Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Proyecto Fin de Carrera de Ingeniería en Informática

Estudio Comparativo de Frameworks de Instrumentación Dinámica de Ejecutables

Juan Antonio Artal Lozano

Director: Ricardo J. Rodríguez Fernández

Ponente: José Javier Merseguer Hernáiz

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Abril de 2012
Curso 2011/2012

A mi mujer, Conchita.

Those types are not 'abstract';
they are as real as `int` *and* `float`*.*
Doug McIlroy

Agradecimientos

A mis padres, hermanos y abuela.
A todos mis compañeros de promoción.
A Ricardo y su paciencia conmigo.
A José Merseguer.
A mis compañeros de trabajo.
GRACIAS.

Estudio comparativo de frameworks de Instrumentación Dinámica de Ejecutables

RESUMEN

La Instrumentación Dinámica de Ejecutables *(Dynamic Binary Instrumentation*, DBI) es una técnica muy potente que permite analizar el comportamiento, en tiempo de ejecución, de cualquier aplicación. DBI se puede usar, por ejemplo, para contar el número de instrucciones que ejecuta o contar todas las transferencias (lectura y/o escritura) a memoria que realiza un determinado programa.

DBI tiene diferentes usos según sea el perfil de la persona que lo use. Por ejemplo, para un programador, DBI ayudará a identificar las partes críticas del código; para un desarrollador de un procesador nuevo, DBI simulará esta nueva arquitectura; y para un programador de compiladores en una nueva arquitectura, DBI ayudará a la colocación de las instrucciones para mejorar el paralelismo o cómo preparar *profile-guided optimizacions* (PGO).

Un framework de DBI es una plataforma software que incluye programas, librerías, documentación y una API para manipulación de instrucciones en tiempo de ejecución. Existen diferentes frameworks de DBI (p.e., Pin, Valgrind, DynamoRIO, Paradyn/Dyninst), que proporcionan APIs muy extensas para que cada ingeniero pueda desarrollar sus propias herramientas de análisis dinámico, llamadas herramientas DBA (*Dynamic Binary Analysis*). Las herramientas DBA permiten analizar, generar optimizaciones y monitorizar el comportamiento de programas.

El objetivo de este PFC es realizar un **estudio comparativo** centrado a nivel de impacto en rendimiento (*performance*) de diferentes frameworks de DBI. Es decir, se comprobará el rendimiento de una aplicación ejecutada de forma nativa, sin instrumentar, y se comparará con esta misma aplicación instrumentada por herramientas programadas bajo diferentes frameworks de DBI. De esta forma se obtiene el impacto en rendimiento de cada uno de los frameworks. Para poder llevar a cabo este estudio, se han seleccionado un conjunto de aplicaciones para crear un **benchmark**, que nos dará información de rendimiento de cada framework de DBI.

Además, se pretende comparar cada framework de DBI atendiendo a las siguientes características: plataformas y tipos de ejecutables que aceptan, necesidad de disponer del código fuente, API proporcionada, facilidad de programación de herramientas DBA, licencia/coste y la posibilidad de vincular a un proceso en ejecución.

Índice

Indice de figuras

Indice de tablas

Capítulo 1

Introducción

El término de instrumentación se refiere a la inserción de código adicional sobre un determinado software. Principalmente, hay dos tipos de instrumentación: de código fuente, cuando el programador añade líneas de código antes de la compilación; y de ejecutable, si hay otra aplicación que modifica el programa una vez compilado. La instrumentación permite incorporar al programa desarrollado código adicional para recoger información en tiempo de ejecución, que principalmente tiene dos usos diferentes: para estudio de arquitecturas, donde se puede hacer modelado de cachés y simulación de instrucciones nuevas de procesadores; y para análisis de código, donde se puede generar información para análisis de rendimiento, p.e., para averiguar cuándo, dónde y por qué nuestro código tarda tanto en ejecutar cierta tarea.

Hay diferentes tipos de instrumentación, como la manual, en la que el programador añade directamente las líneas de código que le interesan; como por ejemplo, para calcular tiempos de ejecución, contar eventos o llamadas a una interfaz de programación de aplicaciones (API). Hay herramientas tipo *automated source level* que modifican el código fuente para añadir instrumentación de acuerdo a una determinada configuración. Existen otros tipos de instrumentación como la asistida por el compilador, que es añadida en tiempo de compilación, y *binary translation*, donde se modifica el software con llamadas a una API de instrumentación para que la propia aplicación genere información en tiempo de ejecución, siendo esta aplicación la que se instrumenta a sí misma. Finalmente, existe la instrumentación dinámica de ejecutables (DBI), donde una aplicación externa inserta código adicional en tiempo de ejecución al ejecutable instrumentado. Este proyecto se centra en DBI porque es la opción más completa de todas y más moderna, permitiendo además monitorizar y controlar una aplicación mientras se ejecuta desde su inicio hasta el final.

Sin embargo, una desventaja que tiene la instrumentación al ser añadida en tiempo de ejecución es la sobrecarga tan elevada que conlleva su adición. Debido a esto, las aplicaciones instrumentadas tienen un rendimiento muy malo comparándolas consigo mismas ejecutándose de forma nativa, es decir, sin instrumentar. Esto es un factor determinante a la hora de trabajar con DBI.

Para programar aplicaciones que soporten DBI se puede usar un framework de DBI.

Éste facilita una API para que se puedan desarrollar herramientas de análisis dinámico. Estas herramientas permiten instrumentar un software en el momento que el programador determine.

En la actualidad existen diferentes frameworks de DBI, como Valgrind [NS07] o Pin [LCM+05], pero no hay suficiente información comparativa sobre ellos a nivel de rendimiento, ya que apenas hay trabajos que traten directamente el tema de rendimiento, y a nivel de programación no se ha encontrado nada que compare los frameworks de DBI.

1.1. Objetivo

El objetivo de este PFC es realizar un estudio comparativo a nivel de impacto en rendimiento de diferentes frameworks de DBI, ya que una aplicación instrumentada puede tardar en ejecutarse hasta 35 veces más lenta, como se ha comprobado con los resultados de este PFC. Se ejecutarán diferentes tipos de aplicaciones seleccionadas para el estudio, mediante un benchmark propio, bajo diferentes instrumentaciones en diferentes frameworks de DBI. Posteriormente se analizarán los resultados obtenidos.

El procedimiento para obtener esos resultados ha sido el siguiente:

- Hacer un estudio de los frameworks disponibles, buscando cuáles son los que se están usando en la actualidad.
- Seleccionar los frameworks interesantes para este estudio, indicando los criterios de selección que se han utilizado.
- Compilación e instalación del framework a partir del código fuente, para comprobar que se dispone de todo el software necesario para después desarrollar herramientas.
- Estudio de manuales, tutoriales y APIs de cada framework, para después poder desarrollar herramientas DBA.
- Búsqueda de aplicaciones estándar para generar un benchmark. Éste reúne un conjunto de programas en diferentes categorías como: cálculo entero, cálculo real, E/S de ficheros y acceso a memoria.
- Probar herramientas DBA desarrolladas bajo el benchmark, de tal forma que permita obtener información sobre el rendimiento de estas.

Finalmente el benchmark permite obtener los datos de sobrecarga en tiempo de las aplicaciones por la instrumentación y los requisitos adicionales de memoria.

