].

La principal causa del bajo rendimiento del recolector actual es, probablemente, lo simple de su algoritmo principal de recolección. Más allá de una organización del *heap* moderadamente apropiada y de utilizar conjuntos de bits para la fase de marcado, el resto del algoritmo es casi la versión más básica de marcado y barrido. Hay mucho lugar para mejoras en este sentido.

Configurabilidad

Si bien el recolector actual tiene algunas características configurables, todas son seleccionables sólo en tiempo de compilación del recolector (no del programa del usuario), como por ejemplo las opciones descriptas en *Herramientas para depuración*. Por lo tanto, a nivel práctico, es como si no tuviera posibilidad alguna de ser configurado por el usuario, ya que no es parte del ciclo de desarrollo normal el recompilar el recolector o *runtime* de un lenguaje.

Dado que es imposible que un recolector sea óptimo para todo tipo de programas, es muy deseable permitir una configuración de parámetros del recolector que permitan al usuario ajustarlos a las necesidades particulares de sus aplicaciones.

Factor de ocupación del *heap*

Otro problema potencialmente importante del recolector actual es que no se tiene ningún cuidado con respecto a que, luego de una recolección, se haya recuperado una buena parte del *heap*. Por lo tanto, en casos extremos, el recolector tiene que hacer una recolección por cada petición de memoria, lo que es extremadamente ineficiente.

Para evitar esto, habría que usar algún esquema para evaluar cuando una recolección no fue lo suficientemente *exitosa* y en ese caso pedir más memoria al sistema operativo.

Detalles

Finalmente hay varios detalles en la implementación actual que podrían mejorarse:

Listas de libres
: Hay 12 listas de libres, como para guardar bloques de tamaño de `B_16` a `B_2048`, `B_PAGE`, `B_PAGEPLUS`, `B_UNCOMMITTED` y `B_FREE`; sin embargo solo tienen sentido los bloques de tamaño `B_16` a `B_2048`, por lo que 4 de esas listas no se utilizan.

Conjuntos de bits para indicadores
: Los indicadores para la fase de marcado y otras propiedades de un bloque son almacenados en conjuntos de bits que almacenan los indicadores de todos los bloques de un *pool*. Si bien se ha mencionado esto como una ventaja, hay lugar todavía como para algunas mejoras. Como un *pool* tiene páginas con distintos tamaños de bloque, se reserva una cantidad de bits igual a la mayor cantidad posible de bloques que puede haber en el *pool*; es decir, se reserva 1 bit por cada 16 bytes del *pool*. Para un *pool* de 1 MiB (tamaño mínimo), teniendo en cuenta que se utilizan 5 conjuntos de bits (`mark`, `scan`, `finals`, `freebits` y `noscan`), se utilizan 40 KiB de memoria para conjuntos de bits (un 4 % de *desperdicio* si, por ejemplo, ese *pool* estuviera destinado por completo a albergar un solo objeto grande; lo que equivaldría al 2560 objetos de 16 bytes desperdiciados en bits inutilizados).

Repetición de código
Hay algunos fragmentos de código repetidos innecesariamente. Por ejemplo en varios lugares se utilizan arreglos de tamaño variable que se implementan repetidas veces (en general como un puntero al inicio del arreglo más el tamaño actual del arreglo más el tamaño de la memoria total asignada actualmente). Esto es propenso a errores y difícil de mantener.

Uso de señales
El recolector actual utiliza las señales del sistema operativo `SIGUSR1` y `SIGUSR2` para pausar y reanudar los hilos respectivamente. Esto puede traer inconvenientes a usuarios que desean utilizar estas señales en sus programas (o peor aún, si interactúan con bibliotecas de C que hacen uso de estas señales) [NGD5821].

Marcado iterativo
Si bien esto se mencionó como algo bueno del recolector actual, es un compromiso entre tiempo y espacio, y puede ser interesante analizar otros métodos para evitar la recursión que no requieran tantas pasadas sobre el *heap*.

4.3 Análisis de viabilidad

Ya conociendo el lenguaje de programación D (con sus necesidades particulares), el estado del arte en recolección de basura y el recolector actual de D es posible evaluar la viabilidad de los distintos algoritmos vistos en el capítulo *Recolección de basura*. Se recuerda que dentro del análisis de viabilidad de considera de gran importancia la viabilidad social y política de la mejora, es decir, se presta particular atención en encontrar una mejora que tenga una buena probabilidad de ser aceptada por la comunidad de D.

4.3.1 Algoritmos clásicos

En esta sección se presenta un análisis de los *algoritmos clásicos*, de forma de poder analizar a grandes rasgos las principales familias para ir determinando la dirección principal de la solución.

Conteo de referencias

Ya se ha propuesto en el pasado la utilización de conteo de referencias en D pero no se ha demostrado un interés real, más allá de soluciones en bibliotecas [NGD38689]. Las razones para no utilizar conteo de referencia son más o menos las mismas que las desventajas mencionadas en la sección *Conteo de referencias* (en el capítulo *Recolección de basura*), siendo la principal la incapacidad de recolectar ciclos. Sin embargo hay otras razones importantes.

Una de ellas es la inter-operatividad con C. El utilizar un contador de referencias requiere la manipulación del contador por parte del código C con el que se interactúe. Si bien este problema ya está presente si código C guarda en su *headp* un puntero a un objeto almacenado en el *heap* del recolector de D, esto es poco común. Sin embargo, mientras que una función de C se está ejecutando, es extremadamente común que pueda almacenar en el *stack* una referencia a un objeto de D y en ese caso el recolector actual puede manejarlo (mientras la función de C esté corriendo en un hilo creado por D). Sin embargo al usar un conteo de referencias esto es más problemático, ya que no se mantiene la invariante del algoritmo si no son actualizados siempre los contadores.

Otro problema es que al liberarse una celda, existe la posibilidad de tener que liberar todo el sub-grafo conectado a ésta. Cuando este sub-grafo es grande, se puede observar una gran pausa.

Si bien estas razones son suficientes como para considerar que el conteo de referencias no es un algoritmo que sea viable en D, hay muchas técnicas y optimizaciones para minimizarlas (como liberación perezosa, conteo de referencias pospuesto, etc. [JOLI96]). Sin embargo hay otra razón importante que descarta esta familia de algoritmos ya que todas las variaciones de conteo de referencias implican, en mayor o menor medida, el entrelazado del trabajo del recolector con el del *mutator*. Si bien esta es una característica en general muy deseable (porque hace que el recolector sea *incremental*), en D no lo es porque tiene como requerimiento no hacer pagar el precio de cosas que no se usan. En D debe ser posible no utilizar el recolector de basura y, al no hacerlo, no tener ningún tipo de trabajo extra asociado a éste. De usarse conteo de referencias esto no sería posible.

Si bien este requerimiento puede ser discutible técnicamente, hay una gran resistencia social y política ante cualquier tipo de recolector que imponga una penalización de rendimiento a alguien que no quiera usarlo [NGD38689]. Además requiere un cambio complejo y profundo en el compilador, siendo éste uno de los eslabones con mayor resistencia a introducir cambios.

Por lo tanto se concluye que el conteo de referencias no es un algoritmo viable para este trabajo.

Marcado y barrido

El marcado y barrido es un algoritmo evidentemente viable debido a que es la base del algoritmo del recolector de basura actual.

En general en la comunidad de D no hay mayores críticas al marcado y barrido en sí, si no más bien a problemas asociados a la implementación actual, principalmente a las grandes pausas o la falta de *precisión* [NGD54084] [NGL13744] [NGD44607] [NGD29291] [NGD87831] [NGD87831] [NGL3937] [NGD22968] [NGA15246] [NGD5622] [NGD2547] [NGD18354].

Esta familia de algoritmos se adapta bien a los requerimientos principales de D en cuanto a recolección de basura (ver *Características y necesidades particulares de D*), por ejemplo permite recolectar de forma conservativa, no impone un *overhead* a menos que se utilice el recolector, permite liberar memoria manualmente, se adapta de forma simple para soportar punteros *interiores* y permite finalizar objetos (con las limitaciones mencionadas en *Orientación a objetos y finalización*).

Sin embargo muchas de las limitaciones del recolector actual (ver *Problemas y limitaciones*), no son inherentes al marcado y barrido, por lo que aún conservando la base del algoritmo, es posible realizar una cantidad de mejoras considerable.

Una de las principales mejoras que pueden realizarse es hacer al recolector *concurrente* y más *preciso*. Estas dos mejoras solamente alcanzarían para mejorar de forma notable el tiempo de pausa en las recolecciones y la cantidad de memoria retenida debido a *falsos positivos*.

Más adelante veremos detalles sobre algunos de estos aspectos y sobre algunos algoritmos particulares que permiten hacer concurrente al recolector actual.

Copia de semi-espacio

La copia de semi-espacio, al igual que cualquier otro tipo de recolector con movimiento, requiere (en la mayoría de los casos) disponer de una *precisión* casi completa. Las celdas para las cuales hay alguna referencia que no es precisa no pueden ser movidas, ya que al no estar seguros que la referencia sea tal, ésta no puede ser actualizada con la dirección de la nueva ubicación de la celda movida porque de no ser una referencia se estarían alterando datos del usuario, corrompiéndolos.

Es por esto que si el recolector no es mayormente preciso, las celdas que pueden ser movidas son muy pocas y, por lo tanto, se pierden las principales ventajas de esta familia de recolectores (como la

capacidad de asignar nueva memoria mediante *pointer bump allocation*).

Este aumento de precisión, sin embargo, es bastante realizable. Es posible, en teoría, hacer que al menos el *heap* sea preciso, aunque es discutible si en la práctica es aceptable el *overhead* en espacio necesario para almacenar la información del tipo de una celda. Esto se analiza en más detalle al evaluar la recolección precisa en la siguiente sección.

Si bien las principales herramientas para que sea viable un recolector por copia de semi-espacio están disponibles en D (como la posibilidad de hacer *pinning* the celdas o el potencial incremento de precisión), este lenguaje nunca va a poder proveer precisión total, haciendo que no sea posible implementar un recolector por copia de semi-espacio puro. Siempre habrá que disponer un esquema híbrido para poder manejar las celdas que no puedan moverse, incrementado mucho la complejidad del recolector.

Si bien un esquema híbrido es algo técnicamente posible, nuevamente la resistencia social a un cambio de esta envergadura es de importancia suficiente como para inclinarse por una solución menos drástica.

4.3.2 Principales categorías del estado del arte

En esta sección se realiza un análisis de la viabilidad de las principales categorías de recolectores según se presentaron en la sección *Estado del arte*.

Recolección directa / indirecta

Como se ha visto al analizar el conteo de referencias, lo más apropiado para D pareciera ser continuar con el esquema de recolección indirecta, de forma tal de que el precio de la recolección solo deba ser pagado cuando el *mutator* realmente necesita del recolector. Es por esto que no parece ser una opción viable introducir recolección directa en este trabajo.

Recolección incremental

La recolección incremental puede ser beneficiosa para D, dado que puede servir para disminuir el tiempo de pausa del recolector. Sin embargo, en general es necesario instrumentar el *mutator* para reportar cambios en el grafo del conectividad al recolector. Además puede contar con los mismos problemas que la recolección directa, puede hacer que el usuario tenga que pagar el precio de la recolección, incluso cuando no la necesita, si por cada asignación el recolector realiza parte de una recolección que no fue solicitada.

Recolección concurrente / paralela / *stop-the-world*

El recolector actual es *stop-the-world*, sin embargo esta es una de las principales críticas que tiene. El recolector se podría ver beneficiado de recolección paralela, tanto para realizar la recolección más velozmente en ambientes *multi-core*, como para disminuir el tiempo de pausa, un factor muy importante para programas que necesiten tener baja latencia, como programas *real-time*.

En general los recolectores concurrentes necesitan también instrumentar el *mutator* para reportar cambios en el grafo de conectividad al recolector, como sucede con la recolección directa o incremental, sin embargo hay algoritmos que no tienen este requerimiento, utilizando servicios del sistema operativo para tener una *fotografía* de la memoria para que la fase de marcado pueda realizarse sin perturbar al *mutator* ni requerir de su cooperación [RODR97]. Este tipo de algoritmos serían un buen candidato para D, dado que requiere pocos cambios y es transparente al *mutator*.

Recolección conservativa / precisa

Si bien D puede proveer al recolector de basura información de tipos para los objetos almacenados en el *heap*, todo recolector para D deberá soportar cierto grado de recolección conservativa (ver *Recolectores conservativos versus precisos*), debido a las siguientes razones:

- Si bien D podría incorporar información de tipos para el *stack* (utilizando, por ejemplo, la técnica de *shadow stack* [HEND02]), para poder interactuar con C/C++, el recolector debe poder interpretar los *stack frames* [8] de estos lenguajes, que no disponen de información de tipos.
- Los registros del procesador tienen un problema similar, con la diferencia de que el costo de implementar algo similar a *shadow stack* para los registros sería impracticable, más allá de que exista la misma limitación que con el *stack* para poder interactuar con C/C++.
- D soporta uniones (ver *Programación de bajo nivel (system programming)*). Para una unión es imposible determinar si un campo es un puntero o no. Por ejemplo:

```
union U {
   size_t x;
   void* p;
}
```

  Aquí el recolector no puede saber nunca si el valor almacenado será un `size_t` o un `void*`, por lo tanto deberá tratar **siempre** esa palabra de forma conservativa (es decir, interpretarla como un *posible* puntero). Este requerimiento puede ser relajado si el usuario proveyera alguna forma de determinar que tipo está almacenando la unión en un determinado momento. Sin embargo el costo de pedir al usuario este tipo de restricción puede ser muy alto.

Durante el desarrollo de este trabajo se encontra un trabajo relacionado avanzando en este sentido, que agrega precisión al marcado del *heap*. David Simcha comienza explorando la posibilidad de agregar precisión parcial al recolector, generando información sobre la ubicación de los punteros para cada tipo [DBZ3463]. Su trabajo se limita a una implementación a nivel biblioteca de usuario y sobre D 2.0. Desafortunadamente su trabajo pasa desapercibido por un buen tiempo.

Sin embargo un tiempo después Vincent Lang (mejor conocido como *wm4* en la comunidad de D), retoma este trabajo, pero modificando el compilador DMD y trabajando con D 1.0 y Tango. Es por esto que el aumento de precisión parece ser un área fértil para este trabajo, en particular si se colabora con el trabajo realizado por David y Vincent.

Recolección con movimiento de celdas

Esta posibilidad ya se ha discutido al analizar la posibilidad de utilizar recolección con copia de semi-espacios. El trabajo mencionado en la sub-sección anterior agrega información suficiente como poder diferenciar que celdas se pueden mover y cuales no, sin embargo queda como incógnita qué proporción de celdas deben permanecer inmovilizadas como para evaluar si un cambio tan grande puede rendir frutos o no.

A priori, pareciera que la relación cantidad y complejidad de cambios sobre beneficios potenciales no fuera muy favorable a esta mejora.

[8] Un *stack frame* (*marco de la pila* en castellano), también conocido como *activation record* (o *registro de activación* en castellano) es una estructura de datos dependiente de la arquitectura que contiene información del estado de una función, incluyendo, por ejemplo, sus variables locales, parámetros y dirección de retorno.

Lista de libres / *pointer bump allocation*

Como consecuencia de los puntos anteriores, no es técnicamente posible realizar *pointer bump allocation* pura en D. Al haber objetos *pinned*, siempre es necesario o bien contar con una lista de libres, o detectar *huecos* en un esquema de *pointer bump allocation*. Es por esto que parece ser más viable conservar el esquema de listas de libres.

Esta mejora también entra en la categoría de opciones viables pero cuya complejidad no parece valer la pena dada la limitada utilidad que se espera dadas las particulares características de D en cuanto a precisión de información de tipos de *stack*, uniones, etc.

Recolección por particiones / generacional

Una vez más la recolección por particiones, en particular la generacional, requiere de la instrumentación del *mutator* para comunicar cambios en el grafo de conectividad al recolector, por lo que es poco viable. Aunque existen algoritmos que no necesitan este tipo de comunicación dado que está garantizado que no existan conexiones entre celdas de las distintas particiones, requiere grandes cambios en el compilador y realizar análisis estático bastante complejo [HIRZ03]. Además al ser D un lenguaje de bajo nivel, es muy difícil garantizar que estas conexiones inter-particiones no puedan existir realmente; y de hacerlo, podría ser demasiado restrictivo.

CAPÍTULO 5

Solución adoptada

Como hemos visto en *Recolección de basura en D*, la mejora del recolector de basura puede ser abordada desde múltiples flancos, con varias alternativas viables. Por lo tanto, para reducir la cantidad de posibilidades hay que tener en cuenta uno de los principales objetivos de este trabajo: encontrar una solución que tenga una buena probabilidad de ser adoptada por el lenguaje, o alguno de sus compiladores al menos. Para asegurar esto, la solución debe tener un alto grado de aceptación en la comunidad, lo que implica algunos puntos claves:

- La eficiencia general de la solución no debe ser notablemente peor, en ningún aspecto, que la implementación actual.
- Los cambios no deben ser drásticos.
- La solución debe atacar de forma efectiva al menos uno de los problemas principales del recolector actual.

Bajo estos requerimientos, se concluye que probablemente el área más fértil para explorar sea la falta de concurrencia por cumplir todos estos puntos:

- Si bien hay evidencia en la literatura sobre el incremento del tiempo de ejecución total de ejecución de un programa al usar algoritmos concurrentes, éste no es, en general, muy grande comparativamente.
- Existen algoritmos de recolección concurrente que no requieren ningún grado de cooperación por parte del lenguaje o el compilador.
- La falta de concurrencia y los largos tiempos de pausa es una de las críticas más frecuentes al recolector actual por parte de la comunidad.

A pesar de ser la concurrencia la veta principal a explorar en este trabajo, se intenta abordar los demás problemas planteados siempre que sea posible hacerlo sin alejarse demasiado del objetivo principal.

5.1 Banco de pruebas

Teniendo en cuenta que uno de los objetivos principales es no empeorar la eficiencia general de forma notable, la confección de un banco de pruebas es un aspecto fundamental, para poder comprobar con cada cambio que la eficiencia final no se vea notablemente afectada.

La confección de un banco de pruebas no es una tarea trivial, mucho menos para un lenguaje con el nivel de fragmentación que tuvo D (que hace que a fines prácticos hayan 3 versiones del lenguaje com-

pitiendo), y cuya masa crítica de usuarios es de aficionados que usualmente abandonan los proyectos, quedando obsoletos rápidamente.

Con el objetivo de confeccionar este banco de pruebas, desde el comienzo del trabajo se han recolectado (usando como fuente principalmente el grupo de noticias de D [1]) programas triviales sintetizados con el único propósito de mostrar problemas con el recolector de basura. Otros programas de este estilo fueron escritos explícitamente para este trabajo.

Además se han recolectado algunos pequeños programas portados de otros lenguajes de programación, que si bien son pequeños y tienen como objetivo ejercitar el recolector de basura, son programas reales que resuelven un problema concreto, lo que otorga un juego de pruebas un poco más amplio que los programas triviales.

Pero probablemente lo más importante para confeccionar un banco de pruebas verdaderamente útil es disponer de programas reales, que hayan sido diseñados con el único objetivo de hacer su trabajo, sin pensar en como impacta el recolector sobre ellos (ni ellos sobre el recolector). Estos programas proveen las pruebas más realistas y amplias. Desgraciadamente no hay muchos programas reales escritos en D disponibles públicamente, y no se encontró en la comunidad tampoco una muestra de voluntad por compartir programas privados para usar como banco de pruebas en este trabajo.

Por lo tanto el banco de pruebas que se conformó como una mezcla de estas tres grandes categorías.

5.1.1 Pruebas sintetizadas

Este es el juego de programas triviales, escritos con el único objetivo de ejercitar un área particular y acotada del recolector.

bigarr

Su objetivo es ejercitar la manipulación de arreglos de tamaño considerable que almacenan objetos de tamaño pequeño o mediano. Esta prueba fue hallada en el grupo de noticias de D y escrita por Babele Dunnit y aunque originalmente fue concebido para mostrar un problema con la concatenación de arreglos (como se aprecia en la sentencia `version(loseMemory)`), ejercita los aspectos más utilizados del del recolector: manipulación de arreglos y petición e memoria. Es una de las pruebas que más estresa al recolector ya que todo el trabajo que realiza el programa es utilizar sus servicios.

Código fuente:

```
const IT = 300;
const N1 = 20_000;
const N2 = 40_000;

class Individual
{
   Individual[20] children;
}

class Population
{
   void grow()
```

[1] Cabe destacar que en general todos los programas recolectados han sido modificados levemente para ajustarlos mejor a las necesidades del banco de prueba (entre las modificaciones más frecuentes se encuentran la conversión de Phobos a Tango y la eliminación de mensajes por salida estándar).

```
    {
        foreach (inout individual; individuals)
            individual = new Individual;
    }
    Individual[N1] individuals;
}

version = loseMemory;

int main(char[][] args)
{
    Population testPop1 = new Population;
    Population testPop2 = new Population;
    Individual[N2] indi;
    for (int i = 0; i < IT; i++) {
        testPop1.grow();
        testPop2.grow();
        version (loseMemory) {
            indi[] = testPop1.individuals ~ testPop2.individuals;
        }
        version (everythingOk) {
            indi[0 .. N1] = testPop1.individuals;
            indi[N1 .. N2] = testPop2.individuals;
        }
    }
    return 0;
}
```

concpu y conalloc

Estos dos programas fueron escritos especialmente para este trabajo con el fin de ejercitar la interacción entre el recolector y un *mutator* con varios hilos. La única diferencia entre ellos es que `concpu` lanza hilos que hacen trabajar de forma intensiva el procesador pero que no utilizan servicios del recolector, salvo en el hilo principal, mientras que `conalloc` utiliza servicios del recolector en todos los hilos lanzados.

El objetivo de estos programas es medir el impacto de las pausas del recolector. Se espera medir dos tipos de pausa principales, por un lado el tiempo máximo de pausa real, que puede involucrar a más de un hilo y por otro el tiempo de *stop-the-world*, es decir, el tiempo en que los hilos son efectivamente pausados por el recolector para realizar una tarea que necesite trabajar con una versión estática de la memoria del programa.

Se espera `concpu` sea capaz de explotar cualquier reducción en el tiempo de *stop-the-world*, ya que los hilos solo son interrumpidos por este tipo de pausa. Por otro lado, se espera que `conalloc` sea afectado por el tiempo máximo de pausa, que podrían sufrir los hilos incluso cuando el *mundo* sigue su marcha, debido al *lock* global del recolector y que los hilos usan servicios de éste.

Código fuente de `concpu`:

```
import tango.core.Thread: Thread;
import tango.core.Atomic: Atomic;
import tango.io.device.File: File;
import tango.util.digest.Sha512: Sha512;
import tango.util.Convert: to;
```

```
auto N = 100;
auto NT = 2;
ubyte[] BYTES;
Atomic!(int) running;

void main(char[][] args)
{
   auto fname = args[0];
   if (args.length > 3)
      fname = args[3];
   if (args.length > 2)
      NT = to!(int)(args[2]);
   if (args.length > 1)
      N = to!(int)(args[1]);
   N /= NT;
   running.store(NT);
   BYTES = cast(ubyte[]) File.get(fname);
   auto threads = new Thread[NT];
   foreach(ref thread; threads) {
      thread = new Thread(&doSha);
      thread.start();
   }
   while (running.load()) {
      auto a = new void[](BYTES.length / 4);
      a[] = cast(void[]) BYTES[];
      Thread.yield();
   }
   foreach(thread; threads)
      thread.join();
}

void doSha()
{
   auto sha = new Sha512;
   for (size_t i = 0; i < N; i++)
      sha.update(BYTES);
   running.decrement();
}
```

El código de `conalloc` es igual excepto por la función `doSha()`, que es de la siguiente manera:

```
void doSha()
{
   for (size_t i = 0; i < N; i++) {
      auto sha = new Sha512;
      sha.update(BYTES);
   }
   running.decrement();
}
```

mcore

Escrito por David Schima y también hallado en el grupo de noticias de D, este programa pretende mostrar como afecta el *lock* global del recolector en ambientes *multi-core*, incluso cuando a simple vista parecen no utilizarse servicios del recolector:

```
import tango.core.Thread;

void main()
{
   enum { nThreads = 4 };
   auto threads = new Thread[nThreads];
   foreach (ref thread; threads) {
      thread = new Thread(&doAppending);
      thread.start();
   }
   foreach (thread; threads)
      thread.join();
}

void doAppending()
{
   uint[] arr;
   for (size_t i = 0; i < 1_000_000; i++)
      arr ~= i;
}
```

El secreto está en que la concatenación de arreglos utiliza por detrás servicios del recolector, por lo tanto un programa multi-hilo en el cual los hilos (aparentemente) no comparten ningún estado, se puede ver considerablemente afectado por el recolector (siendo este efecto más visible en ambientes *multi-core* por el nivel de sincronización extra que significa a nivel de *hardware*). Cabe destacar, sin embargo, que en Linux el efecto no es tan notorio comparado al reporte de David Schima.

split

Este programa trivial lee un archivo de texto y genera un arreglo de cadenas de texto resultantes de partir el texto en palabras. Fue escrito por Leonardo Maffi y también hallado en el grupo de noticias de D. Su objetivo era mostrar lo ineficiente que puede ser concatenar datos a un mismo arreglo repetidas veces y ha desembocado en una pequeña optimización que sirvió para paliar el problema de forma razonablemente efectiva [PAN09].

Código fuente:

```
import tango.io.device.File: File;
import tango.text.Util: delimit;
import tango.util.Convert: to;

int main(char[][] args) {
   if (args.length < 2)
      return 1;
   auto txt = cast(byte[]) File.get(args[1]);
   auto n = (args.length > 2) ? to!(uint)(args[2]) : 1;
   if (n < 1)
      n = 1;
   while (--n)
      txt ~= txt;
   auto words = delimit!(byte)(txt, cast(byte[]) " \t\n\r");
   return !words.length;
}
```

rnddata

Este programa fue escrito por Oskar Linde y nuevamente hallado en el grupo de noticias. Fue construido para mostrar como el hecho de que el recolector sea conservativo puede hacer que al leer datos binarios hayan muchos *falsos positivos* que mantengan vivas celdas que en realidad ya no deberían ser accesibles desde el *root set* del grafo de conectividad.

Código fuente:

```
import tango.math.random.Random;

const IT = 125; // number of iterations, each creates an object
const BYTES = 1_000_000; // ~1MiB per object
const N = 50; // ~50MiB of initial objects

class C
{
   C c; // makes the compiler not set NO_SCAN
   long[BYTES/long.sizeof] data;
}

void main() {
   auto rand = new Random();
   C[] objs;
         objs.length = N;
   foreach (ref o; objs) {
      o = new C;
      foreach (ref x; o.data)
         rand(x);
   }
   for (int i = 0; i < IT; ++i) {
      C o = new C;
      foreach (ref x; o.data)
         rand(x);
      // do something with the data...
   }
}
```

sbtree

Este programa está basado en la prueba de nombre `binary-trees` de The Computer Language Benchmarks Game, una colección de 12 programas escritos en alrededor de 30 lenguajes de programación para comparar su eficiencia (medida en tiempo de ejecución, uso de memoria y cantidad de líneas de código) [SHO10]. De este juego de programas se utilizó solo `binary-trees` por ser el único destinado a ejercitar el manejo de memoria. El programa sólo manipula árboles binarios, creándolos y recorriéndolos inmediatamente (no realiza ningún trabajo útil). La traducción a D fue realizada por Andrey Khropov y fue hallada en el grupo de noticias.

Código fuente:

```
import tango.util.Convert;
alias char[] string;

int main(string[] args)
{
```

```
   int N = args.length > 1 ? to!(int)(args[1]) : 1;
   int minDepth = 4;
   int maxDepth = (minDepth + 2) > N ? minDepth + 2 : N;
   int stretchDepth = maxDepth + 1;
   int check = TreeNode.BottomUpTree(0, stretchDepth).ItemCheck;
   TreeNode longLivedTree = TreeNode.BottomUpTree(0, maxDepth);
   for (int depth = minDepth; depth <= maxDepth; depth += 2) {
      int iterations = 1 << (maxDepth - depth + minDepth);
      check = 0;
      for (int i = 1; i <= iterations; i++) {
         check += TreeNode.BottomUpTree(i, depth).ItemCheck;
         check += TreeNode.BottomUpTree(-i, depth).ItemCheck;
      }
   }
   return 0;
}

class TreeNode
{
   TreeNode left, right;
   int item;

   this(int item, TreeNode left = null, TreeNode right = null)
   {
      this.item = item;
      this.left = left;
      this.right = right;
   }

   static TreeNode BottomUpTree(int item, int depth)
   {
      if (depth > 0)
         return new TreeNode(item,
               BottomUpTree(2 * item - 1, depth - 1),
               BottomUpTree(2 * item, depth - 1));
      return new TreeNode(item);
   }

   int ItemCheck()
   {
      if (left)
         return item + left.ItemCheck() - right.ItemCheck();
      return item;
   }
}
```

5.1.2 Programas pequeños

Todos los pequeños programas utilizados como parte del banco de prueba provienen del Olden Benchmark [CAR95]. Estos programas fueron diseñados para probar el lenguaje de programación Olden; un lenguaje diseñado para paralelizar programas automáticamente en arquitecturas con memoria distribuida. Son programas relativamente pequeños (entre 400 y 1000 líneas de código fuente cada uno) que realizan una tarea secuencial que asigna estructuras de datos dinámicamente. Las estructuras están usualmente organizadas como listas o árboles, y muy raramente como arreglos. Los programas pasan la mayor parte del tiempo solicitando memoria para almacenar datos y el resto usando los datos almacena-

dos, por lo que en general están acotados en tiempo por el uso de memoria (y no de procesador).

La traducción a D fue realizada por Leonardo Maffi y están basadas a su vez en la traducción de este juego de pruebas a Java, JOlden [CMK01]. En Java no se recomienda utilizar este conjunto de pruebas para medir la eficiencia del recolector de basura, dado que se han creado mejores pruebas para este propósito, como DaCapo [BLA06], sin embargo, dada la falta de programas disponibles en general, y de un conjunto de pruebas especialmente diseñado para evaluar el recolector de basura en D, se decide utilizarlas en este trabajo de todos modos. Sin embargo sus resultados deben ser interpretados con una pizca de suspicacia por lo mencionado anteriormente.

En general (salvo para el programa `voronoï`) está disponible el código fuente portado a D, Java y Python, e incluso varias versiones con distintas optimizaciones para reducir el consumo de tiempo y memoria. Además provee comparaciones de tiempo entre todas ellas. Los programas utilizados en este banco de pruebas son la versión traducida más literalmente de Java a D, ya que hace un uso más intensivo del recolector que las otras versiones.