1.2. Motivación

La principal motivación para la elección de este PFC fue la oportunidad de profundizar en el aprendizaje y estudio de herramientas DBI actuales, ya que en las asignaturas de la

carrera no había nada relacionado con ello. Entre las aplicaciones innovadoras de estas herramientas son las relacionadas con la seguridad como la comprobación de fugas de memoria e ingeniería inversa.

Este Proyecto me ha ofrecido la oportunidad de explorar una interesante aplicación: la evaluación del rendimiento del los frameworks de DBI basada en la programación de herramientas DBA y el estudio de sus APIs.

1.3. Organización del documento

El presente documento está dividido en dos partes: la memoria, donde se explica el desarrollo del Proyecto; y los apéndices, donde se amplía la información de ciertos puntos relevantes.

El capítulo 2 define algunos conceptos previos que sirven de ayuda para comprender el resto del documento como qué es DBI, DBA, la granularidad en instrumentación y cómo fueron los inicios de la instrumentación dinámica. El capítulo 3 introduce los frameworks de DBI, expone los criterios de selección para la comparación, sus características, similitudes y diferencias entre ellos y algunos comentarios sobre sus APIs. El capítulo 4 recoge el trabajo relacionado con este proyecto. El proceso de creación del benchmark y la selección de software para las pruebas está en el capítulo 5. El capítulo 6 resume los experimentos realizados con el benchmark definido anteriormente. Además, se muestran los gráficos más relevantes junto a un análisis crítico de los resultados. Finalmente, el capítulo 7 presenta las conclusiones de este trabajo y plantea posibles líneas de trabajo futuro.

Respecto a los apéndices, el apéndice A es donde se hace balance del esfuerzo temporal empleado en la realización del PFC. El apéndice B reúne los problemas que han ido apareciendo. El apéndice C describe las aplicaciones usadas en los benchmark con más detalle. El apéndice D contiene las tablas con los resultados obtenidos de los experimentos y no incluidas en el capítulo 6 y finalmente, en el apéndice E se presenta el código fuente de las aplicaciones DBA para el benchmark.

Capítulo 2

Conocimientos previos

En este capítulo se definen algunos de los conceptos más importantes en los que se basa este proyecto y, en general, se explica el funcionamiento y uso de DBI.

El principal uso de DBI es analizar el comportamiento de un ejecutable durante la ejecución, de tal forma que permita mejorar el funcionamiento de éste. En comparación con el análisis estático, ofrece la ventaja de estudiar qué es lo que está ocurriendo en vez de lo que podría estar ocurriendo. El único inconveniente es que no se ejecutan todos los posibles caminos, porque si una instrucción no se ejecuta, no se llega a instrumentar. DBI se puede utilizar de manera diferente dependiendo de quién lo vaya a utilizar, para un programador, DBI ayudará a identificar las partes críticas del código; para un desarrollador de un procesador nuevo, DBI simulará esta nueva arquitectura; y para un programador de compiladores en una nueva arquitectura, DBI ayudará a la colocación de las instrucciones para mejorar el paralelismo o cómo preparar *profile-guided optimizacions* (PGO).

En un framework de DBI hay dos componentes principales, el **núcleo** y las **herramientas desarrolladas** con él. El núcleo se encarga de enviar fragmentos de código a la herramienta, y ésta se encarga de la inyección de código.

El núcleo es como un compilador *just-in-time* (JIT), donde la entrada al compilador es un ejecutable. Se intercepta la ejecución de la primera instrucción del ejecutable y genera nuevo código, donde se transfiere el control de la secuencia generada. La secuencia de código generada es prácticamente idéntica a la original, pero el núcleo se asegura que se retorne el control cuando se salga de la secuencia. El código generado es guardado en memoria, por lo que puede ser reutilizado sin necesidad de regenerarlo cada vez que se ejecute. Una vez que se ha generado este código se le da la opción al usuario de inyectar su propio código, o sea instrumentarlo.

Una herramienta habitualmente tiene la forma *plug-in* o librería, y su función es añadir código al obtenido previamente del núcleo, para ello tiene dos componentes básicos:

- Instrumentación, se decide dónde y qué código es insertado.
- Análisis, se ejecuta el código añadido en los puntos de inserción.

Cuando se desarrollan herramientas, es más importante afinar el código de análisis que el de instrumentación. Esto es así debido a que la parte de instrumentación en una línea del código se ejecuta una única vez; sin embargo, el análisis, que es el código que se ha inyectado, se puede llegar a ejecutar múltiples veces. Todo el código inyectado al ejecutable original se ejecuta de forma transparente [BZA12] con los frameworks de DBI actuales, de tal forma que este código añadido no pueda interferir en el comportamiento del ejecutable y se modifique el comportamiento original.

El núcleo y la herramienta habitualmente controlan el programa desde el inicio, es decir, desde la primera instrucción a ejecutar. Para los ejecutables enlazados con librerías dinámicas esto implica que la ejecución del cargador dinámico y de las librerías es visible y controlada. También son visibles y controlados el código generado dinámicamente, pero el que se automodifica puede llegar a dar problemas en función del framework de DBI, como en el caso del framework Valgrind [NS07].

Para que el funcionamiento sea correcto, tanto el núcleo como la herramienta tienen que estar trabajando en el mismo espacio de direcciones que el ejecutable. Es decir, tienen que estar todos en espacio de usuario, donde residen las aplicaciones; o bien en espacio del *kernel*, donde residen los módulos o *drivers*.

2.1. Granularidad en DBI

Un ejecutable instrumentado por un framework de DBI se suele instrumentar instrucción a instrucción, pero en función de la instrumentación que se desee realizar y del framework se puede utilizar una granularidad diferente. Las posibles granularidades son:

- *Instrucción*, es la unidad mínima que se puede instrumentar. Son instrucciones en ensamblador de la arquitectura en la que se trabaje.

- *Bloque básico*, es una secuencia de instrucciones que finalizan con una instrucción de control de transferencia como un salto condicional (p.e., en ensamblador x86, `JZ`, salto si flag Z=1)[Int86], incondicional (p.e., `JMP`, salto a una dirección), repeticiones (p.e., instrucciones que tengan el prefijo `REP`, repite la instrucción posterior varias veces), llamada o retorno a procedimiento (p.e., `RET`, retorno de procedimiento) entre otros.

  Aquí se muestra un ejemplo de bloque básico consistente en tres instrucciones x86: una suma entre dos registros del procesador dejando el resultado en el primero de ellos (`ADD`), una comparación entre un número y un registro (`CMP`) y un salto condicional que comprueba si el resultado de la operación previa es menor o igual (`JLE`). Como la última instrucción es de control de transferencia, tras la instrucción del salto, finaliza el bloque básico.

  ```
  comparac: add %ebx,%eax
            cpm $0x7f,%ebx
            jle comparac
  ```

- *Superbloque*, es también una secuencia de instrucciones que tiene un punto de entrada, pero al contrario que los bloques básicos puede tener múltiples puntos de salida.
- *Traza*, es la unión de bloques básicos que se ejecutan uno detrás de otro en secuencia, aunque en el ejecutable no estén consecutivos.
- *Rutina*, corresponde a las funciones y procedimientos típicamente producidos por un compilador de un lenguaje de programación por procedimientos como C.
- *Imagen*, que representa a todas las secciones de un ejecutable, que son las partes en las que se divide, como p.e., `.init`, `.text` o `.fini` para el formato de fichero ejecutable para windows. Hay que tener en cuenta que durante la ejecución de un proceso puede haber más de un objeto imagen en función de las librerías dinámicas a las que acceda.

2.2. Origen de DBI

Las primeras herramientas de instrumentación hacían dos tareas básicas: contar bloques básicos y la generación de trazas de direcciones para modelado de cachés. Entre otras, estaban las herramientas Pixie [SG92], Epoxie [Wal91] y QPT [LB94] que utilizaban el ejecutable ya compilado. Presentaban diferentes problemas, como que no se podía hacer otro tipo de instrumentación y además generaban trazas de datos y direcciones de manera ineficiente, ya que no se podía seleccionar entre qué puntos se quería generar información.

Otro tipo de herramientas eran los simuladores, como Tango Lite [GH93], Proteus [BDCW91] o g88 [Bed90]. Proteus permitía estudiar el comportamiento de un programa con diferentes arquitecturas de cachés y un número simulado de procesadores para poder comprobar la escalabilidad de un programa o algoritmo. El principal problema de los simuladores es la sobrecarga en tiempo que generan, esto es algo que se ha mantenido hasta las herramientas actuales. Además, no eran completamente transparentes para el programa y modificaban el comportamiento del ejecutable.

El primer framework de DBI, ATOM [SE94], apareció en 1993 y funcionaba únicamente para Tru64 Unix en procesadores Alpha. Proveía una API mediante la cual se podían programar herramientas para analizar un ejecutable. Ofrecía la instrumentación de instrucciones, bloques básicos y rutinas; y se podían construir simuladores a nivel de caché e instrucciones. Sin embargo, la mayor desventaja es que se tenía que modificar el código fuente y recompilarlo. El nuevo programa utilizaba las librerías de ATOM y las instrucciones eran directamente ejecutadas bajo el procesador real, sin ningún tipo de simulación.

Capítulo 3

Frameworks de DBI

Este capítulo muestra una introducción a los frameworks de DBI. Se resumen los que se pueden encontrar, después se analizan los criterios de selección y cuáles han sido los seleccionados para hacer el estudio de rendimiento.

Un framework de DBI ofrece un conjunto de APIs de manipulación de instrucciones en tiempo de ejecución para que se puedan hacer, de manera fácil y rápida, herramientas de instrumentación. Los principales frameworks de DBI que se pueden encontrar son:

- **Pin** [LCM+05] (`http://pintool.org`) es un sistema de instrumentación desarrollado para proveer facilidad de uso, portabilidad, transparencia e instrumentación eficiente. Se programa en C/C++ y se creó a partir de ATOM [SE94]. Se crean herramientas DBA *ligeras*, esto significa que se añade la instrumentación y se ejecuta directamente en el procesador de la arquitectura.

- **DynamoRIO** [Bru04] (`http://dynamorio.org`), es un sistema de manipulación de código en tiempo de ejecución que soporta transformaciones de código en cualquier parte de un programa mientras se está ejecutando. Con su API se pueden programar herramientas para análisis de programas, *profiling*, instrumentación, optimización y *binary translation* entre otros. Provee manipulación de código eficiente, transparente y extensa en aplicaciones sin necesidad de recompilarlas. Las herramientas creadas son ligeras.

- **Valgrind** [NS07], (`http://valgrind.org`), es un framework de DBI desarrollado para crear herramientas DBA *pesadas*, esto es, convierte el binario a un lenguaje intermedio, y guarda el estado de todos los registros y memoria accedidos, así como todas las operaciones de lectura y escritura, asignaciones y liberaciones de memoria. Para todo esto, usa una técnica llamada *shadow values*. Es por eso que herramientas DBA ligeras programadas con Pin y DynamoRIO son más rápidas, pero sin embargo, las *pesadas* son más difíciles de hacer o imposibles con esos frameworks.

- **DynInst** (`http://dyninst.org`), ofrece un API para modificar aplicaciones en tiempo de ejecución, con la capacidad de crear herramientas portables proporcio-

nando abstracciones independientes de la arquitectura. Crea herramientas DBA ligeras.

- **Dtrace** (`http://opensolaris.org/os/community/dtrace`) es un framework de rastreo y monitoreo abarcativo y dinámico. Se programa en lenguaje D y fue creado para diagnosticar problemas en el *kernel* y en aplicaciones en tiempo real. Es para Sistema Operativo Solaris.

- **Systemtap** (`http://sourceware.org/systemtap`), provee una infraestructura para simplificar la recogida de información en sistemas GNU/Linux. Ofrece instrumentación vía línea de comando y mediante un lenguaje script propio para acceder al *kernel* y a las aplicaciones de usuario.

- **HDTrans** [SSNB06] (`http://srl.cs.jhu.edu/projects`), es un sistema de instrumentación dinámica ligera para la arquitectura x86 *open-source* y debido a esto se ha optimizado para la simplicidad y modificabilidad. Permite instrumentar con las granularidades de instrucción, bloque básico y traza.

A continuación se definen las características para elegir los frameworks presentados. Para esta selección se ha buscado que cumplan determinados criterios. Uno de las más importantes es que fuera software que actualmente se mantenga **en desarrollo** y no fueran proyectos iniciados y olvidados, por lo que se han buscado frameworks cuyas últimas versiones fueran de 2011 ó 2012. La más reciente que se ha encontrado es DynamoRIO de enero de 2012. Otra característica es que tengan un **tipo de licencia** que permite acceder a su código fuente, siendo además su obtención de manera **gratuita**. Además, que posean una **amplia API** que permita el desarrollo de herramientas/clientes.

Aunque la mayoría de los frameworks son multiplataforma y soportan una amplia variedad de sistemas operativos, como Windows, GNU/Linux, Mac OS X, FreeBSD, Meego o Android; y están soportados para diferentes arquitecturas como x86, x64, Itanium, ARM, PowerPC o S/390 se buscaba que tuvieran un **Sistema Operativo y arquitectura común**.

En base a los criterios planteados, los frameworks que se van a estudiar para evaluar su rendimiento en este PFC son **Pin**, **Valgrind** y **DynamoRIO**. A continuación se describen más ampliamente los detalles.

3.1. Pin

Pin fue diseñado para proveer una funcionalidad similar a la herramienta ATOM de Tru64 para Alpha (visto en la sección 2.2) pero sin necesidad de recompilar la aplicación y soportando los sistemas operativos Linux y Windows. Puede inyectar código escrito en C o C++ en lugares arbitrarios del ejecutable. Además, provee una API muy potente que añade una capa de abstracción permitiendo al programador trabajar de forma transparente sin que le afecte el código original que se esté ejecutando. Su API está muy bien documentada y ofrece una gran amplitud de ejemplos para el programador. Una

opción muy interesante es que se puede vincular a un proceso en ejecución para que sólo instrumente la parte de código que interese.

Es desarrollado por Intel, y aunque lleva una licencia propietaria el coste es gratuito para uso no comercial. Un inconveniente que tiene es que sólo funciona correctamente con procesadores Intel. Puede funcionar con procesadores de las arquitecturas x86 y x64 de otras marcas (p.e., AMD, Cyrix), pero advierten que puede haber incompatibilidades con instrucciones propietarias o incompatibles de estos procesadores.