A continuación se da una pequeña descripción de cada uno de los 5 programas traducidos y los enlaces en donde encontrar el código fuente (y las comparaciones de tiempos estar disponibles).

bh

Este programa computa las interacciones gravitatorias entre un número N de cuerpos en tiempo $O(NlogN)$ y está basado en árboles heterogéneos de 8 ramas, según el algoritmo descripto por Barnes & Hut [BH86].

Código fuente disponible en: http://www.fantascienza.net/leonardo/js/dolden_bh.zip

bisort

Este programa ordena N números, donde N es una potencia de 2, usando un ordenamiento *Bitonic* adaptativo, un algoritmo paralelo óptimo para computadoras con memoria compartida, según describen Bilardi & Nicolau [BN98]. Utiliza árboles binarios como principal estructuras de datos.

Código fuente disponible en: http://www.fantascienza.net/leonardo/js/dolden_bisort.zip

em3d

Este programa modela la propagación de ondas electromagnéticas a través de objetos en 3 dimensiones. Realiza un cálculo simple sobre un grafo irregular bipartito (implementado utilizando listas simplemente enlazadas) cuyos nodos representan valores de campo eléctrico y magnético. El algoritmo es el descripto por Culler, et al. [CDG93].

Código fuente disponible en: http://www.fantascienza.net/leonardo/js/dolden_em3d.zip

tsp

Este programa implementa una heurística para resolver el problema del viajante (*traveling salesman problem*) utilizando árboles binarios balanceados. El algoritmo utilizado es el descripto por Karp [KAR77].

Código fuente disponible en: http://www.fantascienza.net/leonardo/js/dolden_tsp.zip

voronoï

Este programa genera un conjunto aleatorio de puntos y computa su diagrama de Voronoï, una construcción geométrica que permite construir una partición del plano euclídeo, utilizando el algoritmo descripto por Guibas & Stolfi [GS85].

Código fuente disponible en: http://codepad.org/xGDCS3KO

5.1.3 Programas *reales*

Dil (escrito en su mayor parte por Aziz Köksal y publicado bajo licencia GPL) es, lamentablemente, el único programa real hallado que, a pesar de estar incompleto, es lo suficientemente grande, mantenido y estable como para ser incluido en el banco de pruebas. Se trata de un compilador de D escrito en D y está incompleto porque no puede generar código (falta implementar el análisis semántico y la generación de código). Es principalmente utilizado para generar documentación a partir del código.

El programa está compuesto por:

- 32.000 líneas de código fuente (aproximadamente).
- 86 módulos (o archivos).
- 322 diferentes tipos de datos definidos por el usuario, de los cuales 34 son tipos *livianos* (`struct`) y 288 tipos polimórficos (`class`), de los que 260 son subtipos (sub-clases).

Puede observarse entonces que a pesar de ser incompleto, es una pieza de software bastante compleja y de dimensión considerable.

Además, al interpretar código fuente se hace un uso intensivo de cadenas de texto que en general presentan problemas muy particulares por poder ser objetos extremadamente pequeños y de tamaños poco convencionales (no múltiplos de palabras, por ejemplo). A su vez, el texto interpretado es convertido a una representación interna en forma de árbol (o *árbol de sintaxis abstracta*) modelado por tipos *livianos* y polimórficos que están organizados en arreglos dinámicos contiguos y asociativos (que usan muchos servicios del recolector). Finalmente estos objetos son manipulados para obtener y generar la información necesaria, creando y dejando de usar objetos constantemente (pero no como única forma de procesamiento, como otras pruebas sintetizadas).

Por último, a diferencia de muchos otros programas escritos en D, que dadas algunas de las ineficiencias del recolector invierten mucho trabajo en limitar su uso, este programa no está escrito pensando en dichas limitaciones, por lo que muestra un funcionamiento muy poco sesgado por estas infortunadas circunstancias.

Por todas estas razones, Dil es el ejemplar que tal vez mejor sirve a la hora de medir de forma realista los resultados obtenidos o los avances realizados. Si bien, como se ha dicho anteriormente, las demás pruebas del banco pueden ser útiles para encontrar problemas muy particulares, está es la que da una lectura más cercana a la realidad del uso de un recolector.

5.2 Modificaciones propuestas

Se decide realizar todas las modificaciones al recolector actual de forma progresiva e incremental, partiendo como base del recolector de la versión 0.99.9 de Tango. Las razones que motivan esta decisión son varias; por un lado es lo más apropiado dados los requerimientos claves mencionados al principio de este capítulo. Por ejemplo, al hacer cambios incrementales es más fácil comprobar que la eficiencia no

se aleja mucho del actual con cada modificación y una modificación gradual impone menos resistencia a la aceptación del nuevo recolector.

Además la construcción de un recolector de cero es una tarea difícil considerando que un error en el recolector es extremadamente complejo de rastrear, dado que en general el error se detecta en el *mutator* y en una instancia muy posterior al origen real del error. Esto ha sido comprobado de forma práctica, dado que, a modo de ejercicio para interiorizarse en el funcionamiento del *runtime* de D, primero se ha construido desde cero una implementación de un recolector *naïve*, resultando muy difícil su depuración por las razones mencionadas. Por el contrario, comenzar con un recolector en funcionamiento como base hace más sencillo tanto probar cada pequeña modificación para asegurar que no introduce fallos, como encontrar y reparar los fallos cuando estos se producen, ya que el código incorrecto introducido está bien aislado e identificado.

A continuación se hace un recorrido sobre cada una de las mejoras propuestas, y en los casos en los que la mejora propone un cambio algorítmico, se analiza la corrección del algoritmo resultante, partiendo de la base de que el algoritmo tomado como punto de partida es un marcado y barrido que utiliza la abstracción tricolor para hacer la fase de marcado de forma iterativa (ver *Marcado y barrido* y *Abstracción tricolor*), cuya corrección ya está probada en la literatura preexistente.

5.2.1 Configurabilidad

Una de las primeras mejoras propuestas es la posibilidad de configurar el recolector de forma más sencilla. El requerimiento mínimo es la posibilidad de configurar el recolector sin necesidad de recompilarlo. La complejidad de esto surge de que el recolector debe ser transparente para el programa del usuario.

Configurar el recolector en tiempo de compilación del programa del usuario probablemente requeriría modificar el compilador, y además, si bien es una mejora sustancial a la configuración en tiempo de compilación del recolector, no termina de ser completamente conveniente para realizar pruebas reiteradas con un mismo programa para encontrar los mejores valores de configuración para ese programa en particular.

Por otro lado, permitir configurar el recolector en tiempo de ejecución, una vez que su estructura interna ya fue definida y creada, puede ser no solo tedioso y complejo, además ineficiente, por lo tanto esta opción también se descarta.

Finalmente, lo que parece ser más apropiado para un recolector, es permitir la configuración en *tiempo de inicialización*. Es decir, configurar el recolectar sin necesidad de recompilar ni el programa del usuario ni el recolector, pero antes de que el programa del usuario inicie, de manera que una vez iniciado el recolector con ciertos parámetros, éstos no cambien nunca más en durante la vida del programa.

Este esquema provee la mejor relación entre configurabilidad, conveniencia, eficiencia y simplicidad. Una posibilidad para lograr esto es utilizar parámetros de línea de comandos, sin embargo no parece ni sencillo (proveer una forma de leer los parámetros de línea de comandos requiere cambios en el *runtime*) ni apropiado (el recolector debería ser lo más transparente posible para el programa del usuario).

Otra posibilidad es utilizar variables de entorno, que parece ser la opción más sencilla y apropiada. Sencilla porque las variables de entorno pueden ser leídas directamente al inicializar el recolector sin necesidad de cooperación alguna del *runtime*, a través de `getenv(3)`. Apropiada porque, si bien el problema de invasión del programa del usuario también existe, es una práctica más frecuente y aceptada la configuración de módulos internos o bibliotecas compartidas a través de variables de entorno.

Por último, antes de comenzar a usar este esquema de configuración, se verifica que tomar ciertas decisiones en tiempo de ejecución no impacten en la eficiencia del recolector. Para esto se convierten algunas opciones que antes eran solo seleccionables en tiempo de compilación del recolector para que puedan ser seleccionables en tiempo de inicialización y se comprueba que no hay una penalización apreciable.

Especificación de opciones

Para especificar opciones de configuración, hay que hacerlo a través de la variable de entorno de nombre **D_GC_OPTS**. El valor de esa variable es interpretado de la siguiente manera (en formato similar a *BNF*):

```
D_GC_OPTS  ::=  option ( ':' option )* <lista de opciones>
option     ::=  name [ '=' value ]
name       ::=  namec namec* <nombre de la opción>
value      ::=  valuec* <valor de la opción>
namec      ::=  valuec - '='
valuec     ::=  [0x01-0xFF] - ':' <cualquiera salvo '\0' y ':'>
```

Es decir, se compone de una lista de opciones separadas por **:**. Cada opción se especifica con un nombre, opcionalmente seguido por un valor (separados por **=**).

El valor de una opción puede ser un texto arbitrario (exceptuando los caracteres `'\0'` y `':'` y de longitud máxima 255), pero cada opción lo interpreta de forma particular. Como caso general, hay opciones booleanas, que toman como valor verdadero un cualquier número distinto de 0 (o si el valor es vació, es decir, solo se indica el nombre de la opción), y como valor falso cualquier otro texto.

A continuación se listan las opciones reconocidas por el recolector (indicando el formato del valor de la opción de tener uno especial):

`mem_stomp`
: Esta es una opción (booleana) disponible en el recolector original, pero que se cambia para que sea configurable en tiempo de inicialización (estando desactivada por omisión). Activa la opción `MEMSTOMP` descripta en *Herramientas para depuración*.

`sentinel`
: Esta opción es también booleana (desactivada por omisión), está disponible en el recolector original, y se la cambia para sea configurable en tiempo de inicialización. Activa la opción `SENTINEL` descripta en *Herramientas para depuración*.

`pre_alloc`
: Esta opción permite crear una cierta cantidad de *pools* de un tamaño determinado previo a que inicie el programa. Si se especifica solo un número, se crea un *pool* con ese tamaño en MiB. Si, en cambio, se especifica una cadena del tipo `3x1`, el primer número indica la cantidad de *pools* y el segundo el tamaño en MiB de cada uno (3 *pools* de 1MiB en este caso). Ver *Pre-asignación de memoria* más adelante para más detalles sobre la utilidad de esta opción.

`min_free`
: El valor de esta opción indica el porcentaje mínimo del *heap* que debe quedar libre luego de una recolección. Siendo un porcentaje, solo se aceptan valores entre 0 y 100, siendo su valor por omisión 5. Ver *Mejora del factor de ocupación del heap* más adelante para más detalles sobre su propósito.

`malloc_stats_file`
: Esta opción sirve para especificar un archivo en el cual escribir un reporte de todas la operaciones de pedido de memoria realizadas por el programa (durante su tiempo de vida). Ver *Recolección de estadísticas* más adelante para más detalles sobre la información provista y el formato del reporte.

`collect_stats_file`
: Esta opción sirve para especificar un archivo en el cual escribir un reporte de todas las recolecciones hechas durante el tiempo de vida del programa. Ver *Recolección de estadísticas* más adelante para más detalles sobre la información provista y el formato del reporte.

conservative
: Esta opción booleana permite desactivar el escaneo preciso del *heap*, forzando al recolector a ser completamente conservativo (excepto por los bloques con el atributo `NO_SCAN` que siguen sin ser escaneados). Ver *Marcado preciso* más adelante para más detalles sobre la existencia de esta opción.

fork
: Esta opción booleana (activada por omisión) permite seleccionar si el recolector debe correr la fase de marcado en paralelo o no (es decir, si el recolector corre de forma concurrente con el *mutator*). Para más detalles ver *Marcado concurrente* más adelante.

eager_alloc
: Esta opción booleana (activada por omisión), sólo puede estar activa si `fork` también lo está y sirve para indicar al recolector que reserve un nuevo *pool* de memoria cuando una petición no puede ser satisfecha, justo antes de lanzar la recolección concurrente. Ver *Creación ansiosa de pools (eager allocation)* más adelante para más detalles sobre el propósito de esta opción.

early_collect
: Esta opción booleana (desactivada por omisión), también sólo puede estar activa si `fork` está activa y sirve para indicar al recolector que lance una recolección (concurrente) antes de que la memoria libre se termine (la recolección temprana será disparada cuando el porcentaje de memoria libre sea menor a `min_free`). Ver *Recolección temprana (early collection)* más adelante para más detalles sobre el propósito de esta opción.

Cualquier opción o valor no reconocido es ignorado por el recolector. Se utilizan los valores por omisión de las opciones que no fueron especificadas, o cuyos valores no pudieron ser interpretados correctamente.

Para cambiar la configuración del recolector se puede invocar el programa de la siguiente manera (usando un intérprete de comandos del tipo *bourne shell*):

```
D_GC_OPTS=conservative:eager_alloc=0:early_collect=1:pre_alloc=2x5 ./prog
```

En este ejemplo, se activan las opciones `conservative` y `early_collect`, se desactiva `eager_alloc` y se crean 2 *pools* de 5MiB cada uno al inicializar el recolector.

5.2.2 Reestructuración y cambios menores

Si bien se decide no comenzar una implementación desde cero, se ha mostrado (ver *Complejidad del código y documentación*) que la implementación actual es lo suficientemente desprolija como para complicar su modificación. Es por esto que se hacen algunas reestructuraciones básicas del código, reescribiendo o saneando de forma incremental todas aquellas partes que complican su evolución.

Además de las modificaciones puramente estéticas (aunque no por eso menos valuables, ya que la legibilidad y simplicidad del código son un factor fundamental a la hora de ser mantenido o extendido), se hacen otras pequeñas mejoras, que se detallan a continuación.

Remoción de memoria *no-encomendada*

Se elimina la distinción entre memoria *encomendada* y *no-encomendada* (ver *Memoria encomendada*), pasando a estar *encomendada* toda la memoria administrada por el recolector.

Si bien a nivel de eficiencia este cambio no tuvo impacto alguno (cuando en un principio se especuló con que podría dar alguna ganancia en este sentido), se elimina el concepto de memoria *encomendada* para quitar complejidad al código.

Esta mejora no afecta a la corrección del algoritmo, ya que a nivel lógico el recolector solo ve la memoria *encomendada*.

Caché de `Pool.findSize()`

Se crea un caché de tamaño de bloque para el método `findSize()` de un *pool*. Esto acelera considerablemente las operaciones que necesitan pedir el tamaño de un bloque reiteradamente, por ejemplo, al añadir nuevos elementos a un arreglo dinámico. En esencia es una extensión a una de las optimizaciones propuestas por Vladimir Panteleev [PAN09], que propone un caché global para todo el recolector en vez de uno por *pool*.

Esta mejora tampoco afecta a la corrección del algoritmo, ya que nuevamente no afecta su comportamiento a nivel lógico, solo cambia detalles en la implementación de forma transparentes para el algoritmo de recolección.

Optimizaciones sobre `findPool()`

Al analizar los principales cuellos de botella del recolector, es notoria la cantidad de tiempo que pasa ejecutando la función `findPool()`, que dado un puntero devuelve el *pool* de memoria al cual pertenece. Es por esto que se minimiza el uso de esta función. Además, dado que los *pools* de memoria están ordenados por el puntero de comienzo del bloque de memoria manejado por el *pool*, se cambia la búsqueda (originalmente lineal) por una búsqueda binaria. Finalmente, dado que la lista de libre está construida almacenando el puntero al siguiente en las mismas celdas que componen la lista, se almacena también el puntero al *pool* al que dicha celda pertenece (dado que la celda más pequeña es de 16 bytes, podemos garantizar que caben dos punteros, incluso para arquitecturas de 64 bits). De esta manera no es necesario usar `findPool()` al quitar una celda de la lista de libres.

Una vez más, la mejora no afecta la corrección del código.

Pre-asignación de memoria

Esta opción permite crear una cierta cantidad de *pools* de un tamaño determinado previo a que inicie el programa. Normalmente el recolector no reserva memoria hasta que el programa lo pida. Esto puede llegar a evitar que un programa haga muchas recolecciones al comenzar, hasta que haya cargado su conjunto de datos de trabajo.

Se han analizado varios valores por omisión pero ninguno es consistentemente mejor que comenzar sin memoria asignada, por lo tanto no se cambia el comportamiento original, pero se agrega una opción (ver `pre_alloc` en *Especificación de opciones*) para que el usuario pueda experimentar con cada programa en particular si esta opción es beneficiosa.

Esta opción tampoco cambia la corrección del algoritmo de recolección, solo sus condiciones iniciales.

Mejora del factor de ocupación del *heap*

El factor de ocupación del *heap* debe ser apropiado por dos razones. Por un lado, si el *heap* está demasiado ocupado todo el tiempo, serán necesarias muchas recolecciones, lo que puede llegar a ser ex-

tremadamente ineficiente en casos patológicos (ver *Factor de ocupación del heap*). Por otro lado, si el tamaño del *heap* es extremadamente grande (en comparación con el tamaño real del grupo de trabajo del programa), se harán pocas recolecciones pero cada una es muy costosa, porque el algoritmo de marcado y barrido es $O(|Heap|)$ (ver *Marcado y barrido*). Además la afinidad del caché va a ser extremadamente pobre.

Para mantener el factor de ocupación dentro de límites razonables, se agrega la opción `min_free` (ver *Especificación de opciones*). Esta opción indica el recolector cual debe ser el porcentaje mínimo del *heap* que debe quedar libre luego de una recolección. En caso de no cumplirse, se pide más memoria al sistema operativo para cumplir este requerimiento. Además, luego de cada recolección se verifica que el tamaño del *heap* no sea mayor a `min_free`, para evitar que el *heap* crezca de forma descontrolada. Si es mayor a `min_free` se intenta minimizar el uso de memoria liberando *pools* que estén completamente desocupados, mientras que el factor de ocupación siga siendo mayor a `min_free`. Si liberar un *pool* implica pasar ese límite, no se libera y se pasa a analizar el siguiente y así sucesivamente.

Esta modificación no afecta a la corrección del algoritmo, ya que no lo afecta directamente.

Modificaciones descartadas

Se realizan varias otras modificaciones, con la esperanza de mejorar la eficiencia del recolector, pero que, al contrario de lo esperado, empeoran la eficiencia o la mejoran de forma muy marginal en comparación con la complejidad agregada.

Probablemente el caso más significativo, y por tanto el único que vale la pena mencionar, es la conversión de marcado iterativo a marcado recursivo y luego a un esquema híbrido. Como se describe en *Problemas y limitaciones*, el marcado iterativo tiene sus ventajas, pero tiene desventajas también. La conversión a puramente recursivo resulta impracticable dado que desemboca en errores de desbordamiento de pila.

Por lo tanto se prueba con un esquema híbrido, poniendo un límite a la recursividad, volviendo al algoritmo iterativo cuando se alcanza este límite.

La implementación del algoritmo híbrido consiste en los siguientes cambios sobre el algoritmo original (ver *Fase de marcado*):

```
function mark_phase() is
   global more_to_scan = false
   global depth = 0                                    // Agregado
   stop_the_world()
   clear_mark_scan_bits()
   mark_free_lists()
   mark_static_data()
   push_registers_into_stack()
   thread_self.stack.end = get_stack_top()
   mark_stacks()
   pop_registers_from_stack()
   mark_user_roots()
   mark_heap()
   start_the_world()

function mark_range(begin, end) is
   pointer = begin
   global depth++                                      // Agregado
   while pointer < end
      [pool, page, block] = find_block(pointer)
      if block is not null and block.mark is false
```

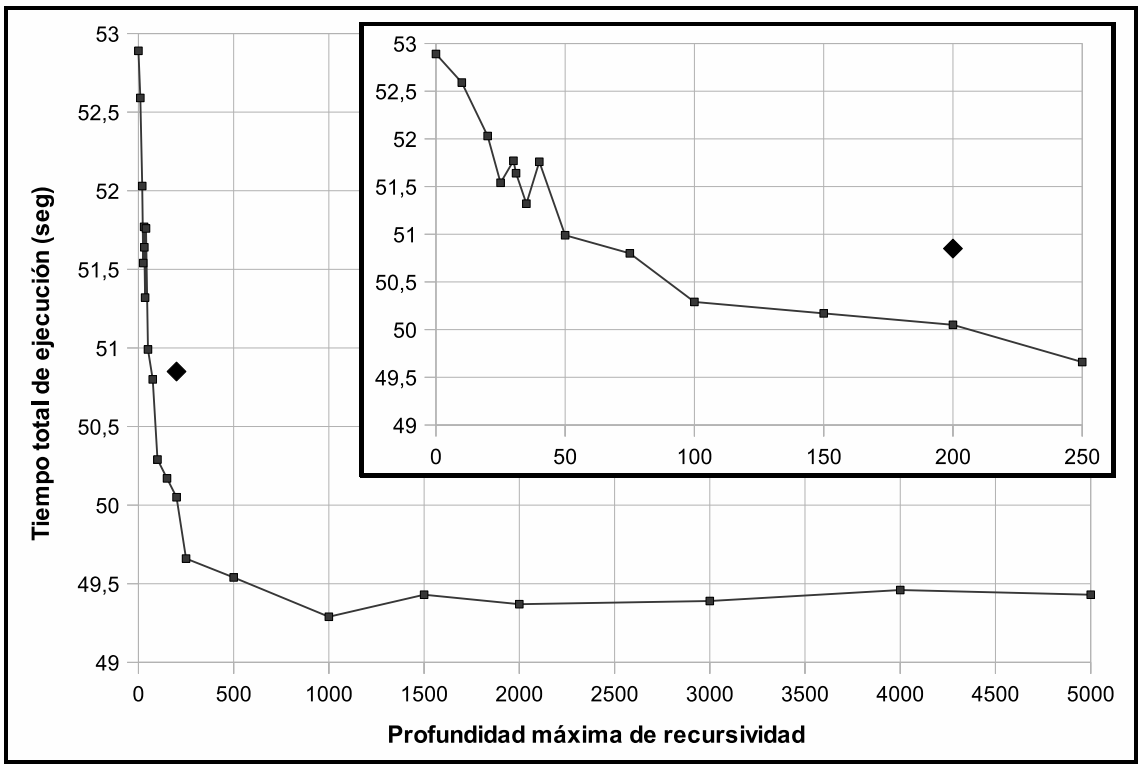


Figura 5.1: Tiempo total de ejecución de Dil al generar la documentación completa del código de Tango en función del valor de `MAX_DEPTH`. El rombo no pertenece a ningún nivel de recursividad, representa el tiempo de ejecución del algoritmo original (puramente iterativo).

```
        block.mark = true
        if block.noscan is false
           block.scan = true
           if (global depth > MAX_DEPTH)          //
              more_to_scan = true                 //
           else                                   // Agregado
              foreach ptr in block.words          //
                 mark(ptr)                        //
  global depth--                                  //
```

Al analizar los resultados de de esta modificación, se observa una mejoría muy level, para valores de `MAX_DEPTH` mayores a cero (en algunos casos bastante mayores). En general para `MAX_DEPTH` cero (es decir, usando el algoritmo de forma completamente iterativa) los resultados son peores, dado que se paga el trabajo extra sin ganancia alguna. En la figura 5.1 se puede ver, por ejemplo, el tiempo total de ejecución de Dil al generar la documentación completa del código de Tango, según varía el valor de `MAX_DEPTH`.

Dado que aumentar el nivel máximo de recursividad significa un uso mayor del *stack*, y que esto puede impactar en el usuario (si el usuario tuviera un programa que esté al borde de consumir todo el *stack*, el recolector podría hacer fallar al programa de una forma inesperada para el usuario, problema que sería muy difícil de depurar para éste), y que los resultados obtenidos no son rotundamente superiores a los resultados sin esta modificación, se opta por no incluir el cambio. Tampoco vale la pena incluirlo como una opción con valor por omisión 0 porque, como se ha dicho, para este caso el resultado es incluso peor que sin la modificación.

Esta modificación mantiene la corrección del recolector dado que tampoco modifica el algoritmo sino su implementación. Además ambos casos extremos son correctos (si `MAX_DEPTH` es 0, el algoritmo es

puramente iterativo y si pudiera ser infinito resultaría en el algoritmo puramente recursivo).

5.2.3 Recolección de estadísticas

Un requerimiento importante, tanto para evaluar los resultados de este trabajo como para analizar el comportamiento de los programas estudiados, es la recolección de estadísticas. Hay muchos aspectos que pueden ser analizados a la hora de evaluar un recolector, y es por esto que se busca que la recolección de datos sea lo más completa posible.

Con este objetivo, se decide recolectar datos sobre lo que probablemente sean las operaciones más importantes del recolector: asignación de memoria y recolección.

Todos los datos recolectados son almacenados en archivos que se especifican a través de opciones del recolector (ver *Especificación de opciones*). Los archivos especificados debe poder ser escritos (y creados de ser necesario) por el recolector (de otra forma se ignora la opción). El conjunto de datos recolectados son almacenados en formato *CSV* en el archivo, comenzando con una cabecera que indica el significado de cada columna.

Los datos recolectados tienen en general 4 tipos de valores diferentes:

Tiempo
: Se guarda en segundos como número de punto flotante (por ejemplo `0.12`).

Puntero
: Se guarda en forma hexadecimal (por ejemplo `0xa1b2c3d4`).

Tamaño
: Se guarda como un número decimal, expresado en bytes (por ejemplo `32`).

Indicador
: Se guarda como el número `0` si es falso o `1` si es verdadero.

Esta modificación mantiene la corrección del recolector dado que no hay cambio algorítmico alguno.

Asignación de memoria

La recolección de datos sobre asignación de memoria se activa asignando un nombre de archivo a la opción `malloc_stats_file`. Por cada asignación de memoria pedida por el programa (es decir, por cada llamada a la función `gc_malloc()`) se guarda una fila con los siguientes datos:

1. Cantidad de segundos que pasaron desde que empezó el programa (*timestamp*).
2. Tiempo total que tomó la asignación de memoria.
3. Valor del puntero devuelto por la asignación.
4. Tamaño de la memoria pedida por el programa.
5. Si esta petición de memoria disparó una recolección o no.
6. Si debe ejecutarse un *finalizador* sobre el objeto (almacenado en la memoria pedida) cuando ésta no sea más alcanzable (cuando sea barrido).
7. Si objeto carece de punteros (es decir, no debe ser escaneada).
8. Si objeto no debe ser movido por el recolector.
9. Puntero a la información sobre la ubicación de los punteros del objeto.

10. Tamaño del tipo del objeto.
11. Primera palabra con los bits que indican que palabras del tipo deben ser escaneados punteros y cuales no (en hexadecimal).
12. Primera palabra con los bits que indican que palabras del tipo son punteros garantizados (en hexadecimal).

Como puede apreciarse, la mayor parte de esta información sirve más para analizar el programa que el recolector. Probablemente solo el punto 2 sea de interés para analizar como se comporta el recolector.

El punto 8 es completamente inútil, ya que el compilador nunca provee esta información, pero se la deja por si en algún momento comienza a hacerlo. Los puntos 9 a 12 proveen información sobre el tipo del objeto almacenado, útil para un marcado preciso (ver *Marcado preciso*).

El punto 6 indica, indirectamente, cuales de los objetos asignados son *pesados*, ya que éstos son los únicos que pueden tener un *finalizador*. Además, a través de los puntos 4 y 10 es posible inferir si lo que va almacenarse es un objeto solo o un arreglo de objetos.

Recolección de basura

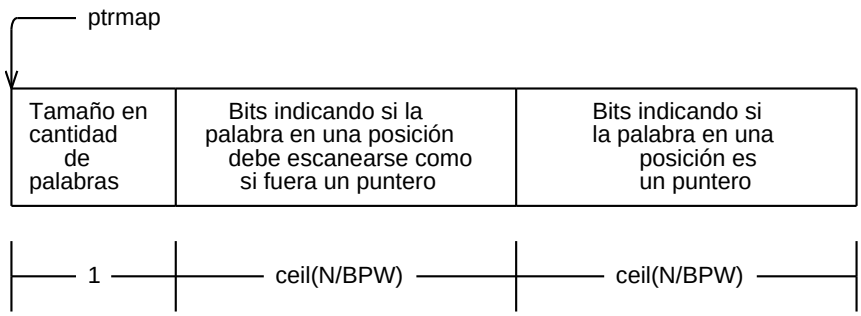
Los datos sobre las recolecciones realizadas se guardan al asignar un nombre de archivo a la opción `collect_stats_file`. Cada vez que se dispara una recolección [2] (es decir, cada vez que se llama a la función `fullcollect()`) se guarda una fila con los siguientes datos:

1. Cantidad de segundos que pasaron desde que empezó el programa (*timestamp*).
2. Tiempo total que tomó la asignación de memoria que disparó la recolección.
3. Tiempo total que tomó la recolección.
4. Tiempo total que deben pausarse todos los hilos (tiempo de *stop-the-world*).
5. Cantidad de memoria usada antes de la recolección.
6. Cantidad de memoria libre antes de la recolección.
7. Cantidad de memoria desperdiciada [3] antes de la recolección.
8. Cantidad de memoria utilizada por el mismo recolector antes de la recolección (para sus estructuras internas).
9. Cantidad de memoria usada después de la recolección.
10. Cantidad de memoria libre después de la recolección.
11. Cantidad de memoria desperdiciada después de la recolección.
12. Cantidad de memoria utilizada por el mismo recolector después de la recolección.

Si bien el punto 4 parece ser el más importante para un programa que necesita baja latencia, dado el *lock* global del recolector, el punto 2 es probablemente el valor más significativo en este aspecto, dado que, a menos que el programa en cuestión utilice muy poco el recolector en distintos hilos, los hilos se verán pausados de todas formas cuando necesiten utilizar el recolector.

[2] Esto es en el sentido más amplio posible. Por ejemplo, cuando se utiliza marcado concurrente (ver *Marcado concurrente*), se guarda esta información incluso si ya hay una recolección activa, pero el tiempo de pausa de los hilos será -1 para indicar que en realidad nunca fueron pausados.

[3] Memoria *desperdiciada* se refiere a memoria que directamente no puede utilizarse debido a la fragmentación. Si por ejemplo, se piden 65 bytes de memoria, dada la organización del *heap* en bloques (ver *Organización del heap*), el recolector asignará un bloque de 128 bytes, por lo tanto 63 bytes quedarán desperdiciados.



Cuadro 5.1: Estructura de la información de tipos provista por el compilador.

5.2.4 Marcado preciso

Para agregar el soporte de marcado preciso se aprovecha el trabajo realizado por Vincent Lang (ver *Principales categorías del estado del arte*) [DBZ3463], gracias a que se basa en D 1.0 y Tango, al igual que este trabajo. Dado el objetivo y entorno común, se abre la posibilidad de adaptar sus cambios a este trabajo, utilizando una versión modificada de DMD (dado que los cambios aún no están integrados al compilador oficial todavía).