Es un software que se actualiza mucho, ya que en 2011 liberó las últimas revisiones de las versiones 2.8, 2.9 y 2.10, la última en noviembre de 2011.

3.2. Valgrind

Valgrind es multiplataforma, ya que soporta cinco arquitecturas diferentes bajo Linux, Android y OS X. La característica principal de Valgrind es que cada vez que lee una instrucción, antes de instrumentarla la convierte a un lenguaje intermedio tipo RISC, independiente de la arquitectura, denominado VEX IR. Es en el proceso de transformación entre VEX IR y el ensamblador de la arquitectura donde se produce la mayoría de la sobrecarga generada por Valgrind. Debido a este proceso de transformación a un lenguaje intermedio no es posible la vinculación a un proceso ya iniciado, siendo obligatorio que se tome el control desde el principio de la ejecución. Además, esto implica otros problemas, como se recoge en el apéndice B.2.

Utiliza la técnica de *shadow values*, que para cada registro o dirección de memoria anota un valor que lo identifica (p.e., inicializado o sin inicializar). Almacena y propaga estos valores en paralelo junto con el valor real del programa. De esta forma se pueden encontrar muchos tipos de *bugs* o problemas de seguridad. Con esta técnica se pueden desarrollar herramientas pesadas, que son más lentas durante la ejecución que otras desarrolladas con otros frameworks como Pin y DynamoRIO, pero que con éstos o no se pueden o son más difíciles de hacer.

Valgrind incorpora numerosas herramientas DBA, como: `memcheck` un comprobador de fallo en asignaciones de memoria; `cachegrind` un profiler de cachés y de predicciones de saltos; y `helgrind` comprobador de errores en threads.

El programador dispone de la información básica para comenzar con Valgrind, pero con muy pocos ejemplos de cómo manejar la API, que no está prácticamente explicada. Su licencia de uso es GNU GPL v2.

A lo largo de 2011 se liberaron las versiones 3.6.1 y 3.7.0, la última en noviembre de 2011.

3.3. DynamoRIO

DynamoRIO proviene de la unión de Dynamo, un optimizador de código en ejecución (desarrollado por HP Labs) junto con The RIO Project, otro optimizador e instrospeccionador en tiempo de ejecución (desarrollado por el MIT). Después fue adquirido por

VmWare y desde 2010 tiene el patrocinio de Google. Está completamente orientado al desarrollo de aplicaciones usando su API, en comparación con Valgrind que está orientado al uso de sus herramientas. Aunque en la versión actual todavía no se puede vincular a un proceso en ejecución, se quiere hacer posible para versiones posteriores.

La mayor ventaja que tiene DynamoRIO para los programadores es la facilidad de uso de su API. Explica claramente cómo hacer una aplicación desde cero, hay muchos tutoriales, hay una gran multitud de ejemplos de uso y su API está muy bien explicada. La licencia de uso es BSD-2.

En 2011 se liberaron las versiones 2.1 y 2.2, y la última versión, la 3.1, ha sido liberada en enero de 2012.

3.4. Similitudes y diferencias entre estos frameworks

Para poder trabajar con todos los frameworks en el mismo entorno de sistema operativo y arquitectura se ha buscado un punto en común. La Tabla 3.1 resume todas las opciones con las que se puede trabajar. Las opciones que tienen en común y se han remarcado en la tabla, son: Linux/x86 y Linux/x64. Entre estas dos opciones, se continúa el estudio con sistema operativo GNU/Linux funcionando en un procesador Intel x86.

	Pin	**Valgrind**	**DynamoRIO**
Linux/x86	✓	✓	✓
Linux/x64	✓	✓	✓
Linux/Itanium	✓		
Linux/PowerPC		✓	
Linux/s390		✓	
Linux/ARM	✓	✓	
FreeBSD/x64	✓		
OS X/x86	✓	✓	
OS X/x64		✓	
Windows/x86	✓		✓
Windows/x64	✓		✓
Android/ARM		✓	

Tabla 3.1: S.O./Arquitecturas soportadas por framework.

A continuación se muestran las similitudes entre ellos. Todos estos frameworks no necesitan recompilar el ejecutable, o re-enlazarlo, ya que directamente trabajan con el binario. Hacen la instrumentación en el momento antes de ejecutar una parte, y son capaces de descubrir código en tiempo de ejecución. Aunque no necesiten el código

fuente de una aplicación, está recomendado compilar con la opción para que se genere la información de depuración y sea más sencillo analizar el correcto funcionamiento de la instrumentación. También recomiendan todos no utilizar en el compilado las opciones de optimización, ya que en ocasiones se puede observar un comportamiento diferente al esperado.

Para crear la herramienta/cliente se suele programar en C/C++, con llamadas a la API del framework y genera o bien una librería o bien un ejecutable. La forma habitual de utilizar una herramienta creada con estos frameworks, también es muy similar, suele ser como sigue:

```
$ <nucleo de Framework> <herramienta/cliente> <ejecutable a instrumentar>
```

Esta herramienta/cliente es la que se encargará de la parte de instrumentación donde puede introducir el código que queramos, o bien sustituir una parte de él. Este código puede ser en C, C++ o en ensamblador de la arquitectura.

Pin y DynamoRIO trabajan con instrucciones en código máquina de la arquitectura, pero no como Valgrind, ya que éste traduce de ensamblador a una representación intermedia independiente de la arquitectura, como se ha comentado en la sección 3.2.

Pin es el único framework que soporta que se pueda vincular a un proceso que ya esté en ejecución. Esta es una opción muy conveniente porque se puede lanzar la aplicación de forma nativa, sin sobrecargas, y cuando llegue el momento que interese, se lanza la herramienta de instrumentación. Pin también permite la desvinculación del proceso.

Llegados a este punto se van a mostrar las diferencias entre los frameworks. En instrumentación hay dos modos de uso diferentes: modo *just-in-time* (JIT) y modo Probe. La forma más común de modo de ejecución es usar un compilador en modo JIT, que regenera una copia modificada de un pequeño trozo de instrucciones inmediatamente antes de ejecutar esas instrucciones. Las instrucciones modificadas son cacheadas desde donde podrán ser reutilizadas para el resto del tiempo de ejecución. El modo JIT es el modelo de ejecución más robusto y es con el que se obtiene un mayor rendimiento en las aplicaciones que reutilizan código (como el basado en bucles) ya que la sobrecarga de regenerar la copia cacheada puede ser amortizada a través del tiempo de ejecución del programa. Para programas muy cortos o con pocas iteraciones es más difícil amortizar la sobrecarga de la regeneración del código JIT.

En el modo Probe el ejecutable es parcheado en memoria y esta versión modificada será usada durante el tiempo de ejecución, en vez de una copia cacheada. La sobrecarga de esta técnica es mucho menor ya que se está ejecutando todo el tiempo código nativo. Los tres frameworks soportan el modo JIT, pero el modo Probe sólo está soportado por Pin y DynamoRIO.

Sobre las granularidades comentadas en la seccion 2.1, no todos los frameworks soportan las mismas. La Tabla 3.2 resume las granularidades de cada framework de DBI.

Finalmente, en la Tabla 3.3 se resumen las similitudes y diferencias nombradas entre los tres frameworks de DBI considerados en el estudio.

	Pin	**Valgrind**	**DynamoRIO**
instrucción	✓	✓	✓
bloque básico	✓		✓
superbloque		✓	
traza	✓		✓
rutina	✓		
imagen	✓		

Tabla 3.2: Granularidades soportadas por framework.

	Pin	**Valgrind**	**DynamoRIO**
Fecha última *release*	11/2011	11/2011	01/2012
Licencia	Tipo BSD	GNU GPL v2	BSD-2
Código fuente libre	✓	✓	✓
Tipo de herramientas DBA	Ligera	Pesada	Ligera
Lenguaje de programación herramientas	C/C++	C/C++	C/C++
Se ejecuta en arquitectura	local	VEX IR	local
Vinculación a procesos en ejecución	✓		
Modo de ejecución	JIT/Probe	JIT	JIT/Probe
Granularidades diferentes	5	2	3

Tabla 3.3: Similitudes y diferencias entre frameworks.

Capítulo 4

Trabajo relacionado

Este capítulo reúne los trabajos previos en los que se evalúa el rendimiento de frameworks de DBI.

Uh et al. en [PA06] describen un método para analizar el rendimiento en herramientas desarrolladas con frameworks de DBI. El método que define hace que sea más fácil identificar el origen de la sobrecarga para encontrar su causa principal. Se prueban dos instrumentaciones: cuenta de bloques básicos y adición de instrucciones con acceso a memoria. Este método se prueba únicamente con Pin. El benchmark que se usa es SPEC CINT 2000 [Cor06] y la máquina para realizar las pruebas es una máquina IA32, con un procesador Intel Xeon 2.8Ghz, bajo GNU/Linux 2.4.21, y compilado con gcc 3.3.2. Como resultados, obtienen un método para poder identificar de dónde viene la sobrecarga durante instrumentación, poder identificar las causas e intentar solucionarlo.

En [Sof07], Guah et al. realizan varios experimentos de rendimiento en instrumentación con Strata, que es un entorno de ejecución virtual que soporta *software dynamic translation* (SDT), esto es, conversión de un juego de instrucciones a otro en tiempo de ejecución donde posteriormente se pueden instrumentar. En uno de los experimentos se comprueba el rendimiento de Strata contra Pin, Valgrind y DynamoRIO. Utilizan como benchmark para medir el rendimiento SPEC CINT 2000. La métrica que utilizan es el *slowdown*, que se define como el tiempo de ejecución con instrumentación comparado con la ejecución nativa. Para esta prueba usan una instrumentación de contar bloques básicos. Obtienen que el framework más eficientes es Pin con un *slowdown* de 2,3x, es decir 2,3 veces más lento que la aplicación sin instrumentar; mientras que DynamoRIO y Valgrind tiene un *slowdown* de 4,9x y 7,5x respectivamente.

Fabrice Bellard en [Bel05] hace un experimento para ver el *slowdown* entre Qemu, Valgrind y Bochs. El benchmark que se usa es BYTEmark. En los experimentos que realiza Qemu tiene un *slowdown* de 4x en operaciones de cálculo entero y de 10x en cálculo real, comparado con la ejecución nativa. Y comparando Qemu con Bochs y Valgrind es siempre la opción más rápida, teniendo Bochs un *slowdown* de 30x y Valgrind de 1,2x.

Ruiz-Alvarez et al. en [RAH08] entre varios experimentos que realizan, hacen pruebas de rendimiento entre Pin y DynamoRIO. Las pruebas las realizan en dos entornos dife-

rentes, un Pentium 4 para 32 bit con una caché de trazas, y un Intel Xeon Core 2 para 64 bit con una caché de instrucciones. Ambos con GNU/Linux kernel 2.6.9 (i686 para 32 bit, y x86_64 para 64 bit). El benchmark utilizado es SPEC CINT 2006. Aquí obtienen que el *slowdown* de media de DynamoRIO es 1,22x y el de Pin es 1,45x. Para realizar las mediciones utilizan los contadores de rendimiento hardware con `PAPI` y `perfex`. Lo que se pretende analizar es el rendimiento de la caché de instrucciones o trazas y otras estructuras de la microarquitectura.

En [WM08], Weaver et al. describen el uso de herramientas creadas por ellos con Valgrind, Qemu y la comparan con otra de Pin. Éstas son ejecutadas en 9 máquinas diferentes con arquitectura IA-32 con Linux. Se usan dos benchmark SPEC CINT 2000 y SPEC CINT 2006. Utilizan los contadores hardware de la CPU para obtener resultados mediante el interfaz `perfmon2`. Para la instrumentación usan la cuenta de bloques básicos. Como conclusiones obtienen que el rendimiento de sus herramientas es similar a otras ya existentes.

Siempre que se instrumenta existe una sobrecarga que hace que se ejecute más lento que de forma nativa, en [CKS+08], Chen et al. buscan métodos para hacer que se ejecute más rápido en instrumentación de grano fino, a nivel de instrucciones. Para esto se usa el benchmark CPU SPEC INT 2000, y las herramientas DBA Addrcheck, Memcheck, TaintCheck y Lockset. Primero son instrumentados con Pin para obtener los accesos a memoria y eventos relacionados con las direcciones. Después, se ejecuta el benchmark instrumentado con las cuatro herramientas y obtiene un *slowdown* medio de 3,2x para Addrcheck, 3,3x para TaintCheck, 4,2x para Lockset y finalmente 7,8x para Memcheck que ha sido programado con Valgrind.

Contribución. Con mi trabajo aporto un benchmark **específico** para la evaluación de frameworks de DBI, ya que lo realmente importante en rendimiento es poder comprobar cuánto tiempo más tarda una aplicación en ser ejecutada cuando es instrumentada. Además, se ofrecen dos métodos de instrumentación: por instrucciones, donde se instrumenta el 100 % de las instrucciones ejecutadas; y por bloques básicos, de forma que se consiga una sobrecarga media. Además muestra la cuenta tanto de bloques básicos como de instrucciones, que es un dato muy importante en la instrumentación. Como dato final también se muestra el consumo de memoria, que no es proporcionado por ningún otro benchmark.

Capítulo 5

Creación de benchmark

Este capítulo resume las alternativas estudiadas para confeccionar un benchmark adecuado para la evaluación de rendimiento entre los diferentes frameworks. Finalmente, se presentan las características y métodos usados en el benchmark creado.

5.1. Alternativas estudiadas

La primera alternativa era utilizar un benchmark ya existente, por lo que se empezó estudiando benchmarks comerciales. Los primeros estudiados fueron los de Futuremark [Fut10], como 3DMark y PCMark, y aunque éstos son de uso libre, tenían el problema de ser únicamente para Windows. Otro benchmark que se estudió fue SysMark 2012, que también es sólo para Windows, de pago y sus aplicaciones entre otras son de Adobe, Microsoft y Google. Para otros sistemas operativos también estaba TPC, pero sus benchmarks son de Procesamiento de Transacciones En Línea (*OnLine Transaction Processing*, OLTP) en los que principalmente se evalúa una base de datos. PARSEC [Bie11], era una muy buena opción porque es de uso libre y ofrece el código fuente de sus aplicaciones, pero es un benchmark centrado en paralelización pensado especialmente para máquinas multiprocesadoras.