Información de tipos provista por el compilador

Con éstas modificaciones, el compilador en cada asignación le pasa al recolector información sobre los punteros del tipo para el cual se pide la memoria. Esta información se pasa como un puntero a un arreglo de palabras con la estructura mostrada en la figura 5.1 y que se describe a continuación.

- La primera palabra indica el tamaño, en **cantidad de palabras**, del tipo para el cual se pide la memoria (N).
- Las siguientes $ceil(\frac{N}{BPW})$ palabras indican, como un conjunto de bits, qué palabras deben ser escaneadas por el recolector como si fueran punteros (donde BPW indica la cantidad de bits por palabra, por ejemplo 32 para x86).
- Las siguientes $ceil(\frac{N}{BPW})$ palabras son otro conjunto de bits indicando qué palabras son realmente punteros.

Los conjuntos de bits guardan la información sobre la primera palabra en el bit menos significativo. Dada la complejidad de la representación, se ilustra con un ejemplo. Dada la estructura:

```
union U {
   ubyte ub;
   void* ptr;
}

struct S
{
   void* begin1;                          // 1 word
   byte[size_t.sizeof * 14 + 1] bytes;    // 15 words
   // el compilador agrega bytes de "padding" para alinear
   void* middle;                          // 1 word
   size_t[14] ints;                       // 14 words
   void* end1;                            // 1 words
   // hasta acá se almacenan los bits en la primera palabra
   void* begin2;                          // 1 words
```

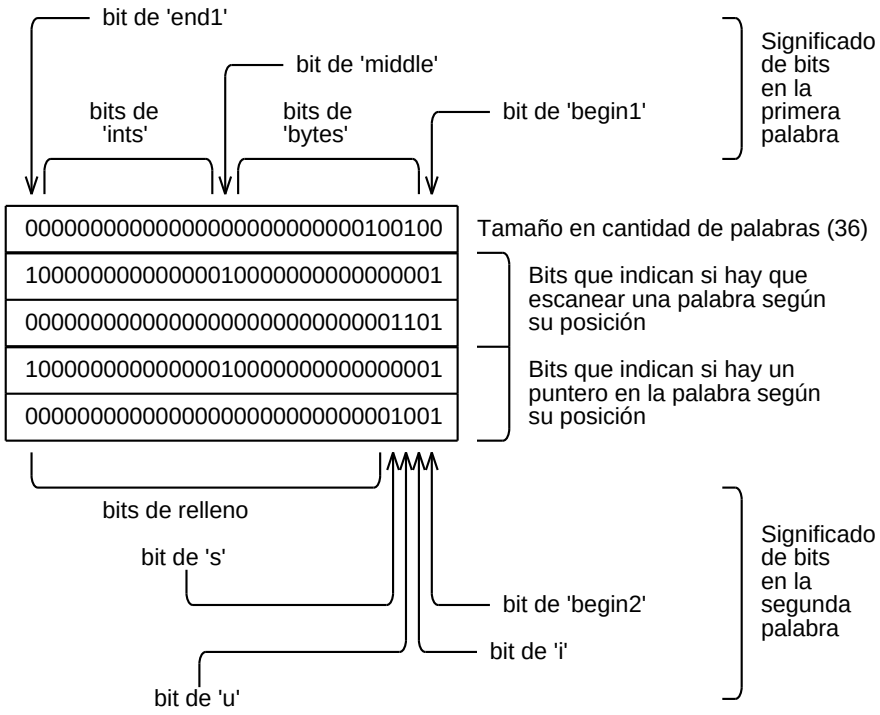


Figura 5.2: Ejemplo de estructura de información de tipos generada para el tipo `S`.

```
    int i;                                  // 1 word
    U u;                                    // 1 word
    S* s;                                   // 1 word
}
```

El compilador genera la estructura que se muestra en la figura 5.2 (asumiendo una arquitectura de 32 bits). Como puede apreciarse, el miembro `u`, al ser una unión entre un puntero y un dato común, el compilador no puede asegurar que lo que se guarda en esa palabra sea realmente un puntero, pero indica que debe ser escaneado. El recolector debe debe ser conservativo en este caso, y escanear esa palabra como si fuera un puntero.

Si una implementación quisiera mover memoria (ver *Movimiento de celdas*), debería mantener inmóvil a cualquier objeto que sea apuntado por una palabra de estas características, ya que no es seguro actualizar la palabra con la nueva posición el objeto movido. Es por esta razón que se provee desglosada la información sobre lo que hay que escanear, y lo que es realmente un puntero (que puede ser actualizado de forma segura por el recolector de ser necesario).

Implementación en el recolector

La implementación está basada en la idea original de David Simcha, pero partiendo de la implementación de Vincent Lang (que está basada en Tango) y consiste en almacenar el puntero a la estructura con la descripción del tipo generada por el compilador al final del bloque de datos. Este puntero solo se almacena si el bloque solicitado no tiene el atributo `NO_SCAN`, dado que en ese caso no hace falta directamente escanear ninguna palabra del bloque.

En la figura 5.3 en la página siguiente se puede ver, como continuación del ejemplo anterior, como se almacenaría en memoria un objeto del tipo `S`.

Un problema evidente de este esquema es que si el tamaño de un objeto se aproxima mucho al tamaño

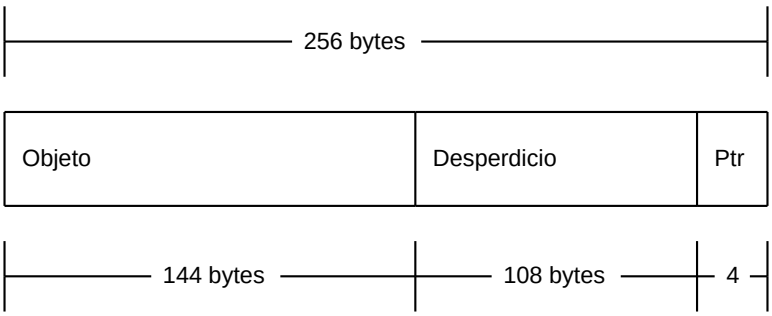


Figura 5.3: Ejemplo de bloque que almacena un objeto de tipo `S` con información de tipo.

de bloque (difiere en menos de una palabra), el objeto ocupará el doble de memoria.

El algoritmo de marcado se cambia de la siguiente forma:

```
// Agregado
global conservative_ptrmap = [1, 1, 0]

// Agregado
function must_scan_word(pos, bits) is
   return bits[pos / BITS_PER_WORD] & (1 << (pos % BITS_PER_WORD))

function mark_range(begin, end, ptrmap) is              // Modificado
   number_of_words_in_type = ptrmap[0]                  // Agregado
   size_t* scan_bits = ptrmap + 1                       // Agregado
   pointer = begin
   while pointer < end
      foreach word_pos in 0..number_of_words_in_type    //
         if not must_scan_word(word_pos, scan_bits)     // Agregado
            continue                                    //
         [pool, page, block] = find_block(pointer)
         if block is not null and block.mark is false
            block.mark = true
            if block.noscan is false
               block.scan = true
               global more_to_scan = true
      pointer += number_of_words_in_type                // Modificado

function mark_heap() is
   while global more_to_scan
      global more_to_scan = false
      foreach pool in heap
         foreach page in pool
            if page.block_size <= PAGE // saltea FREE y CONTINUATION
               foreach block in page
                  if block.scan is true
                     block.scan = false
                     if page.block_size is PAGE // obj grande //
                        begin = cast(byte*) page                //
                        end = find_big_object_end(pool, page) //
                     else // objeto pequeño                    //
                        begin = block.begin                     //
                        end = block.end                         // Modificado
                     ptrmap = global conservative_ptrmap        //
                     if NO_SCAN not in block.attrs              //
                        end -= size_t.sizeof                    //
```

```
                        ptrmap = cast(size_t*) *end           //
                    mark_range(begin, end, ptrmap)            //

function mark_static_data() is
   mark_range(static_data.begin, static_data.end,
         global conservative_ptrmap)              // Agregado

function mark_stacks() is
   foreach thread in threads
      mark_range(thread.stack.begin, thread.stack.end,
            global conservative_ptrmap)                // Agregado

function mark_user_roots() is
   foreach root_range in user_roots
      mark_range(root_range.begin, root_range.end,
            global conservative_ptrmap)             // Agregado
```

Las funciones de asignación de memoria se modifican de forma similar, para guardar el puntero a la información de tipos. Esta implementación utiliza solo la información sobre que palabras hay que tratar como punteros (deben ser escaneadas); la información sobre qué palabras son efectivamente punteros no se utiliza ya que no se mueven celdas.

El algoritmo sigue siendo correcto, puesto que solamente se dejan de escanear palabras que el compilador sabe que no pueden ser punteros. Si bien el lenguaje permite almacenar punteros en una variable que no lo sea, esto es comportamiento indefinido por lo tanto un programa que lo hace no es considerado correcto, por lo cual el recolector tampoco debe ser correcto en esas circunstancias.

Cabe destacar que la información de tipos solo se provee para objetos almacenados en el *heap*, el área de memoria estática, registros del procesador y la pila de todos los hilos siguen siendo escaneados de forma completamente conservativa. Se puede forzar el escaneo puramente conservativo utilizando la opción `conservative` (ver *Especificación de opciones*).

5.2.5 Marcado concurrente

Finalmente se procede al objetivo primario de este trabajo, hacer que la fase más costosa del recolector (el marcado) pueda correr de manera concurrente con el *mutator*, con el objeto principal de disminuir el tiempo de pausa.

Cabe aclarar, una vez más, que si bien los recolectores concurrentes buscan disminuir solo el tiempo de *stop-the-world*, en este caso es también fundamental disminuir el tiempo máximo que está tomado el *lock* global, dado que ese tiempo puede convertirse en una pausa para todos los threads que requieran servicios del recolector.

Se decide basar la implementación en el *paper* "Non-intrusive Cloning Garbage Collector with Stock Operating System Support" [RODR97] por las siguientes razones principales:

- Su implementación encaja de forma bastante natural con el diseño del recolector actual, por lo que requiere pocos cambios, lo que hace más factible su aceptación.
- Está basado en la llamada al sistema `fork(2)`, que no solo está muy bien soportada (y de manera muy eficiente) en Linux, debe estar soportada en cualquier sistema operativo *POSIX*.
- No necesita instrumentar el código incluyendo barreras de memoria para informar al recolector cuando cambia el grafo de conectividad. Este es un aspecto fundamental, dada la filosofía de D de no pagar el precio de cosas que no se usan. La penalización en la eficiencia solo se paga cuando

corre el recolector. Este aspecto también es crítico a la hora de evaluar la aceptación de la solución por parte de la comunidad.

- Dada su sencillez general, no es difícil ofrecer el algoritmo concurrente como una opción, de manera que el usuario pueda optar por usarlo o no.

Llamada al sistema *fork*

El término *fork* proviene del inglés y significa *tenedor* de manera textual, pero se lo utiliza como analogía de una bifurcación. La operación crea una copia (llamada *hijo*) del proceso que la ejecuta (llamado *padre*).

El punto más importante es que se crea un espacio de direcciones de memoria separado para el proceso hijo y una copia exacta de todos los segmentos de memoria del proceso padre. Es por esto que cualquier modificación que se haga en el proceso padre, no se refleja en el proceso hijo (y viceversa), a menos que la memoria sea compartida entre los procesos de forma explícita.

Esto, sin embargo, no significa que la memoria física sea realmente duplicada; en general todos los sistemas operativos modernos (como Linux) utilizan una técnica llamada *COW* (de *copy-on-write* en inglés, *copiar-al-escribir* en castellano) que retrasa la copia de memoria hasta que alguno de los dos procesos escribe en un segmento. Recién en ese momento el sistema operativo realiza la copia de **ese segmento solamente**. Es por esto que la operación puede ser muy eficiente, y la copia de memoria es proporcional a la cantidad de cambios que hayan.

`fork(2)` tiene otra propiedad importante de mencionar: detiene todos los hilos de ejecución en el proceso hijo. Es decir, el proceso hijo se crea con un solo hilo (el hilo que ejecutó la operación de `fork(2)`).

Algoritmo

Lo que propone el algoritmo es muy sencillo, utilizar la llamada al sistema `fork(2)` para crear una *fotografía* de la memoria del proceso en un nuevo proceso. En el proceso padre sigue corriendo el *mutator* y en el proceso hijo se corre la fase de marcado. El *mutator* puede modificar el grafo de conectividad pero los cambios quedan aislados del hijo (el marcado), que tiene una visión consistente e inmutable de la memoria. El sistema operativo duplica las páginas que modifica el padre bajo demanda, por lo tanto la cantidad de memoria física realmente copiada es proporcional a la cantidad y dispersión de los cambios que haga el *mutator*.

La corrección del algoritmo se mantiene gracias a que la siguiente invariante se preserva:

> Cuando una celda se convierte en basura, permanece como basura hasta ser reciclada por el recolector.

Es decir, el *mutator* no puede *resucitar* una celda *muerta* y esta invariante se mantiene al correr la fase de marcado sobre una vista inmutable de la memoria. El único efecto introducido es que el algoritmo toma una aproximación más conservativa; una celda que pasó a estar *muerta* luego de que se inicie la fase de marcado, pero antes de que termine, puede no ser reciclada hasta la próxima recolección, dado que este algoritmo no incluye una comunicación entre *mutator* y recolector para notificar cambios en el grafo de conectividad. Pero esto no afecta la corrección del algoritmo, ya que un recolector es correcto cuando nunca recicla celdas *vivas*.

La única comunicación necesaria entre el *mutator* y el recolector son los bits de marcado (ver *Detalles de implementación*), dado que la fase de barrido debe correr en el proceso padre. No es necesario ningún tipo de sincronización entre *mutator* y recolector más allá de que uno espera a que el otro finalice.

Además de almacenar el conjunto de bits `mark` en memoria compartida entre el proceso padre e hijo (necesario para la fase de barrido), las modificaciones necesarias para hacer la fase de marcado concurrente son las siguientes [4]:

```
function collect() is
   stop_the_world()
   fflush(null) // evita que se duplique la salida de los FILE* abiertos
   child_pid = fork()
   if child_pid is 0 // proceso hijo
      mark_phase()
      exit(0) // termina el proceso hijo
   // proceso padre
   start_the_world()
   wait(child_pid)
   sweep()

function mark_phase() is
   global more_to_scan = false
   // Borrado: stop_the_world()
   clear_mark_scan_bits()
   mark_free_lists()
   mark_static_data()
   push_registers_into_stack()
   thread_self.stack.end = get_stack_top()
   mark_stacks()
   pop_registers_from_stack()
   mark_user_roots()
   mark_heap()
   // Borrado: start_the_world()
```

Como se puede observar, el cambio es extremadamente sencillo. Sigue siendo necesario un tiempo mínimo de pausa (básicamente el tiempo que tarda la llamada al sistema operativo *`fork(2)`*) para guardar una vista consistente de los registros del CPU y *stacks* de los hilos. Si bien el conjunto de bits `mark` es compartido por el proceso padre e hijo dado que es necesario para *comunicar* las fases de marcado y barrido, cabe notar que nunca son utilizados de forma concurrente (la fase de barrido espera que la fase de marcado termine antes de usar dichos bits), por lo tanto no necesitan ningún tipo de sincronización y nunca habrá más de una recolección en proceso debido al *lock* global del recolector.

A pesar de que con estos cambios el recolector técnicamente corre de forma concurrente, se puede apreciar que para un programa con un solo hilo el tiempo máximo de pausa seguirá siendo muy grande, incluso más grande que antes dado el trabajo extra que impone crear un nuevo proceso y duplicar las páginas de memoria modificadas. Lo mismo le pasará a cualquier hilo que necesite hacer uso del recolector mientras hay una recolección en proceso, debido al *lock* global.

Para bajar este tiempo de pausa se experimenta con dos nuevas mejoras, que se describen a continuación, cuyo objetivo es correr la fase de marcado de forma concurrente a **todos** los hilos, incluyendo el hilo que la disparó.

[4] Se omite el manejo de errores y la activación/desactivación del marcado concurrente a través de opciones del recolector para facilitar la comprensión del algoritmo y los cambios realizados. Si devuelve con error la llamada a `fork()` o `waitpid()`, se vuelve al esquema *stop-the-world* como si se hubiera desactivado el marcado concurrente utilizando la opción del recolector `fork=0`.

Creación ansiosa de *pools* (*eager allocation*)

Esta mejora, que puede ser controlada a través de la opción `eager_alloc` (ver *Especificación de opciones*), consiste en crear un nuevo *pool* cuando un pedido de memoria no puede ser satisfecho, justo después de lanzar la recolección. Esto permite al recolector satisfacer la petición de memoria inmediatamente, corriendo la fase de marcado de forma realmente concurrente, incluso para programas con un solo hilo o programas cuyos hilos usan frecuentemente servicios del recolector. El precio a pagar es un mayor uso de memoria de forma temporal (y el trabajo extra de crear y eliminar *pools* más frecuentemente), pero es esperable que el tiempo máximo de pausa **real** se vea drásticamente disminuido.

A simple vista las modificaciones necesarias para su implementación parecieran ser las siguientes:

```
// Agregado
global mark_pid = 0

// Agregado
function mark_is_running() is
   return global mark_pid != 0

function collect() is
   if mark_is_running()                          //
      finished = try_wait(global mark_pid)       //
      if finished                                // Agregado
         mark_pid = 0                            //
         sweep()                                 //
      return                                     //
   stop_the_world()
   fflush(null)
   child_pid = fork()
   if child_pid is 0 // proceso hijo
      mark_phase()
      exit(0)
   // proceso padre
   start_the_world()
   // Borrado: wait(child_pid)
   global mark_pid = child_pid
```

Sin embargo con sólo estas modificaciones el algoritmo deja de ser correcto, ya que tres cosas problemáticas pueden suceder:

1. Puede llamarse a la función `minimize()` mientras hay una fase de marcado corriendo en paralelo. Esto puede provocar que se libere un *pool* mientras se lo está usando en la fase de marcado, lo que no sería un problema (porque el proceso de marcado tiene una copia) si no fuera porque los bits de marcado, que son compartidos por los procesos, se liberan con el *pool*.
2. Si un bloque libre es asignado después de que la fase de marcado comienza, pero antes de que termine, ese bloque será barrido dado la función `rebuild_free_lists()` puede reciclar páginas si todos sus bloques tienen el bit `freebits` activo (ver *Fase de barrido*).
3. El *pool* creado ansiosamente, tendrá sus bits de marcado sin activar, por lo que en la fase de barrido será interpretado como memoria libre, incluso cuando puedan estar siendo utilizados por el *mutator*.

El punto 1 sencillamente hace que el programa finalice con una violación de segmento (en el mejor caso) y 2 y 3 pueden desembocar en la liberación de una celda alcanzable por el *mutator*.

El punto 1 se resuelve a través de la siguiente modificación:

```
function minimize() is
   if mark_is_running()                                  // Agregado
      return                                             //
   for pool in heap
      all_free = true
      for page in pool
         if page.block_size is not FREE
            all_free = false
            break
      if all_free is true
         free(pool.pages)
         free(pool)
         heap.remove(pool)
```

La resolución del punto 2 es un poco más laboriosa, ya que hay que mantener actualizado los `freebits`, de forma que las celdas asignadas después de empezar la fase de marcado no sean barridas por tener ese bit activo:

```
function new_big(size) is
   number_of_pages = ceil(size / PAGE_SIZE)
   pages = find_pages(number_of_pages)
   if pages is null
      collect()
      pages = find_pages(number_of_pages)
      if pages is null
         minimize()
         pool = new_pool(number_of_pages)
         if pool is null
            return null
         pages = assign_pages(pool, number_of_pages)
   pages[0].block.free = true                            // Agregado
   pages[0].block_size = PAGE
   foreach page in pages[1 .. end]
      page.block_size = CONTINUATION
   return pages[0]

function assign_page(block_size) is
   foreach pool in heap
      foreach page in pool
         if page.block_size is FREE
            page.block_size = block_size
            foreach block in page
               block.free = true                         // Agregado
               free_lists[page.block_size].link(block)

function mark_phase() is
   global more_to_scan = false
   // Borrado: clear_mark_scan_bits()
   // Borrado: mark_free_lists()
   clear_scan_bits()                                  // Agregado
   mark_free()                                        //
   mark_static_data()
   push_registers_into_stack()
   thread_self.stack.end = get_stack_top()
   mark_stacks()
   pop_registers_from_stack()
```

```
   mark_user_roots()
   mark_heap()

// Agregado
function clear_scan_bits() is
   // La implementación real limpia los bits en bloques de forma eficiente
   foreach pool in heap
      foreach page in pool
         foreach block in page
            block.scan = false

// Agregado
function mark_free() is
   // La implementación real copia los bits en bloques de forma eficiente
   foreach pool in heap
      foreach page in pool
         foreach block in page
            block.mark = block.free

function free_big_object(pool, page) is
   pool_end = cast(byte*) pool.pages + (PAGE_SIZE * pool.number_of_pages)
   do
      page.block_size = FREE
      page.block.free = true                  // Agregado
      page = cast(byte*) page + PAGE_SIZE
   while page < pool_end and page.block_size is CONTINUATION

function new(size, attrs) is
   block_size = find_block_size(size)
   if block_size < PAGE
      block = new_small(block_size)
   else
      block = new_big(size)
   if block is null
      throw out_of_memory
   if final in attrs
      block.final = true
   if noscan in attrs
      block.noscan = true
   block.free = false          // Agregado
   return cast(void*) block

funciones new_pool(number_of_pages = 1) is
   pool = alloc(pool.sizeof)
   if pool is null
      return null
   pool.number_of_pages = number_of_pages
   pool.pages = alloc(number_of_pages * PAGE_SIZE)
   if pool.pages is null
      free(pool)
      return null
   heap.add(pool)
   foreach page in pool
      page.block_size = FREE
      foreach block in page       //
         block.free = true        // Agregado
         block.mark = true        //
```

```
   return pool
```

Finalmente, el punto número 3 puede ser solucionado con el siguiente pequeño cambio:

```
funciones new_pool(number_of_pages = 1) is
   pool = alloc(pool.sizeof)
   if pool is null
      return null
   pool.number_of_pages = number_of_pages
   pool.pages = alloc(number_of_pages * PAGE_SIZE)
   if pool.pages is null
      free(pool)
      return null
   heap.add(pool)
   foreach page in pool
      page.block_size = FREE
      foreach block in page      // Agregado
         block.mark = true       //
   return pool
```

La solución es conservativa porque, por un lado evita la liberación de *pools* mientras haya una recolección en curso (lo que puede hacer que el consumo de memoria sea un poco mayor al requerido) y por otro asegura que, como se mencionó anteriormente, los cambios hechos al grafo de conectividad luego de iniciar la fase de marcado y antes de que ésta termine, no serán detectados por el recolector hasta la próxima recolección (marcar todos los bloques de un nuevo *pool* como el bit `mark` asegura que que la memoria no sea recolectada por la fase de barrido cuando termine el marcado).

Estas modificaciones son las que hacen que el algoritmo siga siendo correcto, asegurando que no se van a liberar celdas *vivas* (a expensas de diferir la liberación de algunas celdas *muertas* por un tiempo).

Recolección temprana (*early collection*)

Esta mejora, que puede ser controlada a través de la opción `early_collect` (ver *Especificación de opciones*), consiste en lanzar una recolección preventiva, antes de que una petición de memoria falle. El momento en que se lanza la recolección es controlado por la opción `min_free` (ver *Mejora del factor de ocupación del heap*).

De esta forma también puede correr de forma realmente concurrente el *mutator* y el recolector, al menos hasta que se acabe la memoria, en cuyo caso, a menos que la opción `eager_alloc` también esté activada (ver *Creación ansiosa de pools (eager allocation)*), se deberá esperar a que la fase de marcado termine para recuperar memoria en la fase de barrido.

Para facilitar la comprensión de esta mejora se muestran sólo los cambios necesarios si no se utiliza la opción `eager_alloc`:

```
function collect(early = false) is  // Modificado
   if mark_is_running()
      finished = try_wait(global mark_pid)
      if finished
         mark_pid = 0
         sweep()
         return                    //
      else if early                // Agregado
         return                    //
```

```
   stop_the_world()
   fflush(null)
   child_pid = fork()
   if child_pid is 0 // proceso hijo
      mark_phase()
      exit(0)
   // proceso padre
   start_the_world()
   if early                          //
      global mark_pid = child_pid    //
   else                              // Agregado
      wait(child_pid)                //
      sweep()                        //

// Agregado
function early_collect() is
   if not collect_in_progress() and (percent_free < min_free)
      collect(true)

function new(size, attrs) is
   block_size = find_block_size(size)
   if block_size < PAGE
      block = new_small(block_size)
   else
      block = new_big(size)
   if block is null
      throw out_of_memory
   if final in attrs
      block.final = true
   if noscan in attrs
      block.noscan = true
   early_collect()               // Agregado
   return cast(void*) block
```

Es de esperarse que cuando no está activa la opción `eager_alloc` por un lado el tiempo de pausa máximo no sea tan chico como cuando sí lo está (dado que si la recolección no se lanza de forma suficientemente temprana se va a tener que esperar que la fase de marcado termine), y por otro que se hagan más recolecciones de lo necesario (cuando pasa lo contrario, se recolecta más temprano de lo que se debería). Sin embargo, también es de esperarse que el consumo de memoria sea un poco menor que al usar la opción `eager_alloc`.

En cuanto a la corrección del algoritmo, éste solamente presenta los problemas número 1 y 2 mencionados en *Creación ansiosa de pools (eager allocation)*, dado que jamás se crean nuevos *pools* y la solución es la ya presentada, por lo tanto el algoritmo sigue siendo correcto con los cuidados pertinentes.

5.3 Resultados

Los resultados de las modificación propuestas en la sección anterior (ver *Modificaciones propuestas*) se evalúan utilizando el conjunto de pruebas mencionado en la sección *Banco de pruebas*).

En esta sección se describe la forma en la que el conjunto de pruebas es utilizado, la forma en la que se ejecutan los programas para recolectar dichos resultados y las métricas principales utilizadas para analizarlos.

A fines prácticos, y haciendo alusión al nombre utilizado por Tango, en esta sección se utiliza el nombre

TBGC (acrónimo para el nombre en inglés *Tango Basic Garbage Collector*) para hacer referencia al recolector original provisto por Tango 0.99.9 (que, recordamos, es el punto de partida de este trabajo). Por otro lado, y destacando la principal modificación propuesta por este trabajo, haremos referencia al recolector resultante de éste utilizando el nombre **CDGC** (acrónimo para el nombre en inglés *Concurrent D Garbage Collector*).

5.3.1 Ejecución del conjunto de pruebas

Dado el indeterminismo inherente a los sistemas operativos de tiempo compartido modernos, se hace un particular esfuerzo por obtener resultados lo más estable posible.

Hardware y software utilizado

Para realizar las pruebas se utiliza el siguiente hardware:

- Procesador Intel(R) Core(TM)2 Quad CPU Q8400 @ 2.66GHz.
- 2GiB de memoria RAM.

El entorno de software es el siguiente:

- Sistema operativo Debian Sid (para arquitectura *amd64*).
- Linux 2.6.35.7.
- DMD 1.063 modificado para proveer información de tipos al recolector (ver *Marcado preciso*).
- *Runtime* Tango 0.99.9 modificado para utilizar la información de tipos provista por el compilador modificado.
- GCC 4.4.5.
- Embedded GNU C Library 2.11.2.

Si bien el sistema operativo utiliza arquitectura *amd64*, dado que DMD todavía no soporta 64 bits, se compila y corren los programas de D en 32 bits.

Opciones del compilador

Los programas del conjunto de pruebas se compilan utilizando las siguientes opciones del compilador DMD:

`-O`
: Aplica optimizaciones generales.

`-inline`
: Aplica la optimización de expansión de funciones. Consiste en sustituir la llamada a función por el cuerpo de la función (en general solo para funciones pequeñas).

`-release`
: No genera el código para verificar pre y post-condiciones, invariantes de representación, operaciones fuera de los límites de un arreglo y *assert*s en general (ver *Programación confiable*).

Parámetros de los programas

Los programas de prueba se ejecutan siempre con los mismos parámetros (a menos que se especifique lo contrario), que se detallan a continuación.

conalloc
`40 4 bible.txt`

Procesa 40 veces un archivo de texto plano (de 4MiB de tamaño) [5] utilizando 4 hilos (más el principal).

concpu
`40 4 bible.txt`

Procesa 40 veces un archivo de texto plano (de 4MiB de tamaño) utilizando 4 hilos (más el principal).

split
`bible.txt 2`

Procesa dos veces un archivo de texto plano (de 4MiB de tamaño).

sbtree
`16`

Construye árboles con profundidad máxima 16.

bh
`-b 4000`

Computa las interacciones gravitatorias entre 4.000 cuerpos.

bisort
`-s 2097151`

Ordena alrededor de 2 millones de números (exactamente $2^{21} = 2097151$).

em3d
`-n 4000 -d 300 -i 74`

Realiza 74 iteraciones para modelar 4.000 nodos con grado 300.

tsp
`-c 1000000`

Resuelve el problema del viajante a través de una heurística para un millón de ciudades.

voronoi
`-n 30000`

Se construye un diagrama con 30.000 nodos.

dil
```
ddoc $dst_dir -hl --kandil -version=Tango -version=TangoDoc
-version=Posix -version=linux $tango_files
```

Genera la documentación de todo el código fuente de Tango 0.99.9, donde `$dst_dir` es el directorio donde almacenar los archivos generados y `$tango_files` es la lista de archivos fuente de Tango.

[5] El archivo contiene la Biblia completa, la versión traducida al inglés autorizada por el Rey Jaime o Jacobo (*Authorized King James Version* en inglés). Obtenida de: http://download.o-bible.com:8080/kjv.gz

El resto de los programas se ejecutan sin parámetros (ver *Banco de pruebas* para una descripción detallada sobre cada uno).

Recolectores y configuraciones utilizadas

En general se presentan resultados para TBGC y varias configuraciones de CDGC, de manera de poder tener una mejor noción de que mejoras y problemas puede introducir cada una de las modificaciones más importantes.

CDGC se utiliza con siguientes configuraciones:

cons
: En modo conservativo. Específicamente, utilizando el juego de opciones:

```
conservative=1:fork=0:early_collect=0:eager_alloc=0
```

prec
: En modo preciso (ver *Marcado preciso*). Específicamente, utilizando el juego de opciones:

```
conservative=0:fork=0:early_collect=0:eager_alloc=0
```

fork
: En modo preciso activando el marcado concurrente (ver *Marcado concurrente*). Específicamente, utilizando el juego de opciones:

```
conservative=0:fork=1:early_collect=0:eager_alloc=0
```

ecol
: En modo preciso activando el marcado concurrente con recolección temprana (ver *Recolección temprana (early collection)*). Específicamente, utilizando el juego de opciones:

```
conservative=0:fork=1:early_collect=1:eager_alloc=0
```

eall
: En modo preciso activando el marcado concurrente con creación ansiosa de *pools* (ver *Creación ansiosa de pools (eager allocation)*). Específicamente, utilizando el juego de opciones:

```
conservative=0:fork=1:early_collect=0:eager_alloc=1
```

todo
: En modo preciso activando el marcado concurrente con recolección temprana y creación ansiosa de *pools*. Específicamente, utilizando el juego de opciones:

```
conservative=0:fork=1:early_collect=1:eager_alloc=1
```

Métricas utilizadas

Para analizar los resultados se utilizan varias métricas. Las más importantes son:

- Tiempo total de ejecución.
- Tiempo máximo de *stop-the-world*.

- Tiempo máximo de pausa real.
- Cantidad máxima de memoria utilizada.
- Cantidad total de recolecciones realizadas.

El tiempo total de ejecución es una buena medida del **rendimiento** general del recolector, mientras que la cantidad total de recolecciones realizadas suele ser una buena medida de su **eficacia** [6].

Los tiempos máximos de pausa, *stop-the-world* y real, son una buena medida de la **latencia** del recolector; el segundo siendo una medida más realista dado que es raro que los demás hilos no utilicen servicios del recolector mientras hay una recolección en curso. Esta medida es particularmente importante para programas que necesiten algún nivel de ejecución en *tiempo-real*.

En general el consumo de tiempo y espacio es un compromiso, cuando se consume menos tiempo se necesita más espacio y viceversa. La cantidad máxima de memoria utilizada nos da un parámetro de esta relación.

Métodología de medición

Para medir el tiempo total de ejecución se utiliza el comando `time(1)` con la especificación de formato `%e`, siendo la medición más realista porque incluye el tiempo de carga del ejecutable, inicialización del *runtime* de D y del recolector.

Todas las demás métricas se obtienen utilizando la salida generada por la opción `collect_stats_file` (ver *Recolección de estadísticas*), por lo que no pueden ser medidos para TBGC. Sin embargo se espera que para esos casos los resultados no sean muy distintos a CDGC utilizando la configuración **cons** (ver sección anterior).

Cabe destacar que las corridas para medir el tiempo total de ejecución no son las mismas que al utilizar la opción `collect_stats_file`; cuando se mide el tiempo de ejecución no se utiliza esa opción porque impone un trabajo extra importante y perturbaría demasiado la medición del tiempo. Sin embargo, los tiempos medidos internamente al utilizar la opción `collect_stats_file` son muy precisos, dado que se hace un particular esfuerzo para que no se haga un trabajo extra mientras se está midiendo el tiempo.

Al obtener el tiempo de *stop-the-world* se ignoran los apariciones del valor `-1`, que indica que se solicitó una recolección pero que ya había otra en curso, por lo que no se pausan los hilos realmente. Como tiempo de pausa real (ver *Marcado concurrente* para más detalles sobre la diferencia con el tiempo de *stop-the-world*) se toma el valor del tiempo que llevó la asignación de memoria que disparó la recolección.

Para medir la cantidad de memoria máxima se calcula el valor máximo de la sumatoria de: memoria usada, memoria libre, memoria desperdiciada y memoria usada por el mismo recolector (es decir, el total de memoria pedida por el programa al sistema operativo, aunque no toda este siendo utilizada por el *mutator* realmente).

Por último, la cantidad total de recolecciones realizadas se calcula contando la cantidad de entradas del archivo generado por `collect_stats_file`, ignorando la cabecera y las filas cuyo valor de tiempo de *stop-the-world* es `-1`, debido a que en ese caso no se disparó realmente una recolección dado que ya había una en curso.

[6] Esto no es necesariamente cierto para recolectores con particiones (ver *Recolección por particiones / generacional*) o incrementales (ver *Recolección incremental*), dado que en ese caso podría realizar muchas recolecciones pero cada una muy velozmente.

Programa	Normal	-R	-L
bh	0.185	0.004	0.020
bigarr	0.012	0.002	0.016
bisort	0.006	0.003	0.006
conalloc	0.004	0.004	0.004
concpu	0.272	0.291	0.256
dil	0.198	0.128	0.199
em3d	0.006	0.033	0.029
mcore	0.009	0.009	0.014
rnddata	0.015	0.002	0.011
sbtree	0.012	0.002	0.012
split	0.025	0.000	0.004
tsp	0.071	0.068	0.703
voronoi	0.886	0.003	0.006

(a) Del tiempo total de ejecución.

Programa	Normal	-R	-L
bh	0.001	0.000	0.001
bigarr	0.001	0.000	0.001
bisort	0.000	0.000	0.000
conalloc	0.753	0.000	0.001
concpu	0.002	0.000	0.001
dil	0.055	0.028	0.013
em3d	0.000	0.001	0.001
mcore	0.447	0.482	0.460
rnddata	0.000	0.000	0.000
sbtree	0.000	0.000	0.000
split	0.000	0.000	0.000
tsp	0.000	0.001	0.000
voronoi	0.001	0.000	0.000

(b) Del consumo máximo de memoria.

Cuadro 5.2: Variación entre corridas para TBGC. La medición está efectuada utilizando los valores máximo, mínimo y media estadística de 20 corridas, utilizando la siguiente métrica: $\frac{max-min}{\mu}$. La medida podría realizarse utilizando el desvío estándar en vez de la amplitud máxima, pero en este cuadro se quiere ilustrar la variación máxima, no la típica.

Además, ciertas pruebas se corren variando la cantidad de procesadores utilizados, para medir el impacto de la concurrencia en ambientes con un procesador solo y con múltiples procesadores. Para esto se utiliza el comando `taskset(1)`, que establece la *afinidad* de un proceso, *atándolo* a correr en un cierto conjunto de procesadores. Si bien las pruebas se realizan utilizando 1, 2, 3 y 4 procesadores, los resultados presentados en general se limitan a 1 y 4 procesadores, ya que no se observan diferencias sustanciales al utilizar 2 o 3 procesadores con respecto a usar 4 (solamente se ven de forma más atenuadas las diferencias entre la utilización de 1 o 4 procesadores). Dado que de por sí ya son muchos los datos a procesar y analizar, agregar más resultados que no aportan información valiosa termina resultando contraproducente.

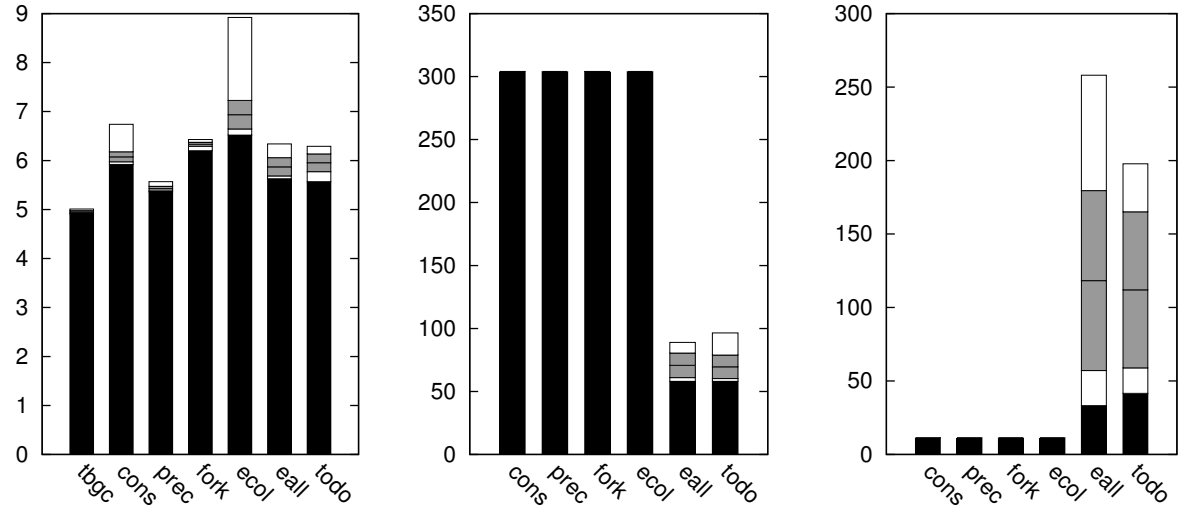
En los casos donde se utilizan otro tipo de métricas para evaluar aspectos particulares sobre alguna modificación se describe como se realiza la medición donde se utiliza la métrica especial.

Variabilidad de los resultados entre ejecuciones

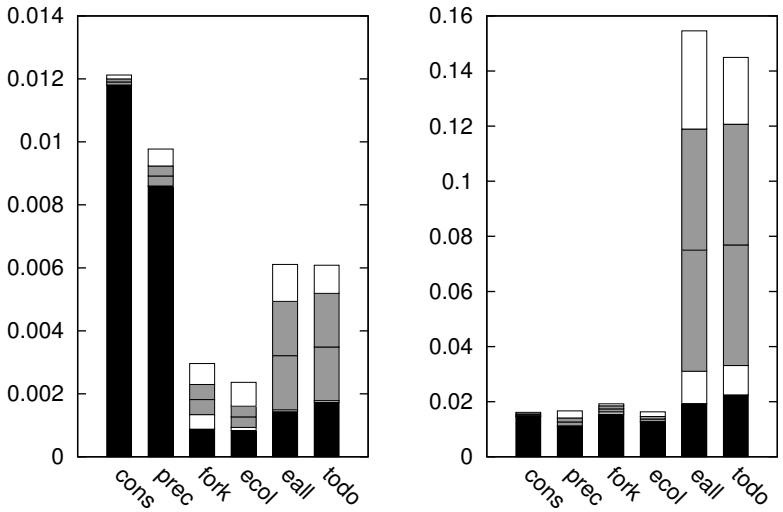
Es de esperarse que haya una cierta variación en los resultados entre corridas, dada la indeterminación inherente a los sistemas operativos de tiempo compartido, que compiten por los recursos de la computadora.

Para minimizar esta variación se utilizan varias herramientas. En primer lugar, se corren las pruebas estableciendo máxima prioridad (-19 en Linux) al proceso utilizando el comando `nice(1)`. La variación en la frecuencia del reloj los procesadores (para ahorrar energía) puede ser otra fuente de variación, por lo que se usa el comando `cpufreq-set(1)` para establecer la máxima frecuencia disponible de manera fija. El acceso a disco, en las pruebas que utilizan archivos, puede ser otro factor importante, por lo tanto se utiliza `ionice(1)` para darle prioridad de entrada/salida de tiempo real al proceso y se realizan las pruebas con el caché de disco en *caliente*.

Sin embargo, a pesar de tomar estas precauciones, se sigue observando una amplia variabilidad entre corridas. Además se observa una variación más importante de la esperada no solo en el tiempo, también en el consumo de memoria, lo que es más extraño. Esta variación se debe principalmente a que Linux



(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)



(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.4: Resultados para `bigarr` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

asigna el espacio de direcciones a los procesos con una componente azarosa (por razones de seguridad). Además, por omisión, la llamada al sistema *mmap(2)* asigna direcciones de memoria altas primero, entregando direcciones más bajas en llamadas subsiguientes [LWN90311].

El comando *setarch(8)* sirve para controlar éste y otros aspectos de Linux. La opción `-L` hace que se utilice un esquema de asignación de direcciones antiguo, que no tiene una componente aleatoria y asigna primero direcciones bajas. La opción `-R` solamente desactiva la componente azarosa al momento de asignar direcciones.

Ambas opciones, reducen notablemente la variación en los resultados (ver cuadro 5.2 en la página 109). Esto probablemente se debe a la naturaleza conservativa del recolector, dado que la probabilidad de tener *falsos positivos* depende directamente de los valores de las direcciones de memoria, aunque las pruebas en la que hay concurrencia involucrada, se siguen viendo grandes variaciones, que probablemente estén vinculadas a problemas de sincronización que se ven expuestos gracias al indeterminismo inherente a los programas multi-hilo.

Si bien se obtienen resultados más estables utilizando un esquema diferente al utilizado por omisión, se decide no hacerlo dado que las mediciones serían menos realistas. Los usuarios en general no usan esta opción y se presentaría una visión más acotada sobre el comportamiento de los programas. Sin embargo, para evaluar el este efecto en los resultados, siempre que sea posible se analizan los resultados de un gran número de corridas observando principalmente su mínima, media, máxima y desvío estándar.

5.3.2 Resultados para pruebas sintizadas

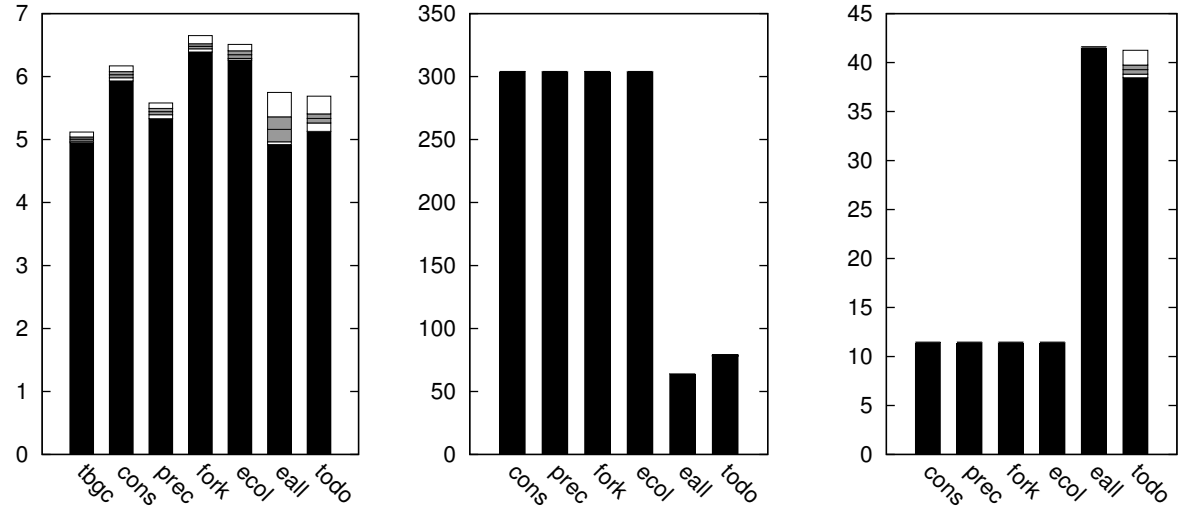
A continuación se presentan los resultados obtenidos para las pruebas sintetizadas (ver *Pruebas sintetizadas*). Se recuerda que este conjunto de resultados es útil para analizar ciertos aspectos puntuales de las modificaciones propuestas, pero en general distan mucho de como se comporta un programa real, por lo que los resultados deben ser analizados teniendo esto presente.

bigarr

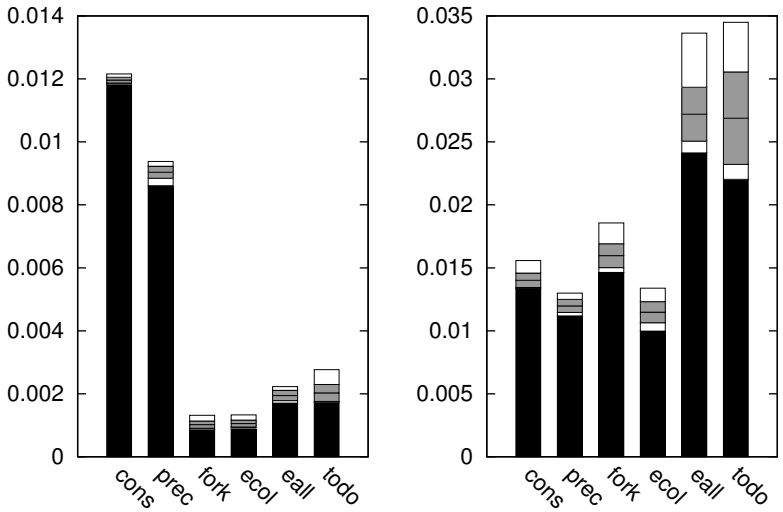
En la figura 5.4 en la página anterior se pueden observar los resultados para `bigarr` al utilizar un solo procesador. En ella se puede notar que el tiempo total de ejecución en general aumenta al utilizar CDGC, esto es esperable, dado esta prueba se limitan a usar servicios del recolector. Dado que esta ejecución utiliza solo un procesador y por lo tanto no se puede sacar provecho a la concurrencia, es de esperarse que el trabajo extra realizado por las modificaciones se vea reflejado en los resultados. En la 5.5 en la página siguiente (resultados al utilizar 4 procesadores) se puede observar como al usar solamente *eager allocation* se recupera un poco el tiempo de ejecución, probablemente debido al incremento en la concurrencia (aunque no se observa el mismo efecto al usar *early collection*).

Observando el tiempo total de ejecución, no se esperaba un incremento tan notorio al pasar de TBGC a una configuración equivalente de CDGC **cons**, haciendo un breve análisis de las posibles causas, lo más probable parece ser el incremento en la complejidad de la fase de marcado dada capacidad para marcar de forma precisa (aunque no se use la opción, se paga el precio de la complejidad extra y sin obtener los beneficios). Además se puede observar como el agregado de precisión al marcado mejora un poco las cosas (donde sí se obtiene rédito de la complejidad extra en el marcado).

En general se observa que al usar *eager allocation* el consumo de memoria y los tiempos de pausa se disparan mientras que la cantidad de recolecciones disminuye drásticamente. Lo que se observa es que el programa es más veloz pidiendo memoria que recolectándola, por lo que crece mucho el consumo de memoria. Como consecuencia la fase de barrido (que no corre en paralelo al *mutator* como la fase de marcado) empieza a ser predominante en el tiempo de pausa por ser tan grande la cantidad de memoria



(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)



(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.5: Resultados para `bigarr` (utilizando 4 procesadores). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

a barrer. Este efecto se ve tanto al usar 1 como 4 procesadores, aunque el efecto es mucho más nocivo al usar 1 debido a la alta variabilidad que impone la competencia entre el *mutator* y recolector al correr de forma concurrente.

Sin embargo, el tiempo de *stop-the-world* es siempre considerablemente más pequeño al utilizar marcado concurrente en CDGC, incluso cuando se utiliza *eager allocation*, aunque en este caso aumenta un poco, también debido al incremento en el consumo de memoria, ya que el sistema operativo tiene que copiar tablas de memoria más grandes al efectuar el *fork* (ver *Marcado concurrente*).

concpu

En la figura 5.6 en la página siguiente se pueden observar los resultados para `concpu` al utilizar un solo procesador. En ella se aprecia que el tiempo total de ejecución disminuye levemente al usar marcado concurrente mientras no se utilice *eager allocation* (si se utiliza vuelve a aumentar, incluso más que sin marcado concurrente).

Con respecto a la cantidad de recolecciones, uso máximo de memoria y tiempo de *stop-the-world* se ve un efecto similar al descripto para `bigarr` (aunque magnificado), pero sorprendentemente el tiempo total de pausa se dispara, además con una variabilidad sorprendente, cuando se usa marcado concurrente (pero no *eager allocation*). Una posible explicación podría ser que al realizarse el *fork*, el sistema operativo muy probablemente entregue el control del único procesador disponible al resto de los hilos que compiten por él, por lo que queda mucho tiempo pausado en esa operación aunque realmente no esté haciendo trabajo alguno (simplemente no tiene tiempo de procesador para correr). Este efecto se cancela al usar *eager allocation* dado que el *mutator* nunca se bloquea esperando que el proceso de marcado finalice.

Además se observa una caída importante en la cantidad de recolecciones al utilizar marcado concurrente. Esto probablemente se deba a que solo un hilo pide memoria (y por lo tanto dispara recolecciones), mientras los demás hilos también estén corriendo. Al pausarse todos los hilos por menos tiempo, el trabajo se hace más rápido (lo que explica la disminución del tiempo total de ejecución) y son necesarias menos recolecciones, por terminar más rápido también el hilo que las dispara.

En la 5.7 en la página 116 se pueden ver los resultados al utilizar 4 procesadores, donde el panorama cambia sustancialmente. El efecto mencionado en el párrafo anterior no se observa más (pues el sistema operativo tiene más procesadores para asignar a los hilos) pero todos los resultados se vuelven más variables. Los tiempos de *stop-the-world* y pausa real (salvo por lo recién mencionado) crecen notablemente, al igual que su variación. No se encuentra una razón evidente para esto; podría ser un error en la medición dado que al utilizar todos los procesadores disponibles del *hardware*, cualquier otro proceso que compita por tiempo de procesador puede afectarla más fácilmente.

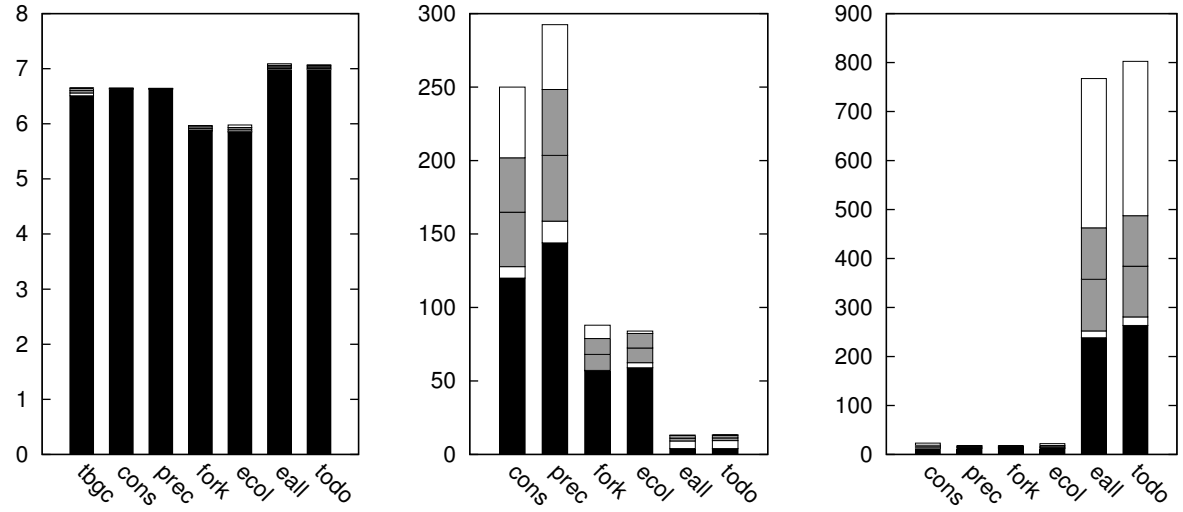
El tiempo total de ejecución crece considerablemente, como se espera, dado que el programa aprovecha los múltiples hilos que pueden correr en paralelo en procesadores diferentes.

Sin embargo, no se encuentra una razón clara para explicar el crecimiento dramático en la cantidad de recolecciones solo al no usar marcado concurrente para 4 procesadores.

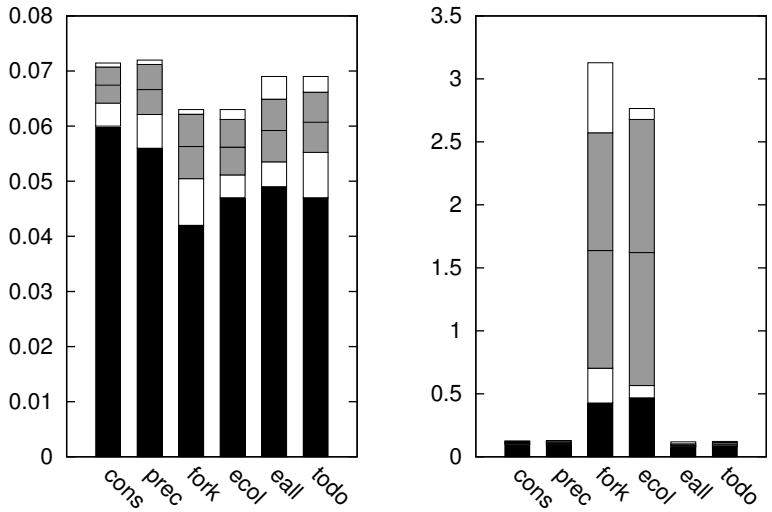
conalloc

En la figura 5.8 en la página 117 se pueden observar los resultados para `conalloc` al utilizar un solo procesador. Los cambios con respecto a lo observado para `concpu` son mínimos. El efecto de la mejoría al usar marcado concurrente pero no *eager allocation* no se observa más, dado que `conalloc` pide memoria en todos los hilos, se crea un cuello de botella. Se ve claramente como tampoco baja la cantidad de recolecciones hecha debido a esto y se invierte la variabilidad entre los tiempos pico de pausa real y *stop-the-world* (sin una razón obvia, pero probablemente relacionado que todos los hilos piden memoria).

Al utilizar 4 procesadores (figura 5.9 en la página 118), más allá de las diferencias mencionadas para 1 procesador, no se observan grandes cambios con respecto a lo observado para `concpu`, excepto que los tiempos de pausa (real y *stop-the-world*) son notablemente más pequeños, lo que pareciera confirmar un error en la medición de `concpu`.

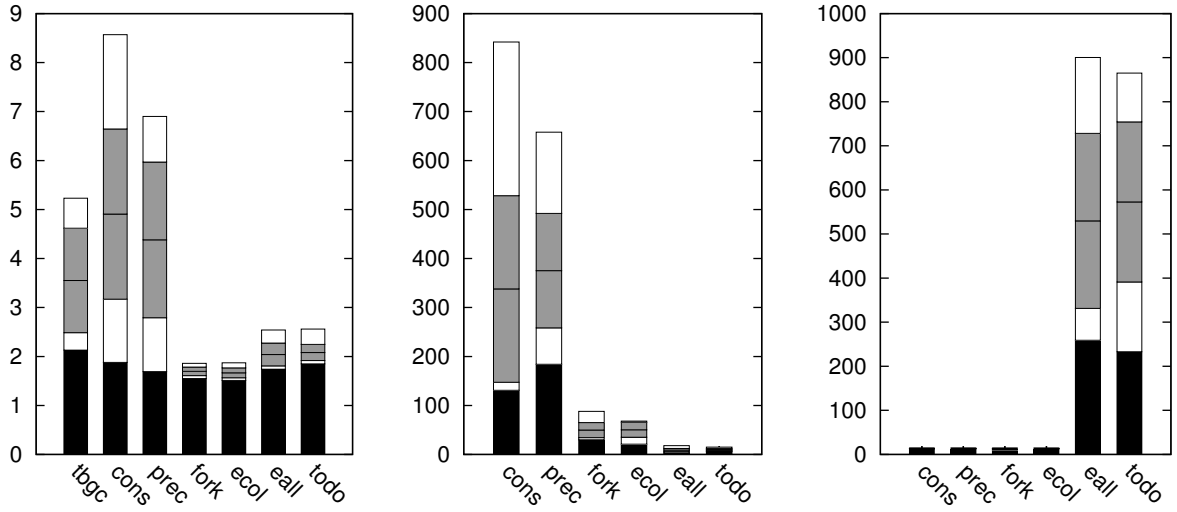


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

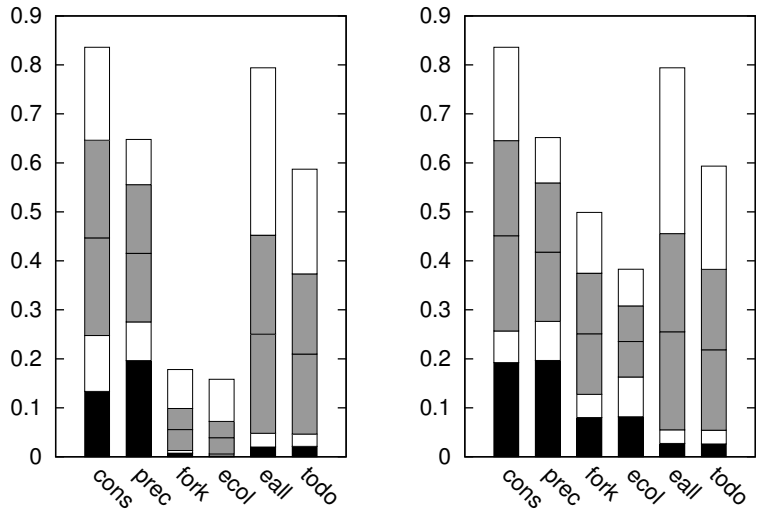


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.6: Resultados para `concpu` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

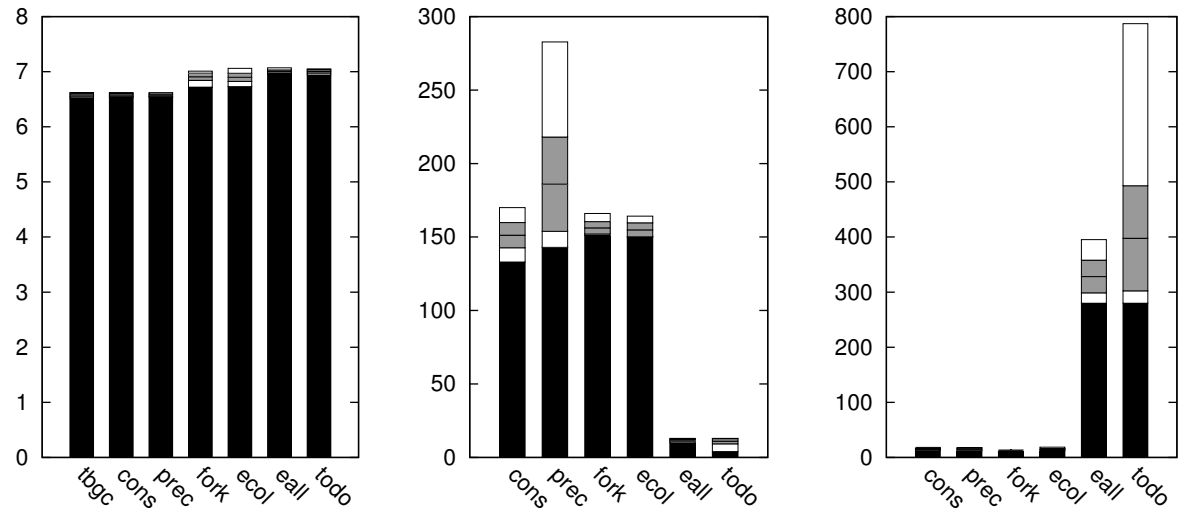


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

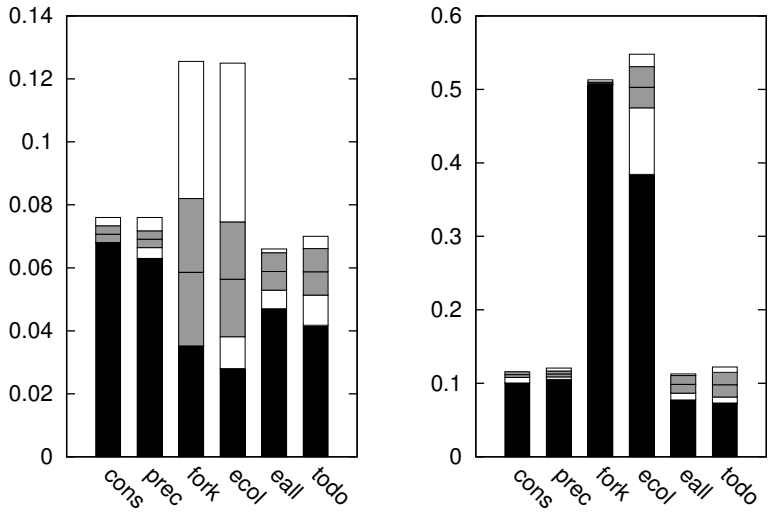


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.7: Resultados para `concpu` (utilizando 4 procesadores). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