Finalmente, el que mejor se acercaba para comprobar el rendimiento de las herramientas programadas era CPU2006 v1.2 de SPEC [Cor06], con versiones para Windows y Linux. En este benchmark tienen una máquina de referencia, una Sun Ultra Enterprise 2 de 1997, y el resultado obtenido en el benchmark es normalizado con respecto a esta máquina. Éste se podría definir como el primero de los problemas, ya que lo que se quiere comparar es un ejecutable consigo mismo instrumentado. El segundo problema es que este benchmark no es un producto gratuito. Estos dos problemas motivaron la creación de un benchmark propio que se presenta a continuación.

5.2. Definición de benchmark

Una definición de benchmark tiene que cumplir una serie de detalles [Cor06]. Habitualmente, los benchmarks realizan un conjunto de operaciones estrictamente definidas:

- una carga de trabajo, y devuelve algún tipo de resultados.
- una métrica, describiendo cómo tienen que ser realizadas las pruebas.

La carga de trabajo en este benchmark será siempre la misma por cada ejecutable, y está preparado para que devuelva siempre el mismo resultado. Lo único importante es que debe realizar uso intensivo de CPU.

Las métricas de los benchmarks normalmente miden:

- velocidad: cómo de rápido se ha realizado la carga de trabajo.
- *throughput*/rendimiento: cuántas unidades de carga de trabajo por unidad de tiempo se han completado.

En este benchmark, para generar información interesante, los datos que se esperan obtener están relacionados con la velocidad y no con el *throughput*. Lo importante es el tiempo que le cuesta de más ejecutar una aplicación. Además se estudiarán los requerimientos de memoria de ejecutar aplicaciones instrumentadas.

Para la base de este benchmark se utilizarán bastantes programas usados por CPU2006. Esto es posible ya que todo el código fuente de estas aplicaciones está disponible libremente. Además, se aprovechará una parte de la metodología, como repeticiones, intercalado y optimizaciones. En CPU2006, cada aplicación usada se ejecuta con diferentes argumentos y optimizaciones. Al igual que en ese benchmark, cada aplicación se ejecutará varias veces para obtener una media del tiempo de ejecución. Aunque en CPU2000 se ejecutaba de forma consecutiva cada prueba, a partir de CPU2006 se ejecutan las pruebas intercaladas, de una forma más real, que es otra parte de la metodología que se usará en este benchmark. Otras opciones más tomadas de CPU2006 son compilar con diferentes niveles de optimización los ejecutables.

Los resultados de la comparativa se centran en tiempo de ejecución, instrucciones ejecutadas y el uso de memoria. La medición se realizará varias veces por ejecutable, y con diferentes niveles de optimización. Primero se ejecutarán sin instrumentar para conocer el tiempo que le cuesta realizar una tarea. Después se instrumentará para conocer el tiempo de ejecución, uso de memoria y datos relacionados con la instrumentación.

Además, las instrumentaciones serán realizadas por diferentes frameworks, por lo que se podrá observar cuál realiza mejor la tarea de instrumentación, el rendimiento en función de las instrucciones o tiempo de ejecución y qué niveles de optimización pueden ser mejores para instrumentar.

Las aplicaciones seleccionadas para el benchmark son todas de uso intensivo de CPU, y se han dividido en cuatro grupos: cálculo entero, cálculo real, gran demanda de E/S y software de acceso a memoria. Las tablas 5.1-5.4 muestran un resumen de las aplicaciones seleccionadas. La primera columna indica el nombre más descriptivo de la aplicación, después la versión utilizada para el benchmark, el lenguaje de programación en el que están desarrolladas, y el tipo de categoría en que se engloban las aplicaciones. Finalmente, se indica si esa aplicación ha sido utilizada en algún otro benchmark.

En la Tabla 5.1 hay software en las que las opciones de cálculo son de números enteros. Se utilizan ficheros de entrada diferentes a los de SPEC para reducir el tiempo

Nombre	Versión	Lenguaje	Tipo	Origen
`bzip2`	1.0.6	C	Compresión	SPEC CINT 2006
`GNU go`	3.8	C	IA - Juegos	SPEC CINT 2006
`hmmer`	3.0	C	Genética	SPEC CINT 2006
`h264ref`	18.2	C	Compresión video	SPEC CINT 2006
`libquantum`	1.0.0	C	Física, computación cuántica	SPEC CINT 2006

Tabla 5.1: Aplicaciones de cálculo entero.

de ejecución, p.e., una ejecución de `bzip2` en este benchmark sin instrumentar dura 20 segundos, y en SPEC CINT 2006 son 848 segundos [NEC08] (42 veces menos).

Nombre	Versión	Lenguaje	Tipo	Origen
`namd`	2.8	C++	Biología, simulación de moléculas	SPEC CFP 2006
`povray`	3.0	C	Renderización	SPEC CFP 2006
`milc`	v6	C	Física, Cromodinámica cuántica	SPEC CFP 2006
`mlucas`	2.8x	C	Numérica, cálculo de números primos	SPEC CFP 2000
`linpack`	29.5.04	Fortran	Numérica, multiplicación de matrices	Linpack benchmark

Tabla 5.2: Aplicaciones de cálculo real.

Para la categoría de software de cálculo real se ha seleccionado software que se puede ver en la Tabla 5.2. Una parte del software es usado en SPEC CFP 2006 y CFP 2000, pero se usan ficheros de entrada diferentes para reducir el tiempo de ejecución. También se utiliza linpack, una librería de resolución de ecuaciones lineales.

Nombre	Versión	Lenguaje	Tipo	Origen
`whirlpool`	2ª Rev	C	Criptografía, Hash	Propio
`ripemd`	160	C	Criptografía, Hash	Propio
`aes`	1	C	Criptografía, cifrado	Propio
`ffmpeg`	0.10	C	Conversión de formatos video/audio	Phoronix Test Suite

Tabla 5.3: Aplicaciones con gran demanda de entrada/salida.

Para este benchmark se han añadido más categorías de las que aparecen en otros benchmarks, como p.e., aplicaciones que también tuvieran uso intensivo de CPU, pero no sólo que trabaje en memoria, sino accediendo a disco leyendo un fichero de entrada

y obteniendo uno de salida de igual tamaño o superior. Este software se puede ver en la Tabla 5.3. Estas aplicaciones realizan cálculo sobre enteros.

Nombre	Versión	Lenguaje	Tipo	Origen
`memtester`	4.0.5	C	Comprobación de memoria defectuosa	Propio

Tabla 5.4: Aplicaciones de acceso a memoria.

Finalmente, se ha añadido otra categoría más donde el software accediera al subsistema de memoria, se ve en la Tabla 5.4. `memtester` es una aplicación de uso intensivo de CPU en cálculo entero.

5.3. Descripción del benchmark

Por cada aplicación que ha sido seleccionada para el benchmark se van a ejecutar un grupo de pruebas. Este grupo de pruebas se forma con el ejecutable de la aplicación, pero con diferentes niveles de optimización (-O0, -O3), que después será ejecutado sin instrumentar e instrumentado con los diferentes frameworks de DBI. Además se ejecutará 3 veces cada prueba, pero no seguidas, sino alternadas. Esto se hace de forma similar a SPEC CPU 2006, como se ha comentado en la sección 5.2 no ejecuta consecutivamente la misma aplicación con las mismas opciones, sino que hay alternancia de ejecución. Otra diferencia que hay con SPEC CPU 2006 es el fichero de entrada que se utiliza por cada aplicación, debido al *slowdown* que aparece en la instrumentación, se usan ficheros que ofrezcan menos carga de proceso. Esta diferencia puede observarse en el apéndice D.16.