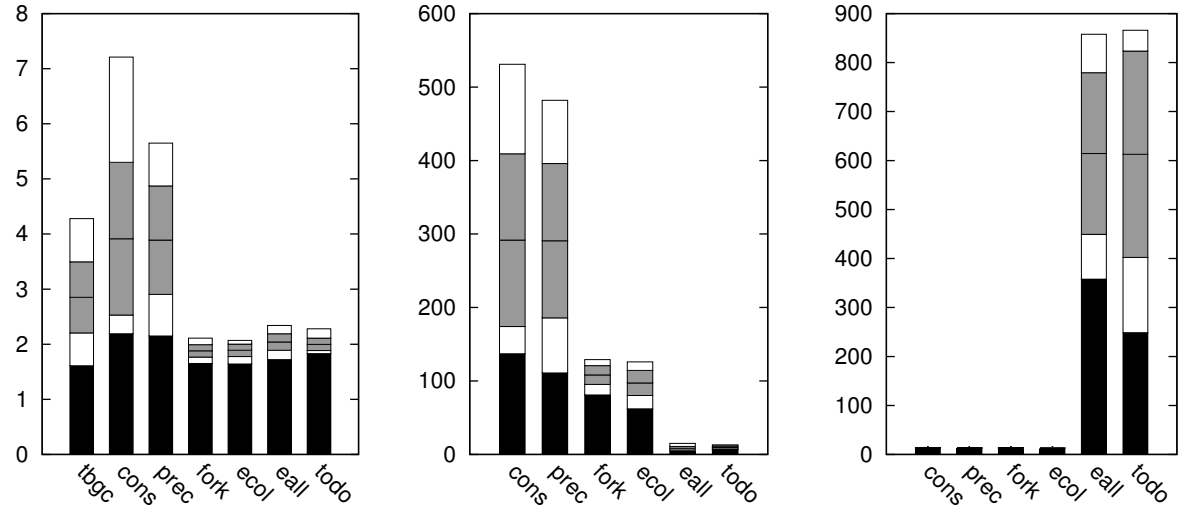


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

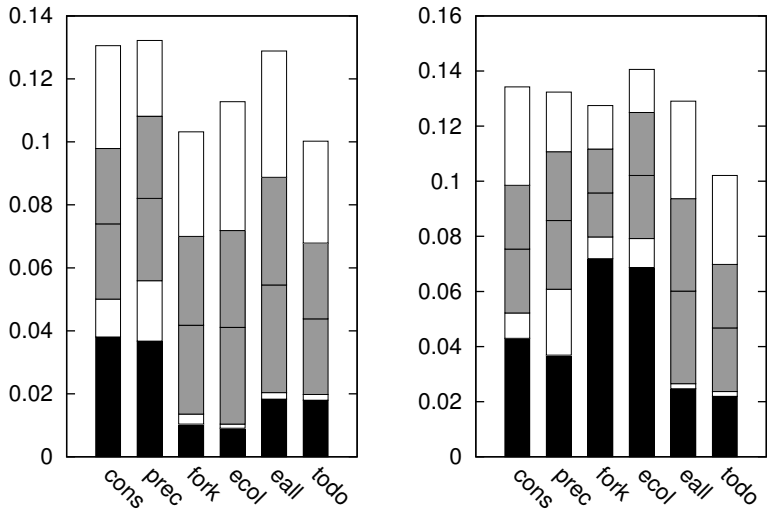


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.8: Resultados para `conalloc` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

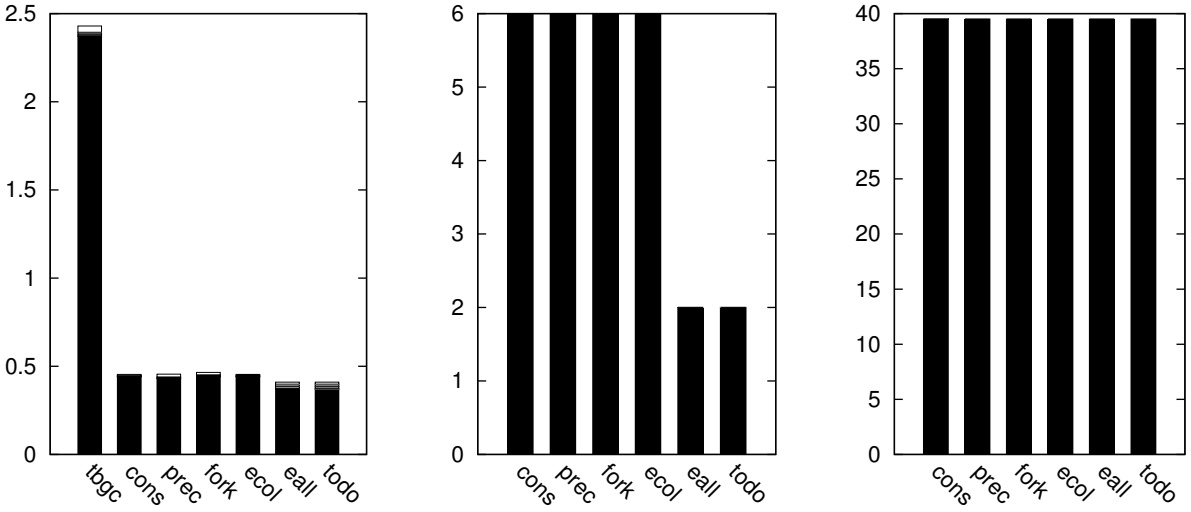


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

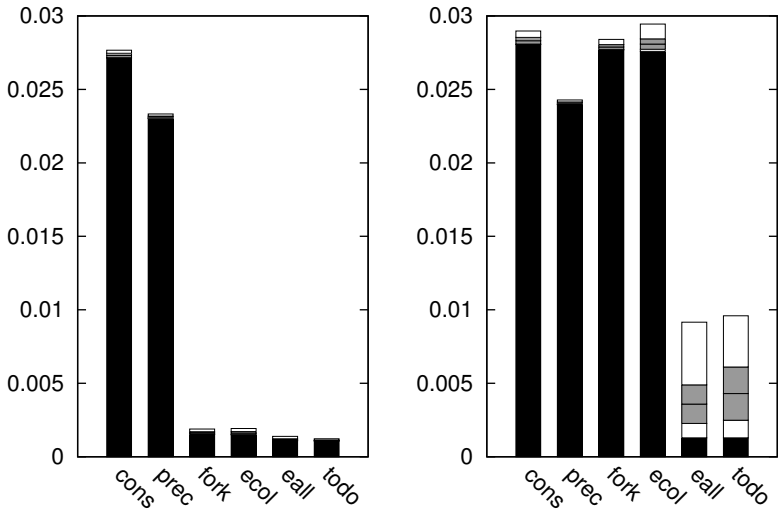


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.9: Resultados para `conalloc` (utilizando 4 procesadores). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).



(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)



(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.10: Resultados para `split` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

split

Este es el primer caso donde se aprecia la sustancial mejora proporcionada por una pequeña optimización, el caché de `findSize()` (ver *Caché de Pool.findSize()*). En la figura 5.10 en la página anterior se puede observar con claridad como, para cualquier configuración de CDGC, hay una caída notable en el tiempo total de ejecución. Sin embargo, a excepción de cuando se utiliza *eager allocation*, la cantidad de recolecciones y memoria usada permanece igual.

La utilización de *eager allocation* mejora (aunque de forma apenas apreciable) el tiempo de ejecución, la cantidad de recolecciones baja a un tercio y el tiempo de pausa real cae dramáticamente. Al usar marcado concurrente ya se observa una caída determinante en el tiempo de *stop-the-world*. Todo esto sin verse afectado el uso máximo de memoria, incluso al usar *eager allocation*.

Se omiten los resultados para más de un procesador por ser prácticamente idénticos para este análisis.

mcore

El caso de `mcore` es interesante por ser, funcionalmente, una combinación entre `concpu` y `split`, con un agregado extra: el incremento notable de la competencia por utilizar el recolector entre los múltiples hilos.

Los efectos observados (en la figura 5.11 en la página siguiente para 1 procesador y en la figura 5.12 en la página 123 para 4) confirman esto, al ser una suma de los efectos observados para `concpu` y `split`, con el agregado de una particularidad extra por la mencionada competencia entre hilos. A diferencia de `concpu` donde el incremento de procesadores resulta en un decremento en el tiempo total de ejecución, en este caso resulta en un incremento, dado que se necesita mucha sincronización entre hilos, por utilizar todos de forma intensiva los servicios del recolector (y por lo tanto competir por su *lock* global).

Otro efecto común observado es que cuando el tiempo de pausa es muy pequeño (del orden de los milisegundos), el marcado concurrente suele incrementarlo en vez de disminuirlo.

rnddata

En la figura 5.13 en la página 124 se presentan los resultados para `rnddata` utilizando 1 procesador. Una vez más estamos ante un caso en el cual se observa claramente la mejoría gracias a una modificación en particular principalmente. En esta caso es el marcado preciso. Se puede ver claramente como mejora el tiempo de total de ejecución a algo más que la mitad (en promedio, aunque se observa una anomalía donde el tiempo baja hasta más de 3 veces). Sin embargo, a menos que se utilice *eager allocation* o *early collection* (que en este caso prueba ser muy efectivo), la cantidad de recolecciones aumenta considerablemente.

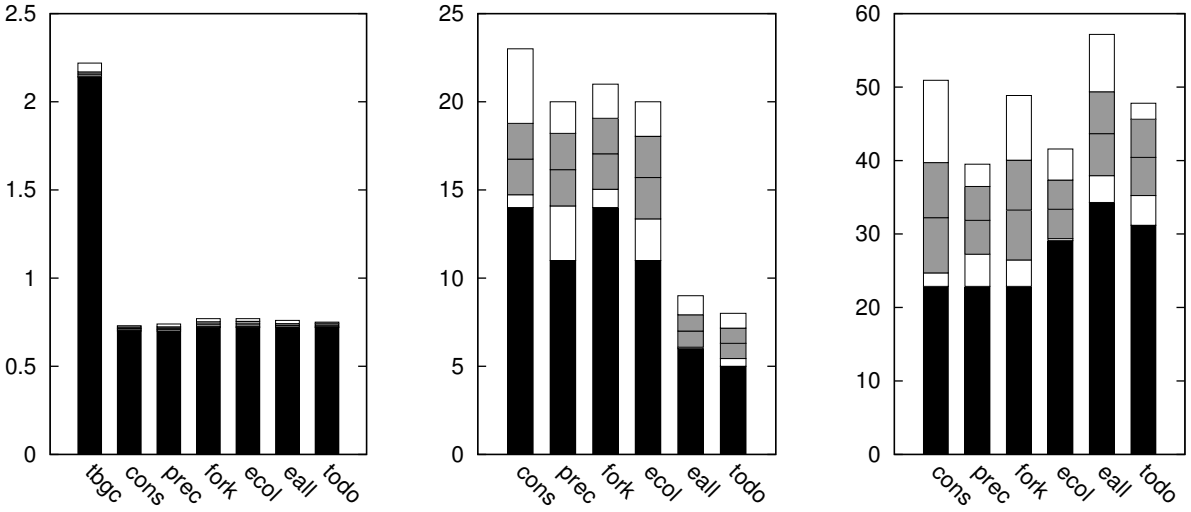
La explicación puede ser hallada en el consumo de memoria, que baja unas 3 veces en promedio usando marcado preciso que además hace disminuir drásticamente (unas 10 veces) el tiempo de pausa (real y *stop-the-world*). El tiempo de *stop-the-world* disminuye unas 10 veces más al usar marcado concurrente y el tiempo de pausa real al usar *eager allocation*, pero en este caso el consumo de memoria aumenta también bastante (aunque no tanto como disminuye el tiempo de pausa, por lo que puede ser un precio que valga la pena pagar si se necesitan tiempos de pausa muy pequeños).

El aumento en el variación de los tiempos de ejecución al usar marcado preciso probablemente se debe a lo siguiente: con marcado conservativo, debe estar sobreviviendo a las recolecciones el total de memoria pedida por el programa, debido a *falsos positivos* (por eso no se observa prácticamente variación en el tiempo de ejecución y memoria máxima consumida); al marcar con precisión parcial, se logra disminuir mucho la cantidad de *falsos positivos*, pero el *stack* y la memoria estática, se sigue marcado de forma conservativa, por lo tanto dependiendo de los valores (aleatorios) generados por la prueba, aumenta o disminuye la cantidad de *falsos positivos*, variando así la cantidad de memoria consumida y el tiempo de ejecución.

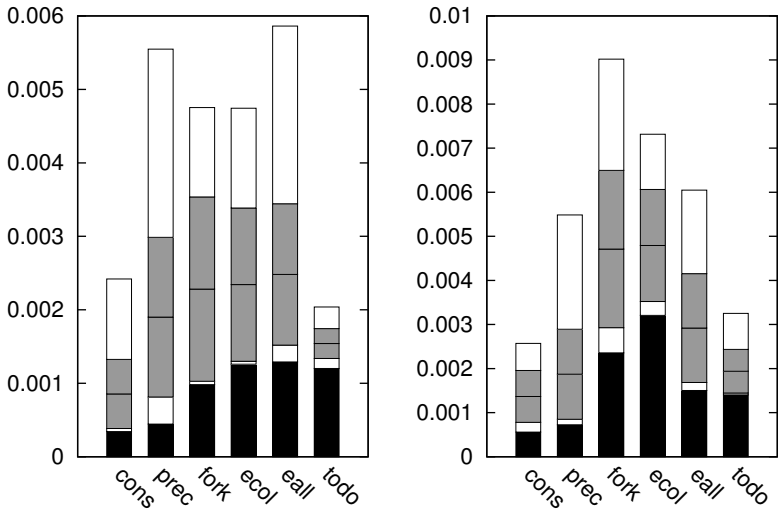
No se muestran los resultados para más de un procesador por ser demasiado similares a los obtenidos utilizando solo uno.

sbtree

Los resultados para `sbtree` son tan similares a los obtenidos con `bigarr` que directamente se omiten por completo, dado que no aportan ningún tipo de información nueva. Por un lado es esperable, dado que ambas pruebas se limitan prácticamente a pedir memoria, la única diferencia es que una pide objetos grandes y otra objetos pequeños, pero esta diferencia parece no afectar la forma en la que se comportan los cambios introducidos en este trabajo.

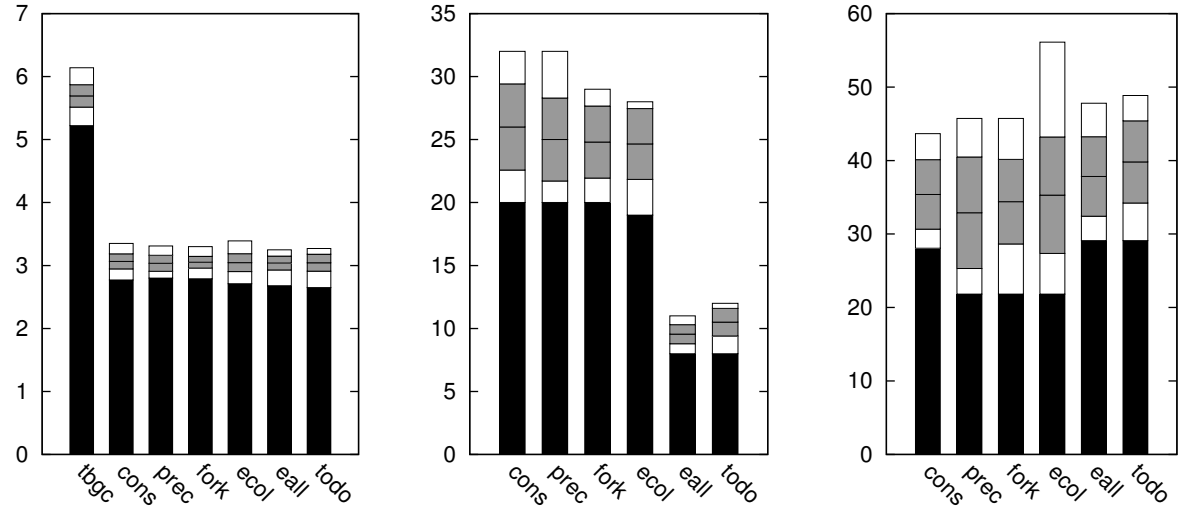


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

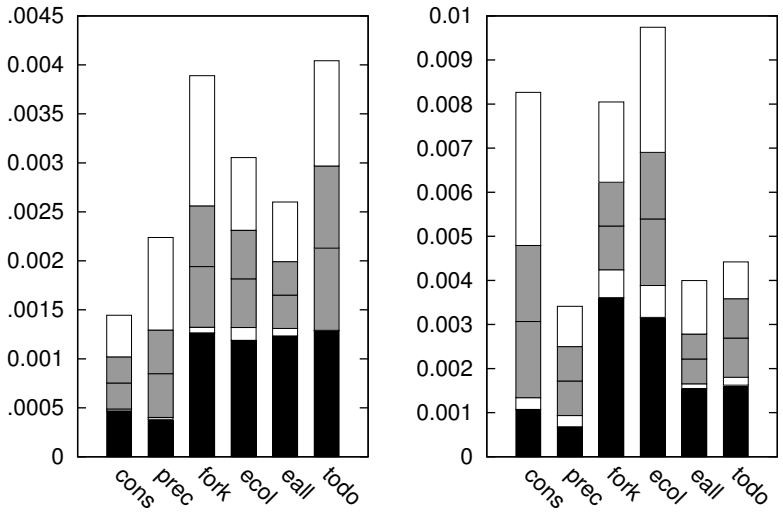


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.11: Resultados para `mcore` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

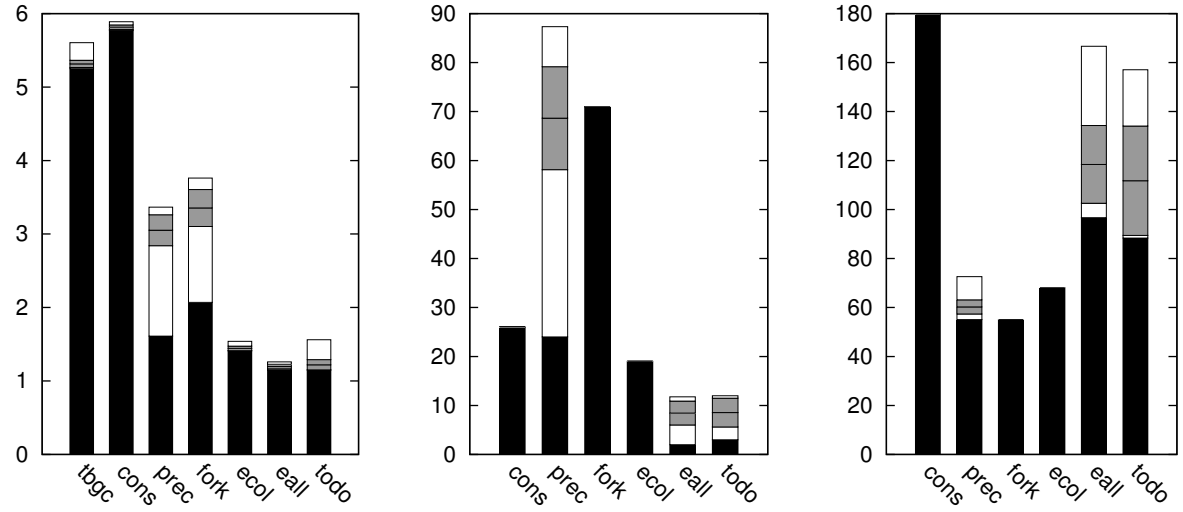


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

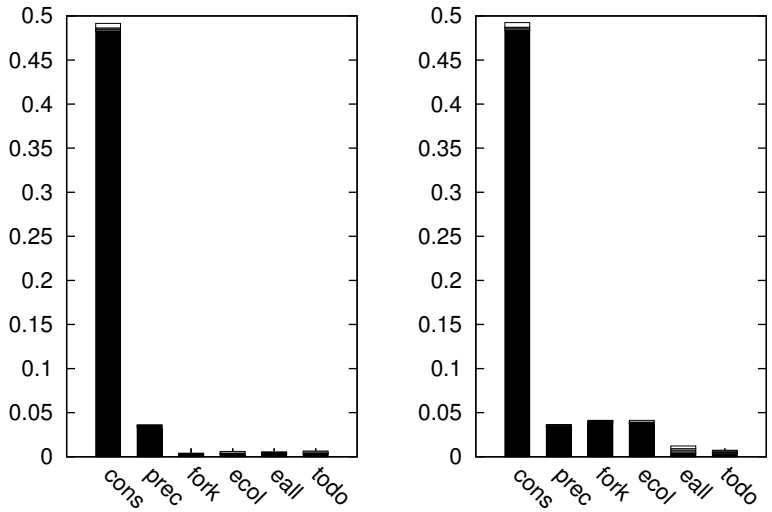


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.12: Resultados para `mcore` (utilizando 4 procesadores). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

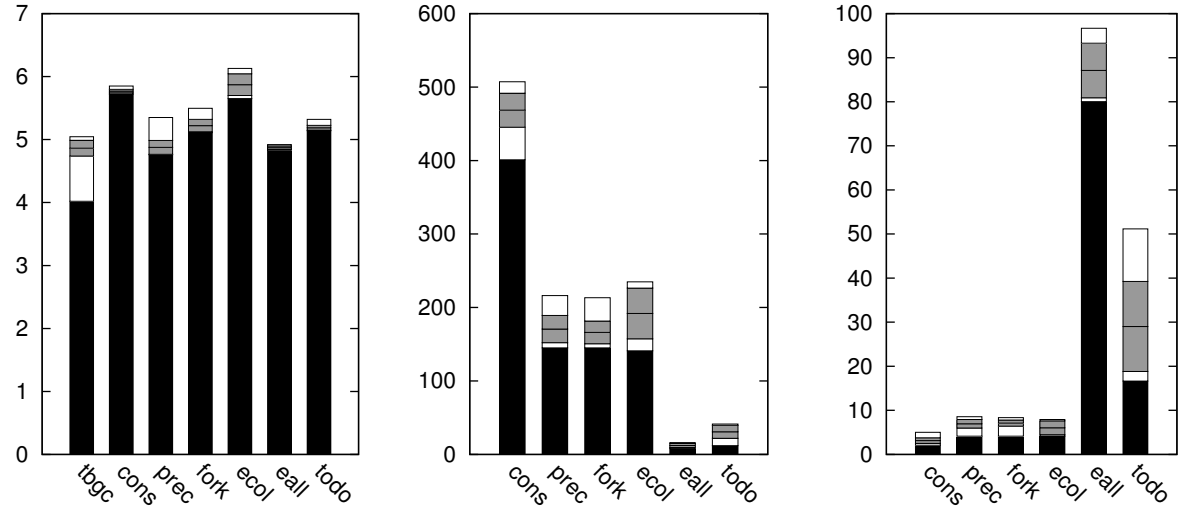


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

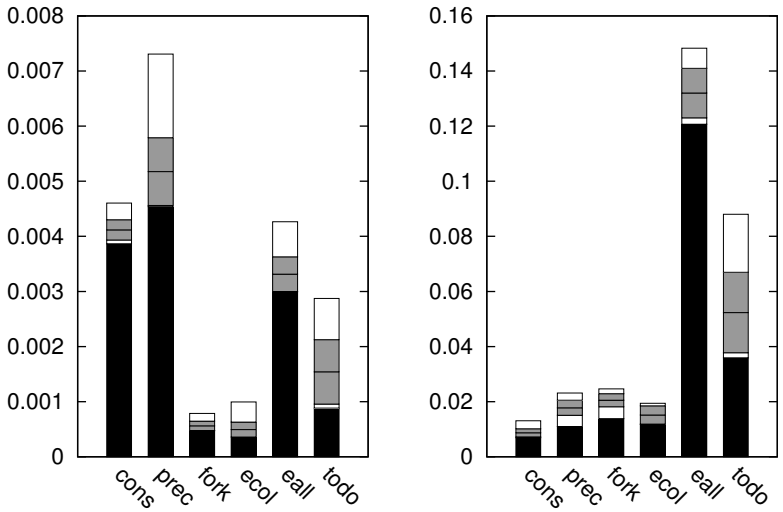


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.13: Resultados para `rnddata` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).



(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)



(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.14: Resultados para `bh` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

Memoria	Pedida (MiB)	Asignada (MiB)	Desperdicio (MiB)
Conservativo	302.54	354.56	52.02 (15 %)
Preciso	302.54	472.26	169.72 (36 %)

Cuadro 5.3: Memoria pedida y asignada para `bh` según modo de marcado conservativo o preciso (acumulativo durante toda la vida del programa).

5.3.3 Resultados para pruebas pequeñas

A continuación se presentan los resultados obtenidos para las pruebas pequeñas (ver *Programas pequeños*). Se recuerda que si bien este conjunto de pruebas se compone de programas reales, que efectúan una tarea útil, están diseñados para ejercitar la asignación de memoria y que no son recomendados para evaluar el desempeño de recolectores de basura. Sin embargo se las utiliza igual por falta de programas más realistas, por lo que hay que tomarlas como un grado de suspicacia.

`bh`

En la figura 5.14 en la página anterior se pueden observar los resultados para `bh` al utilizar un solo procesador. Ya en una prueba un poco más realista se puede observar el efecto positivo del marcado preciso, en especial en la cantidad de recolecciones efectuadas (aunque no se traduzca en un menor consumo de memoria).

Sin embargo se observa también un efecto nocivo del marcado preciso en el consumo de memoria que intuitivamente debería disminuir, pero crece, y de forma considerable (unas 3 veces en promedio). La razón de esta particularidad es el incremento en el espacio necesario para almacenar objetos debido a que el puntero a la información del tipo se guarda al final del bloque (ver *Marcado preciso*). En el cuadro 5.3 se puede observar la cantidad de memoria pedida por el programa, la cantidad de memoria realmente asignada por el recolector (y la memoria desperdiciada) cuando se usa marcado conservativo y preciso. Estos valores fueron tomados usando la opción `malloc_stats_file` (ver *Recolección de estadísticas*).

Más allá de esto, los resultados son muy similares a los obtenidos para pruebas sintetizadas que se limitan a ejercitar el recolector (como `bigarr` y `sbtree`), lo que habla de lo mucho que también lo hace este pequeño programa.

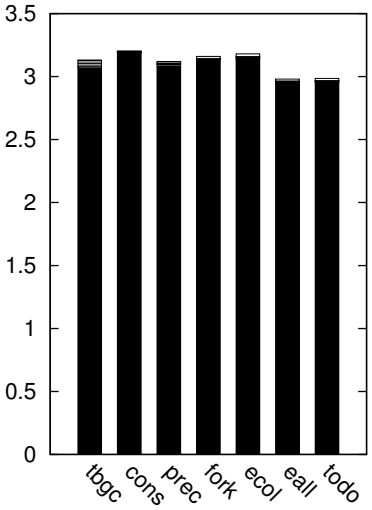
No se muestran los resultados para más de un procesador por ser extremadamente similares a los obtenidos utilizando solo uno.

`bisort`

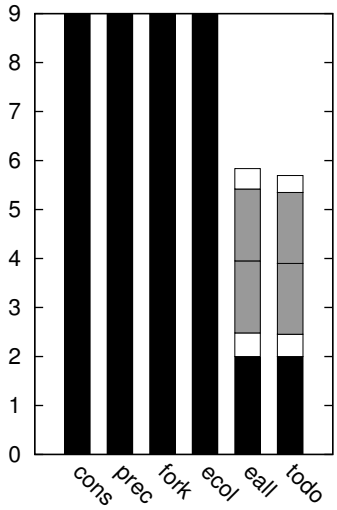
La figura 5.15 en la página siguiente muestra los resultados para `bisort` al utilizar 1 procesador. En este caso el parecido es con los resultados para la prueba sintetizada `split`, con la diferencia que el tiempo de ejecución total prácticamente no varía entre TBGC y CDGC, ni entre las diferentes configuraciones del último (evidentemente en este caso no se aprovecha el caché de `findSize()`).

Otra diferencia notable es la considerable reducción del tiempo de pausa real al utilizar *early collection* (más de 3 veces menor en promedio comparado a cuando se marca de forma conservativa, y más de 2 veces menor que cuando se hace de forma precisa), lo que indica que la predicción de cuando se va a necesitar una recolección es más efectiva que para `split`.

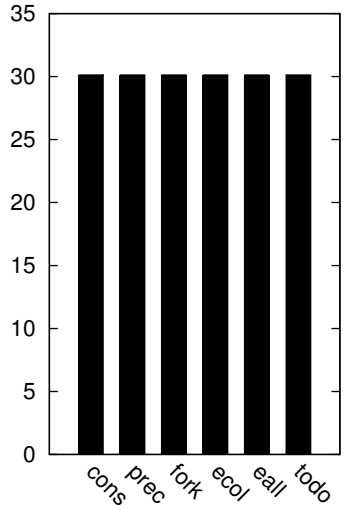
No se muestran los resultados para más de un procesador por ser extremadamente similares a los obtenidos utilizando solo uno.



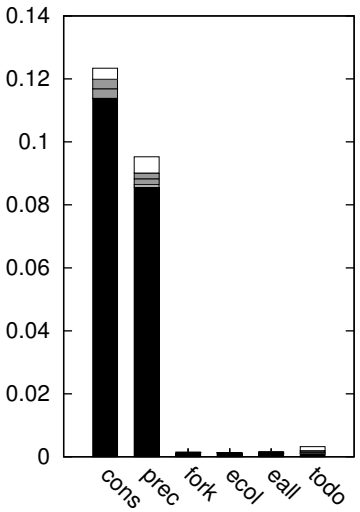
(a) Tiempo de ejecución (seg)



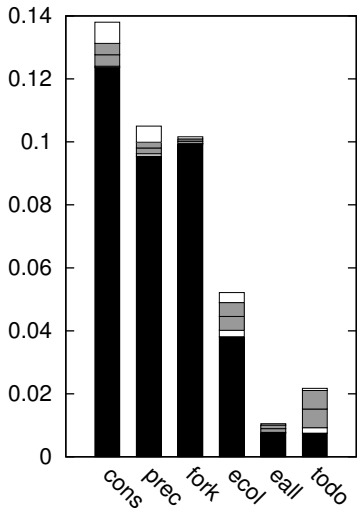
(b) Cantidad de recolecciones



(c) Uso máximo de memoria (MiB)



(d) *Stop-the-world* máximo (seg)



(e) Pausa real máxima (seg)

Figura 5.15: Resultados para `bisort` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

em3d

Los resultados para `em3d` (figura 5.16 en la página siguiente) son sorprendentemente similares a los de `bisort`. La única diferencia es que en este caso el marcado preciso y el uso de *early collection* no parecen ayudar; por el contrario, aumentan levemente el tiempo de pausa real.

Una vez más no se muestran los resultados para más de un procesador por ser extremadamente similares a los obtenidos utilizando solo uno.

tsp

Los resultados para `tsp` (figura 5.17 en la página 130) son prácticamente idénticos a los de `bisort`. La única diferencia es que la reducción del tiempo de pausa real es un poco menor.

Esto confirma en cierta medida la poca utilidad de este juego de pruebas para medir el rendimiento de un recolector, dado que evidentemente, si bien todas resuelven problemas diferentes, realizan todas el mismo tipo de trabajo.

Una vez más no se muestran los resultados para más de un procesador por ser extremadamente similares a los obtenidos utilizando solo uno.

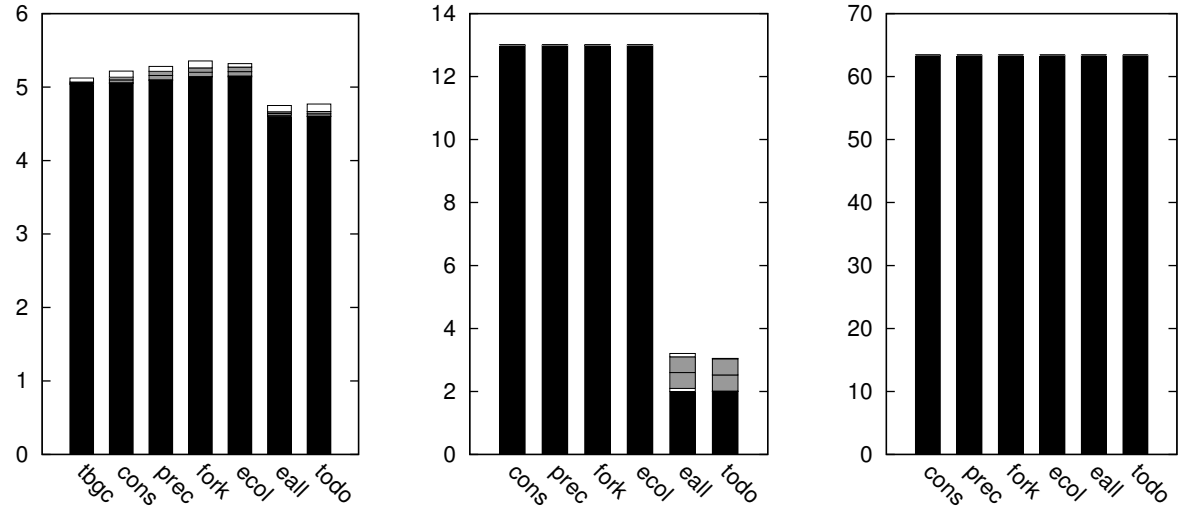
voronoi

En la figura 5.18 en la página 131 se presentan los resultados para `voronoi`, probablemente la prueba más interesante de este conjunto de pruebas pequeñas.

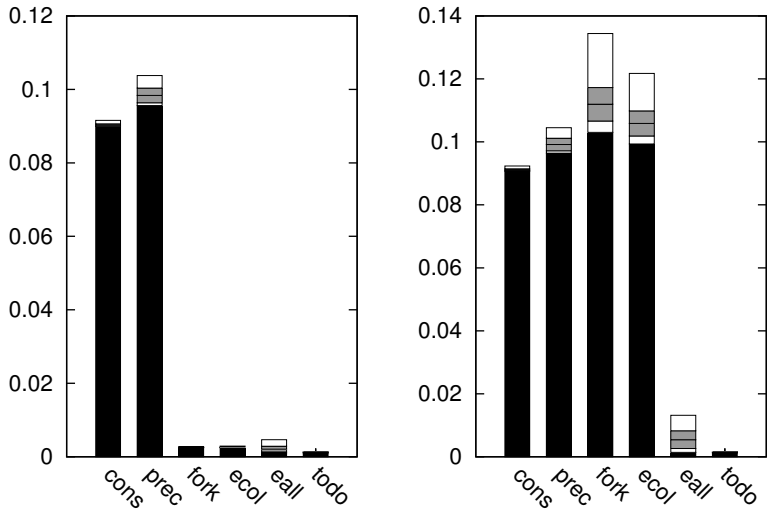
Por un lado se puede observar una vez más como baja dramáticamente el tiempo total de ejecución cuando se empieza a utilizar CDGC. Ya se ha visto que esto es común en programas que se benefician del caché de `findSize()`, pero en este caso no parece provenir toda la ganancia solo de ese cambio, dado que para TBGC se ve una variación entre los resultados muy grande que desaparece al cambiar a CDGC, esto no puede ser explicado por esa optimización. En general la disminución de la variación de los resultados hemos visto que está asociada al incremento en la precisión en el marcado, dado que los *falsos positivos* ponen una cuota de aleatoriedad importante. Pero este tampoco parece ser el caso, ya que no se observan cambios apreciables al pasar a usar marcado preciso.

Lo que se observa en esta oportunidad es un caso patológico de un mal factor de ocupación del *heap* (ver *Mejora del factor de ocupación del heap*). Lo que muy probablemente está sucediendo con TBGC es que luego de ejecutar una recolección, se libera muy poco espacio, entonces luego de un par de asignaciones, es necesaria una nueva recolección. En este caso es donde dificulta la tarea de analizar los resultados la falta de métricas para TBGC, dado que no se pueden observar la cantidad de recolecciones ni de consumo máximo de memoria. Sin embargo es fácil corroborar esta teoría experimentalmente, gracias a la opción `min_free`. Utilizando la `min_free=0` para emular el comportamiento de TBGC (se recuerda que el valor por omisión es `min_free=5`), se obtiene una media de 4 segundos, mucho más parecida a lo obtenido para TBGC.

Otra particularidad de esta prueba es que al utilizar *early collection* el tiempo de pausa real aumenta notablemente al usar un procesador, mientras que al usar 4 (ver figura 5.19 en la página 132 disminuye levemente (además de otros cambios en el nivel de variación, pero en general las medias no cambian).

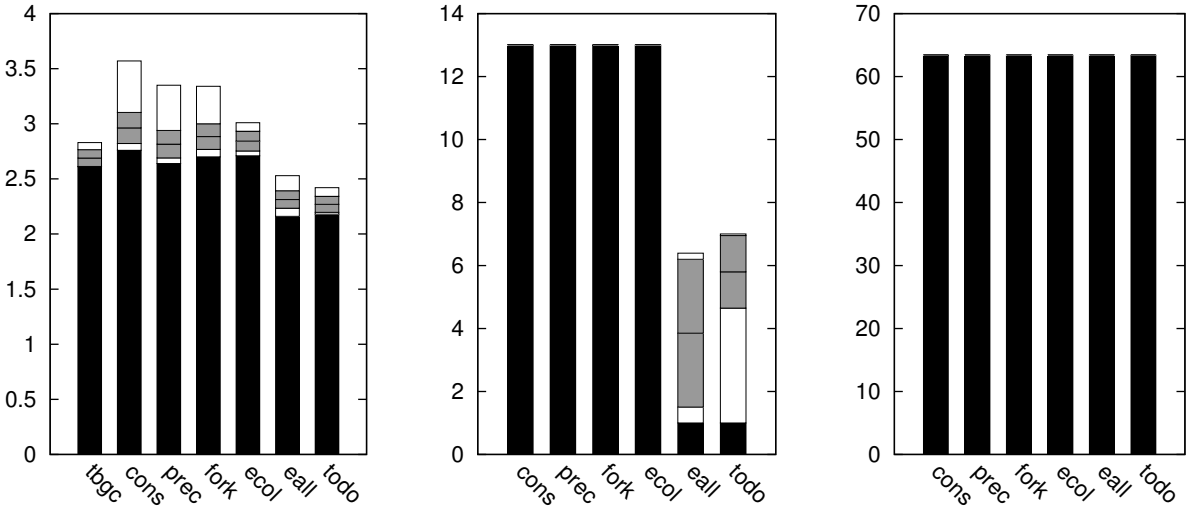


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

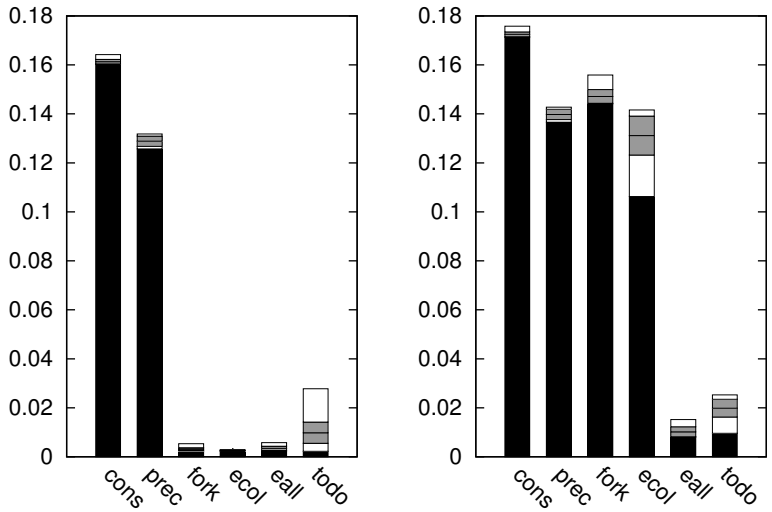


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.16: Resultados para `em3d` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

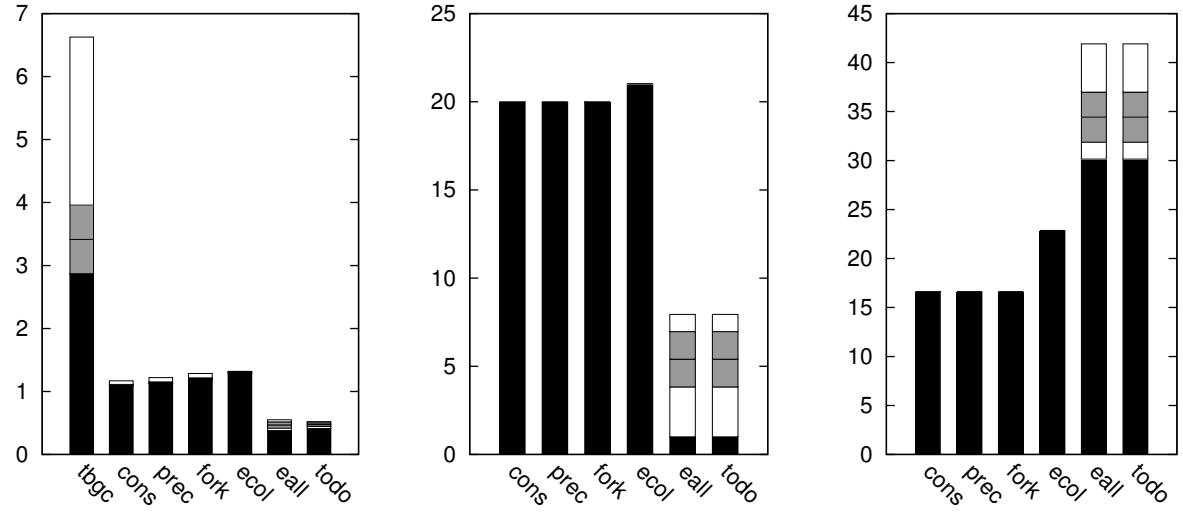


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

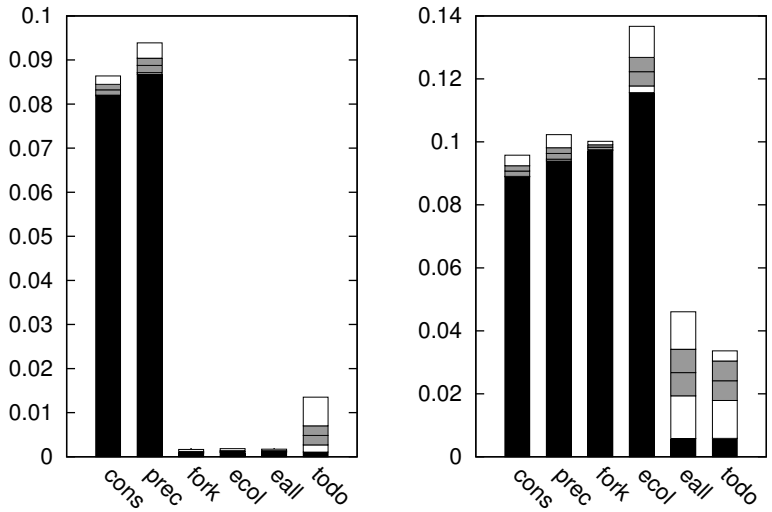


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.17: Resultados para `tsp` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

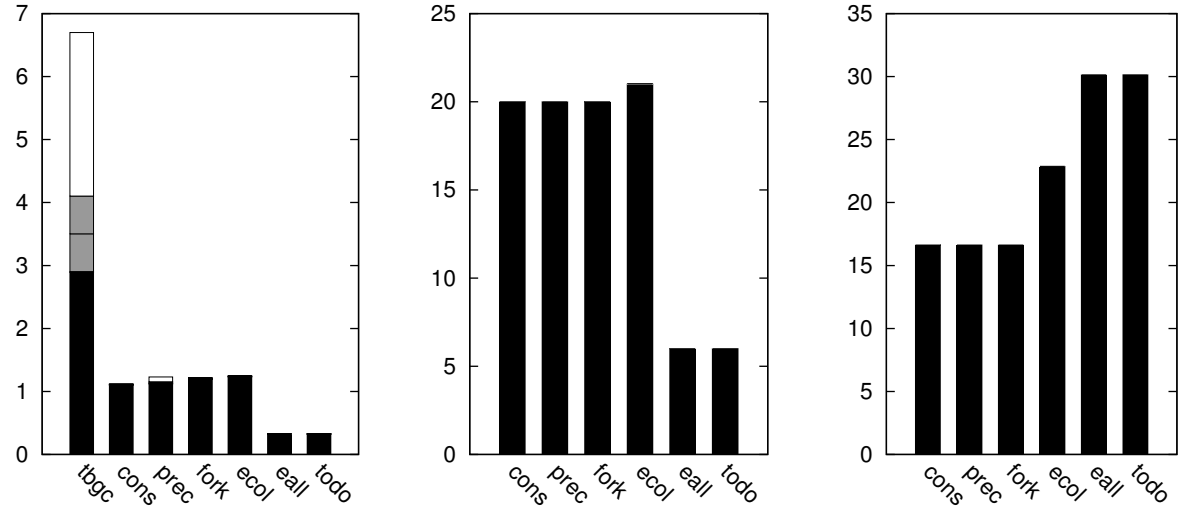


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

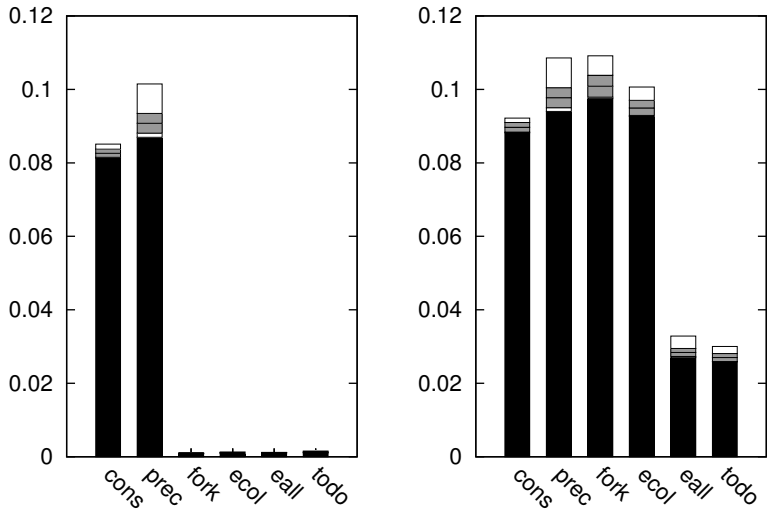


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.18: Resultados para `voronoi` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

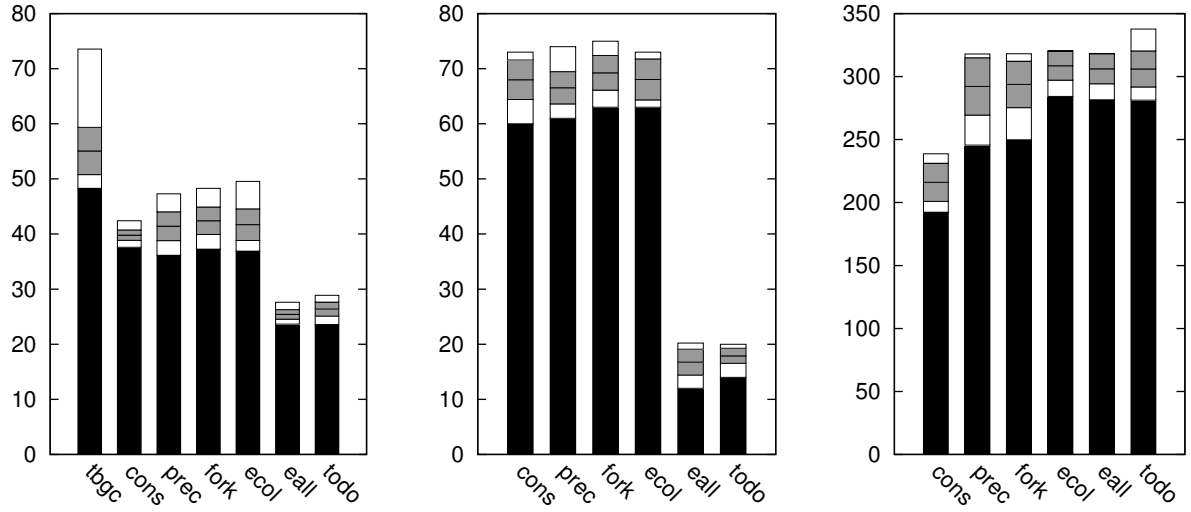


(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)

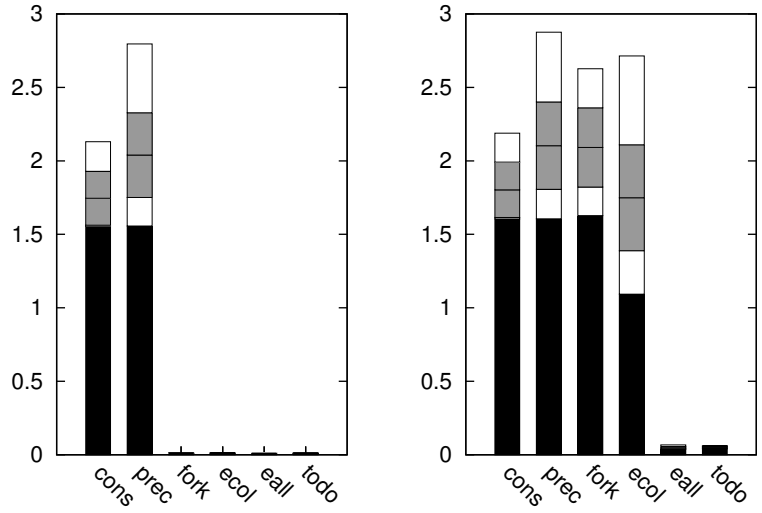


(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.19: Resultados para `voronoi` (utilizando 4 procesadores). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).



(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)



(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.20: Resultados para `dil` (utilizando 1 procesador). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

5.3.4 Resultados para pruebas reales

A continuación se presentan los resultados obtenidos para las pruebas reales (ver *Programas reales*). Recordamos que solo se pudo halla un programa que pueda ser utilizado a este fin, Dil, y que el objetivo principal de este trabajo se centra alrededor de obtener resultados positivos para este programa, por lo que a pesar de ser una única prueba, se le presta particular atención.

`dil`

En la figura 5.20 en la página anterior se presentan los resultados para `dil` al utilizar un procesador. Una vez más vemos una mejoría inmediata del tiempo total de ejecución al pasar de TBGC a CDGC, y una vez más se debe principalmente al mal factor de ocupación del *heap* de TBGC, dado que utilizando CDGC con la opción `min_free=0` se obtiene una media del orden de los 80 segundos, bastante más alta que el tiempo obtenido para TBGC.

Sin embargo se observa un pequeño incremento del tiempo de ejecución al introducir marcado preciso, y un incremento bastante más importante (de alrededor del 30 %) en el consumo máximo de memoria. Nuevamente, como pasa con la prueba `bh`, el efecto es probablemente producto del incremento en el espacio necesario para almacenar objetos debido a que el puntero a la información del tipo se guarda al final del bloque (ver *Marcado preciso*). En el cuadro 5.4 se puede observar la diferencia de memoria desperdiciada entre el modo conservativo y preciso.

El pequeño incremento en el tiempo total de ejecución podría estar dado por la mayor probabilidad de tener *falsos positivos* debido al incremento del tamaño del *heap*; se recuerda que el *stack* y memoria estática se siguen marcado de forma conservativa, incluso en modo preciso.

También se puede observar una gran disminución del tiempo total de ejecución al empezar a usar *eager allocation* (cerca de un 60 %, y más de un 200 % comparado con TBGC), acompañado como es usual de una baja en la cantidad de recolecciones realizadas (esta vez mayor, de más de 3 veces) y de una caída drástica del tiempo de pausa real (alrededor de 40 veces más pequeño); todo esto con un incremento marginal en el consumo total de memoria (aproximadamente un 5 %). En este caso el uso de *early collection* apenas ayuda a bajar el tiempo de pausa real en un 20 % en promedio aproximadamente. El tiempo de *stop-the-world* cae dramáticamente al empezar a realizar la fase de marcado de manera concurrente; es 200 veces más pequeño.

Al utilizar 4 procesadores (ver figura 5.21 en la página 136), hay algunos pequeños cambios. El tiempo total de ejecución es reducido todavía más (un 20 % que cuando se usa 1 procesador) cuando se utiliza *eager allocation*. Además al utilizar *early collection*, hay otra pequeña ganancia de alrededor del 10 %, tanto para el tiempo total de ejecución como para el tiempo de pausa real.

5.3.5 Aceptación

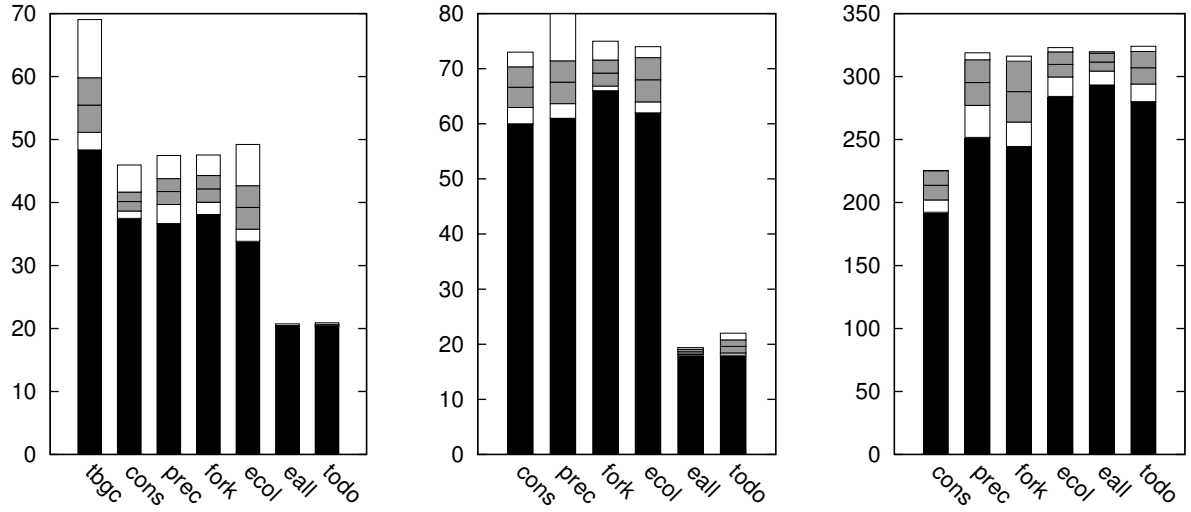
Los avances de este trabajo fueron comunicados regularmente a la comunidad de D a través de un blog [LMTDGC] y del grupo de noticias de D. Los comentarios hechos sobre el primero son en general positivos y denotan una buena recepción por parte de la comunidad a las modificaciones propuestas.

Memoria	Pedida (MiB)	Asignada (MiB)	Desperdicio (MiB)
Conservativo	307.48	399.94	92.46 (23 %)
Preciso	307.48	460.24	152.76 (33 %)

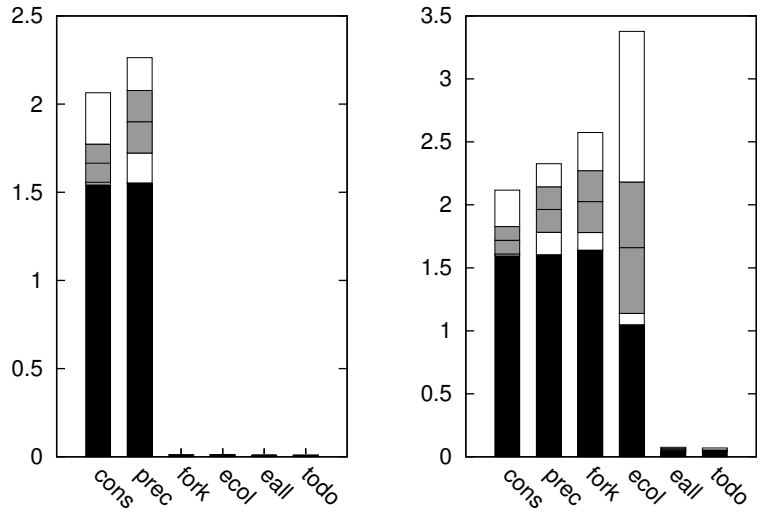
Cuadro 5.4: Memoria pedida y asignada para `dil` según modo de marcado conservativo o preciso (acumulativo durante toda la vida del programa).

Una vez agregado el marcado concurrente se hace un anuncio en el grupo de noticias que también muestra buenos comentarios y aceptación, en particular por parte de Sean Kelly, encargado de mantener el *runtime* de D 2.0, que comienza a trabajar en adaptar el recolector con idea de tal vez incluirlo de manera oficial en el futuro [NGA19235]. Poco después Sean Kelly publica una versión preliminar de la adaptación en la lista de correos que coordina el desarrollo del *runtime* de D 2.0 [DRT117].

También se ha mostrado interés de incluirlo en Tango, por lo que se han publicado los cambios necesarios en el sistema de seguimiento de mejoras y se encuentran actualmente en etapa de revisión [TT1997].



(a) Tiempo de ejecución (seg) (b) Cantidad de recolecciones (c) Uso máximo de memoria (MiB)



(d) *Stop-the-world* máximo (seg) (e) Pausa real máxima (seg)

Figura 5.21: Resultados para `dil` (utilizando 4 procesadores). Se presenta el mínimos (en negro), la media centrada entre dos desvíos estándar (en gris), y el máximo (en blanco) calculados sobre 50 corridas (para tiempo de ejecución) o 20 corridas (para el resto).

CAPÍTULO 6

Conclusión

Durante el desarrollo de este trabajo se introdujo al lenguaje de programación D y a los conceptos básicos de recolección de basura. Luego se analizó el recolector de basura actual y se señalaron sus principales falencias, proponiendo un conjunto de modificaciones con el objeto de subsanarlas. Para evaluar los resultados de las modificaciones se construyó un banco de pruebas variado para poder analizar tanto aspectos particulares como el funcionamiento de programas reales; y se establecieron métricas para cuantificar dichos resultados.

El objetivo principal fue bajar la latencia del recolector, es decir el tiempo máximo de pausa real, y se pudo comprobar que, salvo en casos muy particulares, esto fue conseguido de manera contundente (con tiempos de pausa hasta 200 veces menores que el recolector original de D). La inclusión del marcado concurrente demostró ser una forma eficaz de atacar el problema.

La aceptación de la solución por parte de la comunidad también ha sido un objetivo importante de este trabajo, y si bien en este sentido sigue siendo un trabajo en progreso, la recepción ha sido ampliamente positiva por parte de la comunidad y se espera que el resultado de este trabajo sea incorporado en el corto plazo tanto a D 1.0 a través de Tango, como a D 2.0.

Además de los objetivos principales se cumplieron otros objetivos anexos, pero no por eso menos importantes. Para la aplicación real el tiempo total de ejecución se ha reducido hasta casi una tercera parte, y para otras aplicaciones pequeñas se ha reducido más de 17 veces. Estos resultados han sido particularmente sorprendentes, siendo que la reducción del tiempo total de ejecución no ha sido parte del objetivo principal y no se habían encontrado referencias en la bibliografía de casos similares (por el contrario, en general la baja de la latencia suele estar acompañada de una suba en el tiempo total de ejecución).

Se ha podido experimentar además con el marcado preciso, otro de los problemas del recolector más presentes en la comunidad. Los resultados obtenidos son variados, encontrando casos donde se consigue una mejoría notoria y otros en donde la forma de almacenar la información de tipos produce resultados poco satisfactorios.

La mayor flexibilidad del recolector al ser configurable también ha demostrado ser útil. Por un lado para este mismo trabajo, al permitir realizar mediciones sobre el mismo binario utilizando diferentes configuraciones. Por otro, la amplia gama de resultados dispares obtenidos son una buena muestra de que no existen *balas de plata*, y cada programa tiene necesidades particulares en cuanto a recolección de basura. Por lo tanto, distintos programas pueden verse beneficiados o perjudicados por diferentes configuraciones. Esto hace que la posibilidad de configurar el recolector en tiempo de inicialización sea particularmente útil.

Finalmente, algunas optimizaciones muy pequeñas demostraron ser también muy valiosas para ciertos

casos particulares, logrando reducciones en el tiempo total de ejecución de hasta 5 veces.