Primero serán ejecutados sin instrumentación y después instrumentados. Se esperan obtener los siguientes datos:

- tiempo de ejecución real y total en segundos
- tiempo de ejecución en modo *kernel* en segundos
- tiempo de ejecución en modo de usuario en segundos
- número de veces que el proceso ha sido paginado a disco de memoria
- porcentaje de uso de la cpu
- uso de memoria durante la ejecución
- número de instrucciones/bloques básicos ejecutados

Puede haber pruebas que serán repetidas o descartadas, en función de si ha sido utilizado un tiempo de ejecución excesivo o ha sido paginado muchas veces en comparación con las otras pruebas. Esto es así porque las aplicaciones están preparadas para que el resultado obtenido de ellas sea siempre el mismo, por lo que se espera que su ejecución dure aproximadamente lo mismo. Para esto se utilizarán intervalos de confianza.

5.4. Herramientas para el benchmark

En el benchmark se van a realizar dos herramientas con diferentes tipos de instrumentación:

- A nivel de instrucción
- A nivel de bloque básico

Con la primera herramienta, instrumentación a nivel de instrucción, se quiere conseguir el máximo *slowdown*. Esto es así porque en la parte de la instrumentación, donde se decide en qué punto se inserta la instrumentación, se instrumentan todas las instrucciones. De esta forma, por cada instrucción ejecutada, también se ejecutará el código añadido, que es incrementar un contador. Al finalizar la ejecución se devuelve el resultado del contador total de instrucciones ejecutadas. Esta herramienta se ha desarrollado con los tres frameworks de DBI: Pin, Valgrind y DynamoRIO.

Con la segunda herramienta, instrumentación a nivel de bloque básico, se quiere conseguir una sobrecarga media. En la mayoría de artículos que evalúan frameworks de DBI como [PA06], [Sof07] y [WM08] utilizan este método. En la parte de instrumentación, se instrumentan todos los bloques básicos. Por cada bloque básico ejecutado, se ejecutará el código añadido, que es incrementar un contador. Al finalizar la ejecución se muestra la suma del número total de bloques básicos ejecutados. Esta herramienta se ha desarrollado también con los tres frameworks de DBI: Pin, Valgrind y DynamoRIO.

5.5. Mediciones en el benchmark

En esta sección se van a describir los métodos que utiliza el benchmark para medir el tiempo de ejecución y el consumo de memoria de una aplicación instrumentada y sin instrumentar.

5.5.1. Tiempo

Todas las pruebas se hacen con el comando `/usr/bin/time`, diferente de la variante integrada en la shell `time`, parte del intérprete de comandos. Esta es una forma básica de medir el tiempo que tarda en ejecutarse un programa. Ofrece el tiempo real desde que inicia hasta que acaba y además el tiempo que realmente se está ejecutando en la CPU, distinguiendo entre el código de usuario y sus llamadas al sistema.

La mayoría de la información ofrecida por `/usr/bin/time` está derivada de la llamada al sistema `wait3` o `wait4`, con lo que los datos obtenidos serán tan buenos como los que pueda ofrecer esta llamada. En los sistemas que no esté disponible se usa la llamada `times` que ofrece mucha menos información. En el sistema donde se ejecutará el benchmark la llamada al sistema usada es `wait4`.

5.5.2. Memoria

Para medir el consumo de memoria, el benchmark ejecuta un proceso en *background* después de lanzar la aplicación a evaluar. Este proceso se encarga de revisar en `/proc/<pid de la aplicación>/status` el consumo en el campo `VmPeak` que contiene el pico de memoria utilizada.

Para definir el intervalo en el que se consulta el consumo se usaron valores inferiores a 1 segundo, y se fue aumentando una vez que se pudo comprobar que en las aplicaciones no se incrementaba el consumo de memoria en el último intervalo por encima del valor de pico.

Capítulo 6

Experimentos

En este capítulo se resumen los experimentos realizados con el benchmark, comenzando por la definición del entorno de pruebas y mostrando los resultados con un análisis crítico.

Se ha utilizado un equipo con procesador Intel Core 2 Duo T7300 a 2 Ghz, 2 GiB de memoria RAM, S.O. Fedora Core 14 con *kernel* 2.6.35.14-106.fc14.i686.PAE. Se le ha deshabilitado al equipo uno de los *cores* para evitar que ciertas aplicaciones lanzaran más de un proceso a la vez, y que el tiempo de CPU se encontrara en valores cercanos al 190 %. En las Tablas 6.1 y 6.2 se resume el hardware y software, respectivamente, utilizado para la realización del experimento.

Nombre CPU	Intel®Core™2 Duo CPU T7300
Características	2.00 GHz, 667 MHz bus
CPU(s)	2 cores, 1 desactivado.
Caché primer nivel	32 KiB I + 32 KiB D por core
Caché segundo nivel	4 MiB I+D
Memoria RAM	2 GiB (2x1 GiB DDR2 SODIMM 667 Mhz)
Disco Duro	120 GB HITACHI HTS54161

Tabla 6.1: Hardware utilizado en las pruebas.

Sistema Operativo	Fedora Core 14 32bit
Compilador C	gcc (GCC) 4.5.1 20100924 (Red Hat 4.5.1-4)
Compilador Fortran	GNU Fortran 4.5.1
Sistema de ficheros	ext4
Nivel sistema	Run level 3 (multiusuario)

Tabla 6.2: Software utilizado en las pruebas.

Las versiones de los frameworks de DBI que se han utilizado han sido Pin v.2.10,

Valgrind v.3.7.0 y DynamoRIO v.2.2.0-2.

6.1. Entorno de pruebas

En los experimentos se quiere que el resto de procesos que están funcionando en la máquina afecten lo mínimo posible, por lo que para que se tengan los mínimos procesos funcionando se iniciará la máquina en nivel 3 (multiusuario con red).

Una vez reiniciada la máquina en ese nivel no se tendrá ningún proceso relacionado con el gestor de escritorio o salvapantallas que pueda influir durante la ejecución del benchmark. En este momento y conectado remotamente por `ssh` se inicia el benchmark.

Mientras se ejecuta, se puede ver por el terminal el programa del benchmark que se está ejecutando, si lo hace sin instrumentar o con cúal está instrumentado, el número de repetición de la prueba y si es el benchmark de instrucciones o de bloques básicos. El benchmark genera un fichero de log y un fichero de valores delimitados por comas (csv) en el que se puede encontrar el tiempo real de ejecución, el tiempo en modo *kernel*, el tiempo en modo de usuario, los fallos de página, el porcentaje de CPU usado y la línea de comando. Como datos adicionales se encuentran el consumo de memoria por proceso y el número de instrucciones o de bloques básicos ejecutados por aplicación instrumentada en el log.

6.2. Resultados

Se ha visto previamente en [LCM+05] y [NS07] que la instrumentación provoca una ralentización de la ejecución del programa instrumentado. Aquí, se va a observar la eficiencia de cada framework de DBI intentando inferir cuál es el más adecuado según sea el tipo de aplicación que se va a instrumentar.