6.1 Puntos pendientes, problemas y limitaciones

Si bien los objetivos de este trabajo han sido alcanzados con éxito, hay varias pequeñas mejoras que han quedado pendientes y algunos problemas y limitaciones conocidas. A continuación se describe cada una de ellos.

- Emisión de mensajes informativos para depuración.

  Entre las herramientas de depuración que provee el recolector, no se ha mencionado la posibilidad de emitir opcionalmente mensajes informativos para ayudar a depurar tanto problemas en el recolector como en el programa que lo usa. El recolector actual tiene esa posibilidad pero es configurable en tiempo de compilación. En este trabajo se agregaron las opciones en tiempo de inicialización `log_file` y `verbose` con el propósito de poder elegir un archivo en donde guardar los mensajes informativos y el nivel de detalle de dichos mensajes respectivamente, pero finalmente nunca se implementaron.

- Predicción para estimar cuando lanzar una recolección temprana.

  Las recolecciones se lanzan de manera temprana según la opción `min_free`. Una mejor aproximación podría ser predecir cuando se va a agotar la memoria libre de forma adaptativa, calculando la tasa de asignación de memoria y el tiempo total que tomó la recolección. Esta estimación se podría mejorar guardando un historial de que tan acertada fue para recolecciones pasadas. La predicción ideal debería ser capaz de:

  - Evitar tiempos de pausa (es decir, que la recolección temprana termine antes de que se agote la memoria libre).
  - No realizar recolecciones innecesarias (es decir, no lanzar recolecciones tempranas si el programa no está pidiendo memoria a una tasa suficientemente alta).

- Explosión del uso de memoria con creación ansiosa de *pools*.

  Se ha observado que en situaciones muy particulares, al usar creación ansiosa de *pools* (o *eager allocation*), el uso de memoria crece desmesuradamente. Si bien este efecto se ve principalmente en las pruebas sintetizadas con tal fin, algunos programas reales lo sufren también, pero en general se puede atenuar utilizando también *early collection*. Recordemos además, que lo analizado es el consumo **máximo** de memoria, por lo que una ráfaga de pedidos de memoria podría crear un pico, pero durante la mayor parte del transcurso del programa el consumo de memoria podría ser mucho menor. Queda pendiente analizar los casos puntuales con alguna métrica más detallada sobre el progreso del uso de memoria.

  También queda pendiente buscar alguna estimación de cuándo es conveniente utilizar *eager allocation* de forma adaptativa, dado que en general se ve que cuando explota el consumo de memoria, también explota el tiempo de pausa, lo que quita gran parte del sentido de usar *eager allocation* en primer lugar. Estimando de alguna manera cuanto va a crecer el tiempo de pausa debido a esta opción, se podría desactivar temporalmente cuando no haya ganancia en el tiempo de pausa para evitar esta explosión ante ráfagas de pedidos de memoria.

- Reestructuración y limpieza del código.

  Si bien se han hecho muchas mejoras a nivel de estructura y limpieza de código, ha quedado mucho pendiente. Todavía hay bastante repetición en el código y se mantiene la arquitectura básica del recolector.

- Experimentación con la llamada al sistema *clone(2)*.

  Linux implementa la llamada al sistema *fork(2)* a través de otra de más bajo nivel llamada *clone(2)*. *clone(2)* permite una granularidad a la hora de indicar que partes del proceso deben ser copiadas al hijo y cuales deben ser compartidas mucho mayor que *fork(2)*. Por ejemplo, se puede compartir toda la memoria del proceso, siendo este el mecanismo por el cual Linux implementa los hilos. Para este trabajo podría ser beneficioso usar *clone(2)* para evitar copiar otro tipo de estructuras dado que el proceso hijo, al correr solo la fase de marcado, nunca va a interferir el *mutator*. Se podría experimentar no copiando las siguientes estructuras, por ejemplo:

  CLONE_FILES
  : Tabla de descriptores de archivo.

  CLONE_FS
  : Tabla de sistemas de archivo montados.

  CLONE_IO
  : Contextos de entrada/salida.

  CLONE_SIGHAND
  : Tabla de manejadores de señales.

- Uso de memoria compartida.

  Al realizar marcado concurrente, si el *mutator* usa memoria compartida entre procesos que almacene punteros al *heap* podría haber problemas, dado que la fase de barrido no estaría trabajando con una *fotografía* de la memoria. El grafo de conectividad podría efectivamente cambiar mientras se corre la fase de barrido y por lo tanto el algoritmo deja de ser correcto, existiendo la posibilidad de que se reciclen celdas *vivas*.

  Dado que el usuario debe registrar cualquier puntero que no sea parte de la memoria estática, *stack* o *heap* del recolector como parte del *root set*, se podría agregar un parámetro extra a la función de registro que indique si los punteros agregados residen en memoria compartida. De este modo, al momento de hacer el *fork(2)*, el recolector debería realizar una copia de esos punteros mientras todos los hilos están pausados para obtener efectivamente una *fotografía* estable del *root set*.

- Condición de carrera al utilizar *fork(2)*.

  Existe una condición de carrera si se lanzan hilos usando directamente las llamadas al sistema operativo, es decir si no se lanzan a través del soporte de hilos de D, si el hilo lanzado utiliza archivos con *buffer* de C (`FILE*`). Esto se debe a la siguiente porción de código (introducida por el marcado concurrente):

```
function collect() is
   stop_the_world()
   fflush(null) //    <-------------------------
   child_pid = fork()
   if child_pid is 0
      mark_phase()
      exit(0)
   // proceso padre
   start_the_world()
   wait(child_pid)
   sweep()
```

  La llamada a *fflush(3)* es necesaria para evitar que los archivos con *buffer* escriban su contenido dos veces al dispositivo, ya que la llamada a *fork(2)* duplica el *buffer*, y si bien el archivo

Programa	TBGC	CDGC	CDGC-TBGC	CDGC/TBGC
bh	22208	27604	5396	1.243
bigarr	18820	24212	5392	1.287
bisort	19836	25232	5396	1.272
conalloc	25816	31208	5392	1.209
concpu	25816	31208	5392	1.209
dil	416900	422300	5400	1.013
em3d	20988	26380	5392	1.257
mcore	18564	23988	5424	1.292
rnddata	188940	194332	5392	1.029
sbtree	22196	27588	5392	1.243
split	24312	29736	5424	1.223
tree	18660	24084	5424	1.291
tsp	20772	26168	5396	1.260
voronoi	21184	26580	5396	1.255

Cuadro 6.1: Aumento del tamaño de la memoria estática (bytes).

no se usa en el proceso con la fase de marcado, la biblioteca estándar de C escribe todos los *buffers* pendientes al terminar el proceso. Esto funciona para los hilos registrados por D gracias a que `fflush(3)` se llama cuando todos los hilos están pausados, si no un hilo podría escribir al *buffer* justo después de llamar a `fflush(3)` pero antes de llamar a `fflush(2)`. Es por esto que si hay hilos no registrados por D que utilicen manejo de archivos con *buffer* de C, esta condición sí se puede dar y se pueden observar contenidos duplicados en dichos archivos.

Esta condición de carrera no tiene una solución simple, pero es de esperarse que no sea un problema real dado que no es un escenario común. Sin embargo eventualmente debería analizarse alguna solución más robusta.

- Soporte de referencias débiles.

  Tango 0.99.9 incluye soporte de referencias débiles. Si bien se incorporó el código para manejar las referencias débiles, se espera que no funcione correctamente con CDGC (no se ha podido comprobar por la falta de programas de prueba que lo utilicen). La razón es que el soporte de referencias débiles de Tango 0.99.9 se basa en la premisa de que la fase de marcado corre con todos los hilos pausados, sin embargo al utilizar marcado concurrente, esto no es más cierto. Parecen haber soluciones viables a este problema pero no se han analizado en profundidad aún.

- Pérdida de rendimiento con respecto al recolector original.

  Se ha observado también que, al no utilizar algunas optimizaciones de CDGC (como la mejora del factor de ocupación del *heap*), éste puede tener un rendimiento bastante menor a TBGC. Si bien no se ha investigado en profundidad las causas de esta pérdida de rendimiento, se han identificado algunos factores que podrían ser determinantes.

  Por un lado, se ha observado que la mayor parte del tiempo extra que utiliza CDGC proviene de la fase de marcado, en particular de los cambios introducidos por el marcado preciso. Si bien se puede desactivar el marcado preciso, la lógico en tiempo de ejecución no cambia, por lo que se paga el precio sin obtener los beneficios. Queda pendiente analizar en más detalle las causas de esto y posibles optimizaciones para subsanarlo.

  Además se ha observado un crecimiento importante en el tamaño del área de memoria estática del programa. En el cuadro 6.1 se puede observar dicho crecimiento para cada uno de los programas del banco de pruebas. Esto se debe a que el recolector original está escrito de una forma muy

Programa	TBGC	CDGC	CDGC-TBGC	CDGC/TBGC
bh	138060	159884	21824	1.158
bigarr	192004	213832	21828	1.114
bisort	115164	136988	21824	1.190
conalloc	149848	171676	21828	1.146
concpu	149848	171676	21828	1.146
dil	1859208	1881028	21820	1.012
em3d	116324	142248	25924	1.223
mcore	105748	127576	21828	1.206
rnddata	1492588	1518512	25924	1.017
sbtree	129860	155784	25924	1.200
split	144308	166136	21828	1.151
tree	105844	127672	21828	1.206
tsp	128412	150236	21824	1.170
voronoi	141112	162936	21824	1.155

Cuadro 6.2: Aumento del tamaño del binario (bytes).

primitiva, usando muy pocos tipos de datos definidos por el usuario, mientras que CDGC utiliza varias más, incluyendo algunos parametrizados. D guarda la información de tipos en el área de memoria estática y se genera mucha información por cada tipo. Además no separa el área de memoria estática que debe ser utilizada como parte del *root set* de la que no (no hay necesidad de que la información de tipos sea parte del *root set*). Esto causa que por cada recolección, se tenga que visitar bastante más memoria y, lo que es probablemente peor, que aumente la probabilidad de encontrar *falsos positivos*, dado que este área de memoria se marca siempre de forma conservativa.

Finalmente, en el cuadro 6.2 también se puede observar un incremento en el tamaño del binario, lo que puede ser otra causa de la pérdida de rendimiento, dado que puede afectar a la localidad de referencia del caché, por ejemplo.

6.2 Trabajos relacionados

Dado que D no ha penetrado en ámbitos académicos, se ha encontrado un solo trabajo de investigación relacionado. Sin embargo se ha encontrado otro que si bien no es formal, ha sido de mucha importancia para el desarrollo de esta tesis.

A continuación se describen ambos.

- *Memory Management in the D Programming Language* [PAN09].

  Tesis de licenciatura de Vladimir Panteleev cuya resumen traducido es el siguiente:

  > Este reporte describe el estudio de las técnicas de manejo automático de memoria, su implementación en el lenguaje de programación D, y el trabajo para mejorar el estado del manejo de memoria.

  Si bien plantea pequeñas optimizaciones para el recolector de basura (algunas utilizadas en este trabajo), se centra principalmente en el desarrollo de Diamond, una utilidad para depuración de manejo de memoria en D.

- Integración de marcado preciso del *heap* al recolector de basura [DBZ3463].

  Ya citado varias veces en este trabajo; fue comenzado por David Simcha y publicado en el sistema de seguimiento de fallas de D que se limita a una implementación a nivel biblioteca de usuario y

sobre D 2.0. Vincent Lang (mejor conocido como *wm4* en la comunidad de D) da continuidad a este trabajo pero modificando el compilador DMD y trabajando con D 1.0 y Tango.

El soporte de marcado preciso presentado en este trabajo se basa en las modificaciones hechas al compilador DMD por Vincent Lang (que aún no fueron integradas de forma oficial).

6.3 Trabajos futuros

En la sección *Puntos pendientes, problemas y limitaciones* se mencionan varios aspectos de este trabajo que podrían verse beneficiados por trabajos futuros, sin embargo se trata en general de pequeñas optimizaciones o mejoras de alcance muy limitado.

A continuación se recopilan varios otros aspectos identificados durante el desarrollo del presente trabajo, pero que requieren un nivel de análisis y, potencialmente, de desarrollo mayor a los ya presentados en la sección mencionada.

- Mejoras en la organización de memoria del recolector.

  Si bien se ha mencionado en un principio la organización actual como un aspecto positivo del recolector, varios resultados han demostrado deficiencias importantes. El nivel de espacio desperdiciado por la división de memoria en bloques puede ser muy significativa y la forma en la que se almacena la información de tipos para el marcado preciso puede incluso acentuarlo todavía más (como se demuestra en los resultados para `bh` y `dil`).

  Este problema no solo afecta al consumo de memoria, además genera un efecto dominó por el incremento de la probabilidad de tener *falsos positivos* y perjudica al tiempo total de ejecución por empeorar la localidad de referencia del caché y por hacer que se prolongue la recolección de basura por tener que marcar y barrer más memoria.

  Una posible alternativa es tener una lista de libres por **tipo**, cuyo tamaño de bloque sea exactamente igual al tamaño del tipo que almacena. La información de tipo se almacenaría entonces solo una vez y no habría desperdicio de memoria alguno dejando de lado un posible relleno para completar una página. Este esquema debería tener algún tipo de guarda para programas con una cantidad exuberante de tipos de datos.

  También podría ser conveniente separar los bloques marcados como `NO_SCAN` de los que sí deben ser marcados, de manera que no necesite almacenar directamente los bits de `mark` , `scan` y `noscan`. También se podría proponer algún área de memoria especial para almacenar cadenas de texto (como un caso especial de lo anterior) por tener estas características muy particular (largos muy variables, cambian de tamaño de forma relativamente frecuente, etc.). Las posibilidades son enormes.

- Mejoras en la fase de barrido.

  En este trabajo todas las mejoras propuestas se encargaron de la fase de marcado, pero mucho se pude mejorar en la fase de barrido también. Por un lado se podría agregar barrido perezoso para disminuir aún más el tiempo de pausa real. Se ha mostrado que en muchos casos los tiempos de pausa pueden ser considerablemente altos debido a que la fase de barrido no se realiza en paralelo como el marcado.

  Otra forma de disminuir el tiempo de pausa real sería realizar un barrido concurrente también. Esto no puede realizarse en otro proceso porque el barrido es el encargado de ejecutar los *finalizadores*, pero sí se podría barrer en otro hilo y, por ejemplo, seguir utilizando *eager allocation* hasta que el barrido finalice.

- Mejoras en la precisión del marcado.

  Como se mencionó anteriormente, el área de memoria estática se marca de forma conservativa dada la falta de información de tipos de ésta. Sin embargo es bastante razonable pensar en que el compilador genere información de tipos para el área de memoria estática o que al menos informe mejor al recolector que partes deben ser consideradas parte del *root set* y cuales no. Dado que la memoria estática crece de forma considerable con el incremento de la cantidad de tipos definidos por el usuario, ya solo esa división puede hacer una diferencia importante; en especial considerando como aumenta la memoria estática solamente por usar más tipos de datos en el recolector.

  También podría explorarse el agregado de precisión al *stack* pero esto es realmente muy complicado dado que la única solución que pareciera viable es el uso de *shadow stack* [HEND02] que requiere un trabajo extra por cada llamado a función, cosa que va en contra de la filosofía de D de pagar solo por lo que se usa. Sin embargo podría explorarse agregar un esquema de ese tipo como una opción del compilador, de forma que el usuario pueda decidir si vale la pena para una aplicación particular o no.

- Mejoras en la concurrencia.

  El *lock* global del recolector es otro aspecto que demostró ser problemático. Podrían analizarse formas de minimizar la necesidad de usar *locks* o de hacerlo de forma más granular, de manera que algunas operaciones del recolector puedan ser ejecutadas en paralelo. También se podría experimentar con el uso de estructura de datos libres de *locks* (*lock-free*).

  Otra forma de minimizar la sincronización es utilizando *pools* por hilo, de manera de poder alocar memoria de forma concurrente y hasta explorar la posibilidad de efectuar recolecciones locales a un solo hilo; aunque esto último probablemente sea equivalente a implementar un recolector de basura con particiones (por ejemplo generacional).

- Recolección con movimiento.

  La información de tipos provista por el trabajo hecho por Vincent Lang [DBZ3463] es suficientemente completa como para poder implementar un recolector con movimiento. La efectividad de un recolector de estas características en D está por comprobarse, dado que cualquier celda apuntada por alguna palabra que debió ser marcada de forma conservativa debe quedar inmóvil, por lo que gran parte del éxito de un recolector con movimiento en D está supeditado a la proporción de celdas que queden inmóviles. Sin embargo sea muy probablemente un área que valga la pena explorar.

Glosario

ABI
: Abreviatura en inglés de *Application Binary Interface*, es la interfaz de bajo nivel entre un programa y el sistema operativo u otro programa.

abstracción bicolor
: Método para marcar todas las celdas de un grafo que sea accesibles de forma transitiva a partir de una o más raíces que consiste en *pintar* todas las celdas de blanco inicialmente y luego, a medida que son visitadas, pintarlas de negro. Al finalizar el proceso las celdas accesibles están pintadas de negro y el resto de blanco. Ver *Recorrido del grafo de conectividad*.

abstracción tricolor
: Método para marcar todas las celdas de un grafo que sea accesibles de forma transitiva a partir de una o más raíces que consiste en *pintar* todas las celdas de blanco inicialmente y luego, a medida que son visitadas, pintarlas de gris y finalmente, cuando todas sus *hijas* son visitadas también, de negro. Al finalizar el proceso las celdas accesibles están pintadas de negro y el resto de blanco. Ver *Abstracción tricolor*.

activation record
: Ver *stack frame*.

address space
: Conjunto de posibles direcciones de memoria asignada a un programa. Puede ser un conjunto no contiguo o espaciado.

arreglo
: Disposición de celdas de igual tamaño de forma consecutiva en la memoria de manera que puedan ser fácilmente indizadas.

back-end
: Parte del compilador encargada de convertir la representación intermedia generada por el *front-end* a código de máquina.

basura
: Dependiendo del contexto, se refiere a una celda *muerta*, un conjunto de celdas *muertas* o al conjunto completo de celdas *muertas*.

benchmark
Banco de pruebas utilizado para medir y comparar el rendimiento de un programa, algoritmo o proceso en general.

best-fit
Búsqueda para encontrar la región de memoria contigua libre que mejor se ajuste al tamaño de un objeto (es decir, la región más pequeña lo suficientemente grande como para almacenarlo).

bitset
Ver conjunto de bits.

BNF
Notación de Backus-Naur, una meta-sintaxis usada para expresar gramáticas libres de contexto.

cache
Memoria pequeña (por ser típicamente muy costosa) pero muy veloz.

celda
Porción contigua de memoria destinada a almacenar un objeto o estructura de dato particular.

celda *blanca*
En la abstracción bicolor y tricolor, son celdas que no fueron aún visitadas por la fase de marcado.

celda *gris*
En la abstracción tricolor, son celdas que ya fueron visitadas por la fase de marcado pero deben ser visitadas nuevamente (porque sus *hijas* no fueron visitadas por completo todavía o porque hubo algún cambio en la celda).

celda *hija*
Celda para la cual existe una referencia desde la celda actual. Se dice que H es *hija* de P si P contiene una referencia a H.

celda *jóven*
Celda que no lleva ninguna (o muy pocas) recolecciones sin ser recolectada.

celda *muerta*
Celda de memoria que no puede ser accedida transitivamente a través del *root set*.

celda *negra*
En la abstracción bicolor y tricolor, son celdas que ya fueron visitadas por completo (es decir, incluyendo sus celdas *hijas*) por la fase de marcado.

celda *vieja*
Celda que lleva varias recolecciones sin ser recolectada.

celda *viva*
Celda de memoria que puede ser accedida transitivamente a través del *root set*.

ciclo
Un conjunto de celdas que están referenciadas entre sí de forma tal que siempre se puede llegar de una celda a sí misma a través de las referencias.

ciudadano de primera clase
Tipo soportado por completo por el lenguaje (por ejemplo disponen de expresiones literales anónimas, pueden ser almacenados en variables y estructuras de datos, tienen una identidad intrínseca, etc.).

conjunto de bits
Estructura de datos que sirve para almacenar un conjunto de indicadores de forma eficiente. Ge-

neralmente se implementa utilizando una porción de memoria donde cada bit es un indicador; si el bit está en 0 el indicador no está presente y si está en 1, el indicador está activado. La manipulación de los bits (individuales y en conjunto) en general se realiza de forma eficiente utilizando máscaras.

conteo de referencias
Uno de los tres principales algoritmos clásicos de recolección de basura. Ver *Conteo de referencias*.

context-free grammar
Gramática que no depende del contexto (es decir, de información semántica).

conversión *covariante*
Conversión de tipos que preserva el orden de los tipos de más específicos a más genéricos.

copia de semi-espacio
Uno de los tres principales algoritmos clásicos de recolección de basura. Ver *Copia de semi-espacio*.

copying collector
Nombre alternativo para el algoritmo *copia de semi-espacios*, aunque puede referirse también a una familia más general de algoritmos con movimiento de celdas. Ver *Copia de semi-espacio* y *Movimiento de celdas*.

COW
Del inglés *Copy On Write* (*copiar al escribir* en castellano) es una técnica muy utilizada para disminuir las copias innecesarias, evitando hacer la copia hasta que haya una modificación. Mientras no hayan modificaciones no es necesario realizar una copia porque todos los interesados verán la misma información. Esta técnica es utilizada por el *kernel* de Linux por ejemplo, para que varias instancias de un mismo proceso puedan compartir la memoria.

CSV
Formato simple para almacenar datos en forma de tabla, separados por comas (de ahí el nombre, en inglés *Comma separated values*) en un archivo de texto. Cada línea del archivo es interpretada como una fila de datos relacionados, y cada valor de separado por comas como una columna.

CTFE
Abreviatura en inglés de *Compile-Time Function Execution*, es la capacidad de un lenguaje de programación de ejecutar una función en tiempo de compilación en vez de tiempo de ejecución.

dangling pointer
Puntero que almacena una dirección de memoria inválida (*puntero colgante* en castellano).

DbC
Ver diseño por contrato (del inglés *Design by Contract*).

delegado
Es una estructura simple que modela una función acompañada de un contexto. En general se utiliza para representar un puntero a una función miembro de un objeto en particular o a una función anidada (donde el contexto es el *stack frame* de la función que la contiene).

desbordamiento de pila
Agotamiento de la memoria del *stack*.

determinístico
Algoritmo o proceso que se comporta de forma predecible (dada una cierta entrada siempre produce el mismo resultado y los pasos realizados son exactamente los mismo, pasando por la misma secuencia de estados).

dirección
Una dirección de memoria es la especificación de su ubicación en memoria. Típicamente se representan como enteros sin signo y ocupan una palabra.

diseño por contrato
Técnica de diseño de software que consiste en especificar formalmente, de forma precisa y verificable, la interfaz entre componentes de software.

excepción
Construcción de un lenguaje de programación para manejar la presencia de situaciones anormales (en general errores) cambiando el flujo de ejecución del programa.

exception-safe
Propiedad de un programa que ante un error en tiempo de ejecución manifestado como una *excepción* no provoca efectos indeseados (como pérdida de memoria, corrupción de datos o salida inválida).

falso positivo
Palabra que es tratada como un potencial puntero cuyo valor almacenado coincide con una dirección válida dentro del *heap* pero que en realidad no es un puntero.

fase de barrido
Segunda fase del algoritmo *marcado y barrido*. Ver *Marcado y barrido*.

fase de marcado
Primera fase del algoritmo *marcado y barrido* (entre otros). Ver *Recorrido del grafo de conectividad* y *Marcado y barrido*.

finalización
Referente a la acción de llamar a una función miembro de un objeto, generalmente llamada destructor, cuando éste deja de ser utilizado.

first-fit
Búsqueda para encontrar la primera región de memoria contigua libre donde quepa un objeto (es decir, la primera región lo suficientemente grande como para almacenar el objeto a asignar).

forwarding address
Dirección de memoria de re-dirección utilizada para localizar la nueva ubicación de una celda en algoritmos de recolección con movimiento. Ver *Copia de semi-espacio*.

fragmentación
Incapacidad de usar memoria debido a la disposición de memoria actualmente en uso, que deja la memoria libre dividida en bloques demasiado pequeños.

fromspace
Uno de los dos semi-espacios del algoritmo *copia de semi-espacios*. Ver *Copia de semi-espacio*.

front-end
Parte del compilador encargada de hacer el análisis léxico, sintáctico y semántico del código fuente, generando una representación intermedia que luego el *back-end* convierte a código de máquina.

función *pura*
Función que no tiene efectos secundarios. Una función pura ejecutada con los mismo parámetros siempre devuelve el mismo resultado.

grafo de conectividad
Grafo conformado por la memoria del *heap*. Los vértices son las celdas de memoria y las aristas

las referencias (o punteros) que tiene una celda apuntando a otras. Ver *Conceptos básicos*.

heap
Área de memoria que en la cual se asigna y liberan celdas dinámicamente (durante la ejecución del programa).

hit rate
Frecuencia con la que el caché puede responder con éxito.

lista de libres
Forma de organizar el *heap* en la cual se asigna una nueva celda obteniéndola de una lista de celdas libres. Ver *Lista de libres / pointer bump allocation*.

live set
Conjunto de todas las celdas *vivas*.

localidad de referencia
Medida en que los accesos sucesivos de memoria cercana espacialmente son cercanos también en el tiempo. Por ejemplo, un programa que lee todos los elementos de una matriz contigua de una vez o que utiliza la misma variable repetidamente tiene buena localidad referencia.

lock
También conocido como *mutex* (abreviación de *exclusión mutua* en inglés), es un objeto de sincronización que permite serializar la ejecución de múltiples hilos.

low level allocator
Administrador de memoria de bajo nivel que obtiene la memoria del sistema operativo y la provee al recolector (o al *mutator* directamente).

marcado y barrido
Uno de los tres principales algoritmos clásicos de recolección de basura. Ver *Marcado y barrido*.

memoria estática
Memoria fija destinada a un programa. Es fija en el sentido en que no varía su tamaño ni puede asignarse o liberarse durante la ejecución del programa.

mixin
En D se refiere a un fragmento de código (M) que puede incluirse dentro de otro (O) como si M hubiera sido escrito directamente dentro de O. En general se utiliza para suplantar la herencia múltiple pero tiene muchos otros usos.

multi-core
Arquitectura que combina dos o más núcleos (*cores*) independientes que trabajan a la misma frecuencia, pero dentro de un solo circuito integrado o procesador.

mutator
Parte del programa que realiza cambios al grafo de conectividad.

overhead
Cualquier combinación de exceso directo o indirecto de tiempo de computación, memoria, ancho de banda u otro recurso que sea requerido para cumplir un objetivo particular.

palabra
Tamaño de dato característico de un procesador que permite almacenar una dirección de memoria. Generalmente este tamaño coincide con el tamaño de dato que el procesador puede manipular de forma más eficiente.

parsing
Análisis sintáctico de un lenguaje estructurado.

pattern matching
: Acto de verificar la presencia de un constituyente sintáctico de un patrón dado.

pinning
: Técnica que consiste en marcar una celda como inmóvil. Generalmente se utiliza en recolectores semi-conservativos con movimiento para no mover celdas que son alcanzadas desde palabras para las que no se tiene información de tipos.

pointer bump allocation
: Forma de organizar el *heap* en la cual se asigna una nueva celda incrementando un puntero. Ver *Lista de libres / pointer bump allocation*.

POSIX
: Familia de estándares de llamadas al sistema operativo definidos por la IEEE y especificados formalmente en IEEE 1003. El nombre proviene del acrónimo de *Portable Operating System Interface*, agregando la X final como representación de *UNIX*, como seña de identidad de la API.

puntero *interior*
: Puntero que en vez de apuntar al inicio de una celda, apuntan a una dirección arbitraria dentro de ella.

página
: Unidad mínima de memoria que asigna el sistema operativo a un programa (típicamente el tamaño de página es de 4096 bytes).

RAII
: Técnica que consiste en reservar recursos por medio de la construcción de un objeto y liberarlos cuando éste se libera (del inglés *Resourse Adquisition Is Initialization*).

recolección con movimiento de celdas
: Recolección en la cual una celda de memoria puede ser movida a otra ubicación en el *heap*. Ver *Movimiento de celdas*.

recolección concurrente
: Recolección que puede correr en paralelo con el *mutator*. Ver *Recolección concurrente / paralela / stop-the-world*.

recolección conservativa
: Recolección que no tiene información de tipos y trata cada palabra del *root set* o *heap* como un posible puntero. Ver *Recolectores conservativos versus precisos*.

recolección directa
: Recolección en la cual el compilador o lenguaje instrumenta al *mutator* de forma tal que la información sobre el grafo de conectividad se mantenga activamente cada vez que hay un cambio en él. Ver *Recolección directa / indirecta*.

recolección generacional
: Caso particular de *recolección por particiones* en el cual las particiones se realizan utilizando la cantidad de recolecciones que *sobrevive* una celda. Ver *Recolección por particiones / generacional*.

recolección incremental
: Recolección que se realiza de forma intercalada con el *mutator*. Ver *Recolección incremental*.

recolección indirecta
: Recolección que, generalmente, no interfiere con el *mutator* en cada actualización del grafo de conectividad. Ver *Recolección directa / indirecta*.

recolección paralela
: Recolección que puede correr en paralelo en varios hilos. Ver *Recolección concurrente / paralela / stop-the-world*.

recolección por particiones
: Recolección en la que se divide el *heap* en particiones con el objetivo de recolectar la partición con mayor concentración de *basura*. Ver *Recolección por particiones / generacional*.

recolección precisa
: Recolección que tiene información de tipos completa y puede determinar exactamente que palabras son punteros y cuales no. Ver *Recolectores conservativos versus precisos*.

recolección semi-precisa
: Recolección que tiene información de tipos parcial y puede determinar para algunas palabras si son punteros o no, y para otras las trata como punteros potenciales. Ver *Recolectores conservativos versus precisos*.

recolección *stop-the-world*
: Recolección que detiene todos los hilos del *mutator*. Ver *Recolección concurrente / paralela / stop-the-world*.

recolector
: Parte del programa que recupera celdas *muertas* (no realiza cambios en el grafo de conectividad).

recolector *híbrido*
: Recolector que emplea distintas técnicas de recolección dependiendo de distintas características de las celdas (por ejemplo cuantas recolecciones lleva sin ser recolectado, el tamaño de celda, etc.).

referencia débil
: Referencia que no es tomada en cuenta en el grafo de conectividad (es decir, si un objeto es solamente alcanzable a través de una referencia débil, puede ser reciclado por el recolector).

registro
: Memoria muy veloz del procesador que por lo general tiene el tamaño de una palabra. En general son muy escasos y es donde los procesadores hacen realmente los cálculos.

root set
: Conjunto de celdas de memoria que sirven como punto de partida para recorrer el grafo de conectividad. En general se compone de memoria estática, registros y el *stack*.

RTTI
: Abreviatura del inglés *Run-Time Type Identification*, es la información de tipos disponible en tiempo de ejecución.

runtime
: Biblioteca de un lenguaje que provee los servicios básicos (como creación de objetos, manejo de hilos u otras construcciones que ofrezca el lenguaje).

semi-space
: Nombre alternativo para el algoritmo *copia de semi-espacios*. Ver *Copia de semi-espacio*.

semántica de referencia
: Propiedad de los tipos que son tratados como si fueran un puntero. Nunca se hacen copias del objeto, siempre se pasa por referencia implícitamente.

semántica de valor
: Propiedad de los tipos son tratado como si fuera un valor concreto. En general se pasa por valor y

se hacen copias a menos que se utilice explícitamente un puntero.

sistema de tipos
Forma en que un lenguaje de programación clasifica valores y expresiones en tipos, como los manipula y como interactúan éstos entre sí.

slicing
Problema que surge cuando los objetos polimórficos tienen semántica de valor, consiste en pasar una clase derivada a una función que acepta una clase base por valor como parámetro. Al realizarse la copia, se utiliza solo la parte de la clase base y se pierden (o *rebanan*) los atributos de la clase derivada, y la información de tipos en tiempo de ejecución (*RTTI*).

stack
Área de memoria organizada en forma de pila donde se almacenan típicamente las variables locales, parámetros, valor de retorno y dirección de retorno de las subrutinas.

stack frame
Estructura de datos dependiente de la arquitectura que contiene información del estado de una función, incluyendo, por ejemplo, sus variables locales, parámetros y dirección de retorno.

stack overflow
Ver *desbordamiento de pila*.

stop-the-world
Ver *recolección stop-the-world*.

string
Secuencia lineal de caracteres utilizada normalmente en los lenguajes de programación para representar texto (aunque pueden ser utilizados para representar una secuencia lineal de bytes de cualquier tipo también).

system programming
Se refiere a programación de bajo nivel. En general involucra manipulación de punteros, acceso directo al lenguaje de máquina y por consiguiente al *hardware*.

templates
Técnica para construir algoritmos genéricos incluyendo parámetros como tipos o valores.

thread-safe
Propiedad de una función o fragmento de código que permite que corra concurrentemente en dos o más hilos de ejecución paralelos sin provocar efectos indeseados (como pérdida de memoria, corrupción de datos o salida inválida).

***tipado* dinámico**
Verificación de tipos en tiempo de ejecución.

***tipado* estático**
Verificación de tipos en tiempo de compilación.

tospace
Uno de los dos semi-espacios del algoritmo *copia de semi-espacios*. Ver *Copia de semi-espacio*.

two level allocators
Administrador de memoria que utiliza dos niveles para organizar las celdas de memoria; obtiene del sistema operativo páginas completas y éstas a su vez de dividen en bloques que son utilizados para almacenar las celdas.

two-space
Nombre alternativo para el algoritmo *copia de semi-espacios*. Ver *Copia de semi-espacio*.

type-safe
: Operación que no compromete ni subvierte la verificación de tipos.

verificación de tipos
: Forma en la que un sistema de tipos asigna tipos y verifica sus interacciones.

weak reference
: Ver referencia débil.

working set
: Conjunto de celdas con la que trabaja el programa de forma intensiva durante un período considerable de tiempo.

Bibliografía

[ALX10] Andrei Alexandrescu. The D Programming Language. Addison-WesleyProfessional, 2010. ISBN 978-0321635365.

[BEZO06] Emery D. Berger, Benjamin G. Zorn. DieHard: probabilisticmemory safety for unsafe languages. PLDI '06: Proceedings of the 2006ACM SIGPLAN conference on Programming language design andimplementation. 2006. ISBN 1-59593-320-4.

[BH86] J. Barnes and P. Hut. A hierarchical o(N log N) force-calculationalgorithm. Nature Volumen 324, páginas 446-449. Diciembre 1986.

[BKIP08] Kris Macleod Bell, Lars Ivar Igesund, Sean Kelly, and MichaelParker. Learn to Tango with D. Apress, 2007. ISBN 1-59059-960-8.

[BLA06] Blackburn et ál. The DaCapo benchmarks: java benchmarkingdevelopment and analysis. OOPSLA '06: Proceedings of the 21st annual ACMSIGPLAN conference on Object-oriented programming systems, languages, andapplications. ISBN 1-59593-348-4. Páginas 169-190. 2006.

[BLAC08] Stephen Blackburn and Kathryn McKinley. Immix Garbage Collection:Mutator Locality, Fast Collection, and Space Efficiency [1]. Proceedings ofthe ACM SIGPLAN '08 Conference on Programming Language Design andImplementation. Páginas 22-32. Junio 2008.

[BN98] G. Bilardi and A. Nicolau. Adaptive Bitonic Sorting: An optimalparallel algorithm for shared-memory machines. SIAM J. Comput.Volumen 18, número 2, páginas 216-228. 1998.

[BOEH88] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in anUncooperative Environment [2]. Software Practice and Experience Volumen 18,Número 9. Páginas 807-820. Septiembre 1988.

[BOEH91] Hans-Juergen Boehm and Alan J. Demers and Scott Shenker. MostlyParallel Garbage Collection [3]. Proceedings of the ACM SIGPLAN '91Conference on Programming Language Design and Implementation, Volumen 26,Número 6. Páginas 157-164. Junio 1991.

[1] http://cs.anu.edu.au/techreports/2007/TR-CS-07-04.pdf
[2] http://www.hpl.hp.com/personal/Hans_Boehm/spe_gc_paper/preprint.pdf
[3] http://www.hpl.hp.com/personal/Hans_Boehm/gc/papers/pldi91.ps.Z

[BOEH93] Hans-Juergen Boehm. Space Efficient Conservative GarbageCollection [4]. Proceedingsof the ACM SIGPLAN '93 Conference on Programming Language Design andImplementation, Volumen 28, Número 6. Páginas 197-206. Junio 1993.

[BOEHWD] Hans-J. Boehm. Conservative GC Algorithmic Overview [5]. HPLabs / SGI. Obtenido en Junio de 2009.

[CAR95] A. Rogers and M. Carlisle and J. Reppy and L. Hendren. SupportingDynamic Data Structures on Distributed Memory Machines. Transactions onProgramming Languages and Systems, volúmen 17, número 2. Marzo 1995.

[CDG93] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta,T. von Eicken and K. Yelick. Parallel Programming in Split-C.Supercomputing 1993, páginas 262-273. 1993.

[CMK01] B. Cahoon and K. S. McKinley. Data Flow Analysis for SoftwarePrefetching Linked Data Structures in Java [6]. International Conference onParallel Architectures and Compilation Techniques (PACT). Barcelona,España. Septiembre 2001.

[DBZ3463] David Simcha, et ál. Integrate Precise Heap Scanning Into theGC [7]. The D Programming Language Issue Tracking System. Issue 3463.Noviembre 2009.

[DRT117] Sean Kelly. D Concurrent GC [8]. Lista de correo de Druntime. 15de septiembre de 2010.

[DWAA] Walter Bright. D Programming Language 1.0 Reference, Associativearrays [9]. Digital Mars. Octubre 2010.

[DWAB] Walter Bright. D Programming Language 1.0 Reference, D ApplicationBinary Interface [10]. Digital Mars. Octubre 2010.

[DWAL] Walter Bright. D Programming Language 1.0 Reference, Alignattribute [11]. Digital Mars. Octubre 2010.

[DWAR] Walter Bright. D Programming Language 1.0 Reference, Arrays [12].Digital Mars. Octubre 2010.

[DWCC] Walter Bright. D Programming Language 1.0 Reference, Interfacing toC [13]. Digital Mars. Octubre 2010.

[DWCF] Walter Bright. D Programming Language 1.0 Reference, Compile TimeFunction Execution (CTFE) [14]. Digital Mars. Octubre 2010.

[DWCL] Walter Bright. D Programming Language 1.0 Reference, Classes [15].Digital Mars. Octubre 2010.

[DWCP] Walter Bright. D Programming Language 1.0 Reference, Contracts [16].Digital Mars. Octubre 2010.

[4] http://www.hpl.hp.com/personal/Hans_Boehm/gc/papers/pldi93.ps.Z
[5] http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcdescr.html
[6] ftp://ftp.cs.umass.edu/pub/osl/papers/pact01-prefetch.ps.gz
[7] http://d.puremagic.com/issues/show_bug.cgi?id=3463
[8] http://lists.puremagic.com/pipermail/d-runtime/2010-September/000117.html
[9] http://www.digitalmars.com/d/1.0/hash-map.html
[10] http://www.digitalmars.com/d/1.0/abi.html
[11] http://www.digitalmars.com/d/1.0/attribute.html#align
[12] http://www.digitalmars.com/d/1.0/arrays.html
[13] http://www.digitalmars.com/d/1.0/interfaceToC.html
[14] http://www.digitalmars.com/d/1.0/function.html#interpretation
[15] http://www.digitalmars.com/d/1.0/class.html
[16] http://www.digitalmars.com/d/1.0/dbc.html

[DWDC] Walter Bright. D Programming Language 1.0 Reference,Declarations [17]. Digital Mars. Octubre 2010.

[DWDE] Walter Bright. D Programming Language 1.0 Reference, Classdestructors [18]. Digital Mars. Octubre 2010.

[DWDO] Walter Bright. D Programming Language 1.0 Reference, Embeddeddocumentation [19]. Digital Mars. Octubre 2010.

[DWEB] Walter Bright. D Programming Language Website [20]. Digital Mars.Octubre 2010.

[DWES] Walter Bright. D Programming Language 1.0 Reference, Exception safepogramming [21]. Digital Mars. Octubre 2010.

[DWEX] Walter Bright. D Programming Language 1.0 Reference, Trystatement [22]. Digital Mars. Octubre 2010.

[DWFE] Walter Bright. D Programming Language 1.0 Reference, Foreachstatement [23]. Digital Mars. Octubre 2010.

[DWFU] Walter Bright. D Programming Language 1.0 Reference, Functions [24].Digital Mars. Octubre 2010.

[DWGC] Walter Bright. D Programming Language 1.0 Reference, Garbagecollection [25]. Digital Mars. Octubre 2010.

[DWGT] Walter Bright. D Programming Language 1.0 Reference, Gotostatement [26]. Digital Mars. Octubre 2010.

[DWIA] Walter Bright. D Programming Language 1.0 Reference, D x86 InlineAssembler [27]. Digital Mars. Octubre 2010.

[DWIE] Walter Bright. D Programming Language 1.0 Reference, Isexpressions [28]. Digital Mars. Octubre 2010.

[DWIF] Walter Bright. D Programming Language 1.0 Reference, Interfaces [29].Digital Mars. Octubre 2010.

[DWIN] Walter Bright. D Programming Language 1.0 Reference, Autodeclaration [30]. Digital Mars. Octubre 2010.

[DWLR] Walter Bright. D Programming Language 1.0 Reference [31]. DigitalMars. Octubre 2010.

[DWME] Walter Bright. D Programming Language 1.0 Reference, Mixinexpressions [32]. Digital Mars. Octubre 2010.

[17] http://www.digitalmars.com/d/1.0/declaration.html
[18] http://www.digitalmars.com/d/1.0/class.html#Destructor
[19] http://www.digitalmars.com/d/1.0/ddoc.html
[20] http://www.digitalmars.com/d/
[21] http://www.digitalmars.com/d/1.0/exception-safe.html
[22] http://www.digitalmars.com/d/1.0/statement.html#TryStatement
[23] http://www.digitalmars.com/d/1.0/statement.html#ForeachStatement
[24] http://www.digitalmars.com/d/1.0/function.html
[25] http://www.digitalmars.com/d/1.0/garbage.html
[26] http://www.digitalmars.com/d/1.0/statement.html#GotoStatement
[27] http://www.digitalmars.com/d/1.0/iasm.html
[28] http://www.digitalmars.com/d/1.0/expression.html#IsExpression
[29] http://www.digitalmars.com/d/1.0/interface.html
[30] http://www.digitalmars.com/d/1.0/declaration.html#AutoDeclaration
[31] http://www.digitalmars.com/d/1.0/lex.html
[32] http://www.digitalmars.com/d/1.0/expression.html#MixinExpression

[DWMM] Walter Bright. D Programming Language 1.0 Reference, Memorymanagement [33]. Digital Mars. Octubre 2010.

[DWMO] Walter Bright. D Programming Language 1.0 Reference, Modules [34].Digital Mars. Octubre 2010.

[DWMT] Walter Bright. D Programming Language 1.0 Reference, Templatemixins [35]. Digital Mars. Octubre 2010.

[DWMX] Walter Bright. D Programming Language 1.0 Reference, Mixins [36].Digital Mars. Octubre 2010.

[DWNC] Walter Bright. D Programming Language 1.0 Reference, Nestedclasses [37]. Digital Mars. Octubre 2010.

[DWOO] Walter Bright. D Programming Language 1.0 Reference, Operatoroverloading [38]. Digital Mars. Octubre 2010.

[DWOV] Walter Bright. D Programming 1.0 Language Overview [39]. DigitalMars. Octubre 2010.

[DWPR] Walter Bright. D Programming Language 1.0 Reference, Properties [40].Digital Mars. Octubre 2010.

[DWSI] Walter Bright. D Programming Language 1.0 Reference, Staticif [41]. Digital Mars. Octubre 2010.

[DWSR] Walter Bright. D Programming Language 1.0 Reference, Strings [42].Digital Mars. Octubre 2010.

[DWST] Walter Bright. D Programming Language 1.0 Reference, Struct& unions [43]. Digital Mars. Octubre 2010.

[DWSY] Walter Bright. D Programming Language 1.0 Reference, Synchronizedstatement [44]. Digital Mars. Octubre 2010.

[DWTO] Walter Bright. D Programming Language 1.0 Reference, Typeof [45].Digital Mars. Octubre 2010.

[DWTP] Walter Bright. D Programming Language 1.0 Reference, Templates [46].Digital Mars. Octubre 2010.

[DWTY] Walter Bright. D Programming Language 1.0 Reference, Types [47].Digital Mars. Octubre 2010.

[DWUT] Walter Bright. D Programming Language 1.0 Reference, Unit tests [48].Digital Mars. Octubre 2010.

[33] http://www.digitalmars.com/d/1.0/memory.html
[34] http://www.digitalmars.com/d/1.0/module.html
[35] http://www.digitalmars.com/d/1.0/template-mixin.html
[36] http://www.digitalmars.com/d/1.0/mixin.html
[37] http://www.digitalmars.com/d/1.0/class.html#nested
[38] http://www.digitalmars.com/d/1.0/operatoroverloading.html
[39] http://www.digitalmars.com/d/1.0/overview.html
[40] http://www.digitalmars.com/d/1.0/property.html
[41] http://www.digitalmars.com/d/1.0/version.html#staticif
[42] http://www.digitalmars.com/d/1.0/arrays.html#strings
[43] http://www.digitalmars.com/d/1.0/struct.html
[44] http://www.digitalmars.com/d/1.0/statement.html#SynchronizedStatement
[45] http://www.digitalmars.com/d/1.0/declaration.html#typeof
[46] http://www.digitalmars.com/d/1.0/template.html
[47] http://www.digitalmars.com/d/1.0/type.html
[48] http://www.digitalmars.com/d/1.0/unittest.html

[DWVI] Walter Bright. D Programming Language 1.0 Reference, Voidinitializations [49]. Digital Mars. Octubre 2010.

[GCBIB] Richard Jones. The Garbage Collection Bibliography [50]. 1996-2009.

[GS85] L. Guibas and J. Stolfi. General Subdivisions and Voronoi Diagrams.ACM Trans. on Graphics Volumen 4, número 2, páginas 74-123. 1985.

[HDH03] Martin Hirzel, Amer Diwan, and Matthew Hertz, Proceedingsof the 18th Conference on Object-Oriented Programming, Systems,Languages, and Applications (OOPSLA 2003), Anaheim, CA, Oct. 26-30,2003.

[HEND02] Fergus Henderson. Accurate garbage collection in an uncooperativeenvironment. Proceedings of the Third International Symposium on MemoryManagement (2002). ACM. Páginas 150-156. Febrero, 2003.

[HIRZ03] Martin Hirzel and Amer Diwan and Matthew Hertz.Connectivity-based Garbage Collection [51]. Proceedings of the ACM OOPSLA'03 Conference on Object-Oriented Programming, Systems, Languages andApplications. Noviembre 2003.

[HUEL98] Lorenz Huelsbergen and Phil Winterbottom. Very ConcurrentMark-&-Sweep Garbage Collection without Fine-Grain Synchronization.Proceedings of the International Symposium on Memory Management. Páginas166-175. ACM. 1998. ISBN 1-58113-114-3.

[IEEE754] IEEE Task P754. IEEE 754-2008, Standard for Floating-PointArithmetic. IEEE. ISBN 0-7381-5753-8. 2008.

[JOLI96] Richard Jones, Rafael D Lins. Garbage Collection: Algorithmsfor Automatic Dynamic Memory Management. John Wiley & Sons, 1996.ISBN 0-471-94148-4.

[KAR77] R. Karp. Probabilistic analysis of partitioning algorithms for thetraveling-salesman problem in the plane. Mathematics of OperationsResearch Volumen 2, número 3, páginas 209-224. Agosto 1977.

[LINS05] Rafael D Lins. A New Multi-Processor Architecture forParallel Lazy Cyclic Reference Counting. Proceedings of the 17thInternational Symposium on Computer Architecture on High PerformanceComputing - Volume 00 (páginas 35-43), 2005. IEEE Press.

[LMTDGC] Leandro Lucarella. Luca's Meaninless Thoughts, tag DGC [52].2008-2010.

[LWN90311] Ingo Molnar. flexible-mmap-2.6.7-D5 [53]. Linux Weekly News.Junio 2004.

[MOLA06] Paolo Molaro. A new Mono GC [54]. Octubre 2006.

[MOLAWE] Paolo Molaro. Compacting GC [55]. The Mono Project. Obtenido enJunio de 2009.

[NAS00] Neil Schemenauer. Garbage Collection for Python [56]. 2000.

[NGA15246] Mason Green. Blaze 2.0. Grupo de noticiasdigitalmars.D.announce, 16 de marzo de 2009. Mensaje número 15246 [57].

[49] http://www.digitalmars.com/d/1.0/declaration.html#VoidInitializer
[50] http://www.cs.ukc.ac.uk/people/staff/rej/gcbib/gcbib.html
[51] http://www-plan.cs.colorado.edu/diwan/cbgc.pdf
[52] http://llucax.com.ar/blog/blog/tag/dgc
[53] http://lwn.net/Articles/90311/
[54] http://www.go-mono.com/meeting06/mono-sgen.pdf
[55] http://www.mono-project.com/Compacting_GC
[56] http://arctrix.com/nas/python/gc/
[57] http://www.digitalmars.com/d/archives/digitalmars/D/announce/Blaze_2.0_15246.html

[NGA19235] Leandro Lucarella. D Concurrent GC. Grupo de noticiasdigitalmars.D.announce, 9 de septiembre de 2010. Mensaje número 19235 [58].

[NGA6842] Walter Bright. Transitioning to a type aware GarbageCollector. Grupo de noticias digitalmars.D.announce, 22 de enero de2007. Mensaje número 6842 [59].

[NGA9103] Myron Alexander. ANN: WeakObjectReference - class to hold weakreferences. Grupo de noticias digitalmars.D.announce, 23 de junio de2007. Mensaje número 9103 [60].

[NGD13301] Pete Poulos. Weak references. Grupo de noticias digitalmars.D,5 de agosto de 2008. Mensaje número 13301 [61].

[NGD18354] Jarrett Billingsley. Does the GC ever perform a generationalcollect? Grupo de noticias digitalmars.D, 5 de marzo de 2005. Mensajenúmero 18354 [62].

[NGD22968] Maxime Larose. GC pauses. Grupo de noticias digitalmars.D, 2 demayo de 2005. Mensaje número 22968 [63].

[NGD2547] Mark T. Real time programming why not? Grupo de noticiasdigitalmars.D, 5 de enero de 2002. Mensaje número 2547 [64].

[NGD26273] Stewart Gordon. Copying/heap compacting GC. Grupo de noticiasdigitalmars.D, 24 de marzo de 2004. Mensaje número 26273 [65].

[NGD29291] Larry Evans. How does RTTI help gc?. Grupo de noticiasdigitalmars.D, 21 de octubre de 2005. Mensaje número 29291 [66].

[NGD35364] Frank Benoit. GC implementation. Grupo de noticiasdigitalmars.D, 18 de marzo de 2006. Mensaje número 35364 [67].

[NGD38689] Frank Benoit. GC, the simple solution. Grupo de noticiasdigitalmars.D, 4 de junio de 2006. Mensaje número 38689 [68].

[NGD42557] Lionello Lunesu. Is a moving GC really needed? Grupo de noticiasdigitalmars.D, 2 de octubre de 2006. Mensaje número 425577 [69].

[NGD43991] Andrey Khropov. [Performance] shootout.binarytrees whenimplemented with gc is 10x slower than C# on .NET 2.0. Grupo de noticiasdigitalmars.D, 11 de noviembre de 2006. Mensaje número 43991 [70].

[NGD44607] Russ Lewis. A TODO for somebody: Full Reflection Gets YouSmarter GC. Grupo de noticias digitalmars.D, 20 de noviembre de 2006.Mensaje número 44607 [71].

[58] http://www.digitalmars.com/d/archives/digitalmars/D/announce/D_Concurrent_GC_19235.html
[59] http://www.digitalmars.com/d/archives/digitalmars/D/announce/Transitioning_to_a_type_aware_Garbage_Collector_6842.html
[60] http://www.digitalmars.com/d/archives/digitalmars/D/announce/ANN_WeakObjectReference_-_class_to_hold_weak_references_9103.html
[61] http://www.digitalmars.com/d/archives/digitalmars/D/learn/weak_references_13301.html
[62] http://www.digitalmars.com/d/archives/digitalmars/D/18354.html
[63] http://www.digitalmars.com/d/archives/digitalmars/D/22968.html
[64] http://www.digitalmars.com/d/archives/2547.html
[65] http://www.digitalmars.com/d/archives/26273.html
[66] http://www.digitalmars.com/d/archives/digitalmars/D/29291.html
[67] http://www.digitalmars.com/d/archives/digitalmars/D/35364.html
[68] http://www.digitalmars.com/d/archives/digitalmars/D/38689.html
[69] http://www.digitalmars.com/d/archives/digitalmars/D/Is_a_moving_GC_really_needed_42557.html
[70] http://www.digitalmars.com/d/archives/digitalmars/D/Performance_shootout.binarytrees_when_implemented_with_gc_is_10x_slower_than_
[71] http://www.digitalmars.com/d/archives/digitalmars/D/A_TODO_for_somebody_Full_Reflection_Gets_You_Smarter_GC_44607.html

[NGD46407] Oskar Linde. The problem with the D GC. Grupo de noticiasdigitalmars.D, 8 de enero de 2007. Mensaje número 46407 [72].

[NGD54084] Babele Dunnit. Problems with GC, trees and array concatenation.Grupo de noticias digitalmars.D, 1ro de junio de 2007. Mensaje número54084 [73].

[NGD5622] Christian Schüler. D and game programming. Grupo de noticiasdigitalmars.D, 30 de mayo de 2002. Mensaje número 5622 [74].

[NGD5821] Antonio Monteiro. Thread and gc.fullCollect. Grupo de noticiasdigitalmars.D, 11 de enero de 2007. Mensaje número 5821 [75].

[NGD63541] Leonardo Maffi. A smaller GC benchmark. Grupo de noticiasdigitalmars.D, 10 de diciembre de 2007. Mensaje número 63541 [76].

[NGD67673] Leonardo Maffi. Slow GC? Grupo de noticias digitalmars.D, 13 demarzo de 2008. Mensaje número 67673 [77].

[NGD69761] Jarrett Billingsley. Weak references. Grupo de noticiasdigitalmars.D, 12 de abril de 2008. Mensaje número 69761 [78].

[NGD71869] Fawzi Mohamed. large objects and GC. Grupo de noticiasdigitalmars.D, 16 de mayo de 2008. Mensaje número 71869 [79].

[NGD74624] Pete Poulos. Weak References. Grupo de noticias digitalmars.D,6 de agosto de 2008. Mensaje número 74624 [80].

[NGD75952] Alan Knowles. Threading and the Garbage handler mess. Grupo denoticias digitalmars.D, 6 de septiembre de 2008. Mensaje número 75952 [81].

[NGD75995] David Simcha. Partially Static GC. Grupo de noticiasdigitalmars.D, 9 de septiembre de 2008. Mensaje número 75995 [82].

[NGD80695] David Simcha. Moving GC. Grupo de noticias digitalmars.D, 12 dediciembre de 2008. Mensaje número 80695 [83].

[NGD86840] Simon Treny. Keeping a list of instances and garbage-collection.Grupo de noticias digitalmars.D, 29 de marzo de 2009. Mensaje número86840 [84].

[NGD87831] Leandro Lucarella. Re: Std Phobos 2 and logging library? Grupode noticias digitalmars.D, 10 de abril de 2009. Mensaje número 87831 [85].

[NGD88065] Jason House. D2 weak references. Grupo de noticiasdigitalmars.D, 15 de abril de 2009. Mensaje número 88065 [86].

[72] http://www.digitalmars.com/d/archives/digitalmars/D/The_problem_with_the_D_GC_46407.html
[73] http://www.digitalmars.com/d/archives/digitalmars/D/Problems_with_GC_trees_and_array_concatenation_54084.html
[74] http://www.digitalmars.com/d/archives/5622.html
[75] http://www.digitalmars.com/d/archives/digitalmars/D/learn/thread_and_gc.fullCollect_5821.html
[76] http://www.digitalmars.com/d/archives/digitalmars/D/A_smaller_GC_benchmark_63541.html
[77] http://www.digitalmars.com/d/archives/digitalmars/D/Slow_GC_67673.html
[78] http://www.digitalmars.com/d/archives/digitalmars/D/Weak_references._69761.html
[79] http://www.digitalmars.com/d/archives/digitalmars/D/large_objects_and_GC_71869.html
[80] http://www.digitalmars.com/d/archives/digitalmars/D/Weak_References_74624.html
[81] http://www.digitalmars.com/d/archives/digitalmars/D/Threading_and_the_Garbage_handler_mess._75952.html
[82] http://www.digitalmars.com/d/archives/digitalmars/D/Partially_Static_GC_75995.html
[83] http://www.digitalmars.com/d/archives/digitalmars/D/Moving_GC_80695.html
[84] http://www.digitalmars.com/d/archives/digitalmars/D/Keeping_a_list_of_instances_and_garbage-collection_86840.html
[85] http://www.digitalmars.com/d/archives/digitalmars/D/Std_Phobos_2_and_logging_library_87794.html#N87831
[86] http://www.digitalmars.com/d/archives/digitalmars/D/D2_weak_references_88065.html

[NGD88298] Leandro Lucarella. GC object finalization not guaranteed. Grupode noticias digitalmars.D, 18 de abril de 2009. Mensaje número 88298 [87].

[NGD88559] Andrei Alexandrescu. -nogc. Grupo de noticias digitalmars.D, 23de abril de 2009. Mensaje número 88559 [88].

[NGD89302] David Simcha. Destructors and Deterministic Memory Management.Grupo de noticias digitalmars.D, 3 de mayo de 2009. Mensaje número89302 [89].

[NGD89394] David Simcha. Associative Arrays and Interior Pointers. Grupo denoticias digitalmars.D, 4 de mayo de 2009. Mensaje número 89394 [90].

[NGD90977] Anónimo. Why allocation of large amount of small objects so slow(x10) in D? Grupo de noticias digitalmars.D, 21 de mayo de 2009. Mensajenúmero 90977 [91].

[NGL13744] Alan Knowles. Any advice for GC/memory debugging. Grupo denoticias digitalmars.D.learn, 1ro de septiembre de 2008. Mensaje número13744 [92].

[NGL3937] Chris K. GC and realtime threads. Grupo de noticiasdigitalmars.D.learn, 12 de julio de 2006. Mensaje número 3937 [93].

[NGL8264] Robert Fraser. Soft/weak references? Grupo de noticiasdigitalmars.D.learn, 21 de junio de 2007. Mensaje número 8264 [94].

[PAN09] Vladimir Panteleev. Memory Management in the D ProgrammingLanguage [95]. Proyecto de licenciatura. Ministerul Educaţiei şi Tineretuluial Republicii Moldova Universitatea Tehnică a Moldovei Facultatea deCalculatoare, Informatică şi Microelectronică Catedra Filiera Anglofonă.2009.

[PHP530] PHP Team. PHP 5.3.0 Release Announcement [96]. PHP news archive.2009-06-30.

[RODR97] Gustavo Rodriguez-Rivera and Vince Russo. Non-intrusive CloningGarbage Collection with Stock Operating System Support. Software Practiceand Experience Volumen 27, Número 8. Agosto 1997.

[SHO10] Brent Fulgham. The Computer Language Benchmarks Game [97].2004-2010.

[SUTT99] Herb Sutter. Exceptional C++: 47 Engineering Puzzles,Programming Problems, and Solutions, 1ra edición. Addison-WesleyProfessional, 1999. ISBN 0-201-61562-2.

[TT1997] Leandro Lucarella. Integrate CDGC (D Concurrent GarbageCollector) [98]. Tango Issue Tracker. Octubre 2010.

[WBB10] Walter Bright. C++ Compilation Speed [99]. Dr. Dobb's Blog. Agosto2010.

[87] http://www.digitalmars.com/d/archives/digitalmars/D/GC_object_finalization_not_guaranteed_88298.html
[88] http://www.digitalmars.com/d/archives/digitalmars/D/nogc_88559.html
[89] http://www.digitalmars.com/d/archives/digitalmars/D/Destructors_and_Deterministic_Memory_Management_89302.html
[90] http://www.digitalmars.com/d/archives/digitalmars/D/Associative_Arrays_and_Interior_Pointers_89394.html
[91] http://www.digitalmars.com/d/archives/digitalmars/D/why_allocation_of_large_amount_of_small_objects_so_slow_x10_in_D_90977.html
[92] http://www.digitalmars.com/d/archives/digitalmars/D/learn/Any_advice_for_GC_memory_debugging_13744.html
[93] http://www.digitalmars.com/d/archives/digitalmars/D/learn/3937.html
[94] http://www.digitalmars.com/d/archives/digitalmars/D/learn/Soft_weak_references_8264.html
[95] http://thecybershadow.net/d/Memory_Management_in_the_D_Programming_Language.pdf
[96] http://php.net/releases/5_3_0.php
[97] http://shootout.alioth.debian.org/
[98] http://www.dsource.org/projects/tango/ticket/1997
[99] http://www.drdobbs.com/blog/archives/2010/08/c_compilation_s.html