La Figura 6.1 muestra los resultados de tiempo de ejecución para la aplicación `h264ref`, una aplicación de conversión de audio y vídeo. Como se ha dicho previamente, una de las características de la instrumentación es que provoca una ralentización de la ejecución, y aquí se comprueba. Es la prueba más lenta de todas una vez instrumentada. Bajo Valgrind llega a durar 2615 segundos en la versión sin optimizar del ejecutable, y 918 segundos en la optimizada, cuando sin instrumentar son 175 segundos sin optimizar y 52 segundos optimizada. Aunque en relación al *slowdown* (la relación entre el tiempo de ejecución instrumentado y sin instrumentar), ésta no es la peor aplicación de todas: bajo Valgrind se obtiene 11,2x sin optimizar, y 17,43x optimizada.

En la Figura 6.2 se muestra el *slowdown* de la aplicación `ffmpeg` instrumentada por los tres frameworks en ambas optimizaciones en la instrumentación por instrucciones. Esta es la aplicación que mayor *slowdown* presenta. Como en la aplicación anterior, `h264ref`, el mayor *slowdown* se tiene bajo la instrumentación de Valgrind, que en sus versiones sin optimizar y optimizada tiene una ratio de 34,6x y 35,38x respectivamente.

No sólo lo más importante es que tarde más tiempo, sino el número de veces que se puede llegar a ejecutar más lenta una aplicación. Revisando el *slowdown* de todas las

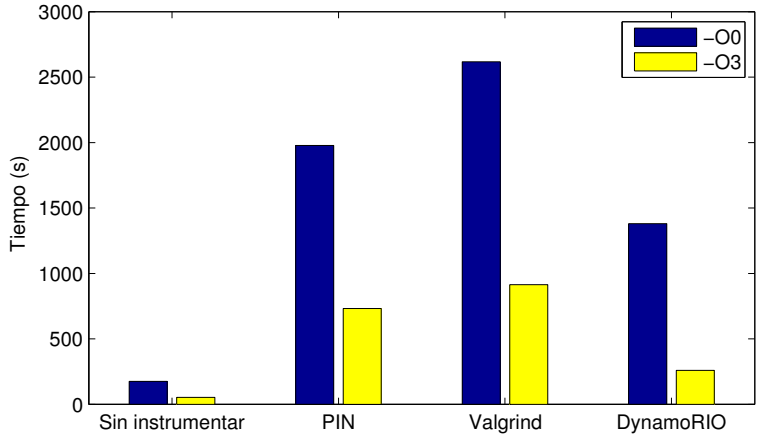

Figura 6.1: Tiempo de ejecución de la aplicación `h264ref` con instrumentación por instrucciones y optimizaciones.

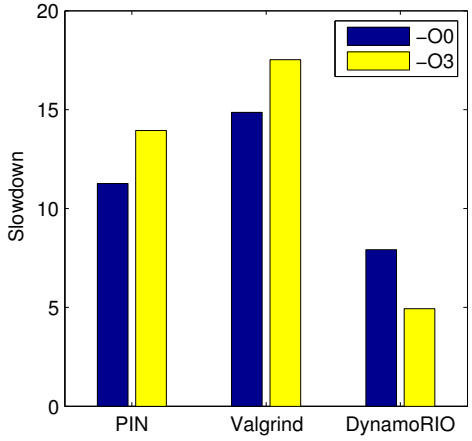

Figura 6.2: *Slowdown* en el benchmark de `ffmpeg` con instrumentación por instrucciones y optimizaciones.

aplicaciones individualmente, y el tiempo de ejecución de ellas (p.e., Figura 6.1) se puede comprobar que la opción más lenta es Valgrind.

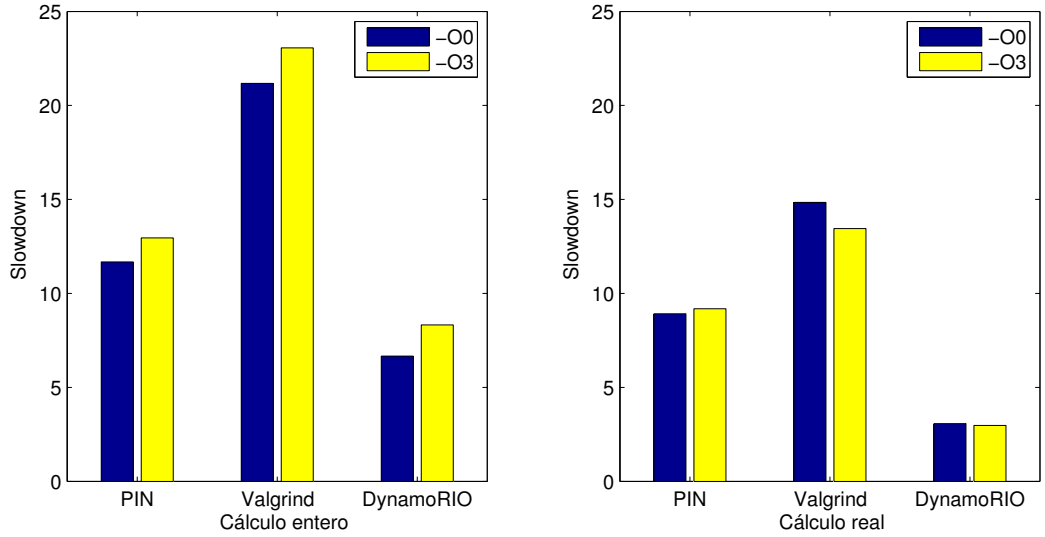

Figura 6.3: *Slowdown* medio en el benchmark usando instrumentación por instrucciones.

En la Figura 6.3 se puede ver el *slowdown* medio por framework de DBI, dividido entre las aplicaciones de cálculo entero y las de cálculo real. Individualmente Valgrind es siempre la opción más lenta y como se puede comprobar en esta figura también lo es en las medias. Otra conclusión que podemos obtener es que, independientemente del framework que se use, todos son más eficientes que Valgrind en cálculo real. También se comprueba en la misma tabla que las aplicaciones serán más rápidamente instrumentadas con DynamoRIO, tanto en cálculo entero, como real y con ambas optimizaciones usadas en la compilación. Además, se observa que la optimización no mejora el *slowdown* en general, salvo en Valgrind y DynamoRIO en cálculo real.

El consumo medio de memoria de las aplicaciones instrumentadas separadas por framework se puede ver en la Figura 6.4, donde se puede comprobar que el framework que menor consumo ofrece es Pin, tanto en cálculo real como en entero. Prácticamente no hay diferencia entre el tipo de cálculo que se haga ni en si se ha optimizado el ejecutable o no.

Otro dato importante de la Figura 6.4 es que el consumo de memoria de la aplicación instrumentada no es proporcional a la memoria consumida sin instrumentar, sino que parece un incremento lineal. En la Tabla 6.3 se puede ver la media del consumo de la herramienta DBA restándole el consumo de la aplicación instrumentada y la desviación media. No deja lugar a dudas que el consumo de DynamoRIO, aunque sea el más elevado de todos, es prácticamente fijo independientemente de la aplicación instrumentada por lo que será un dato muy importante a tener en cuenta para casos en los que se puedan tener restricciones de memoria, como p.e., un sistema empotrado. Sin embargo la opción de Pin será la más eficiente respecto al consumo de memoria y Valgrind, aunque no sea

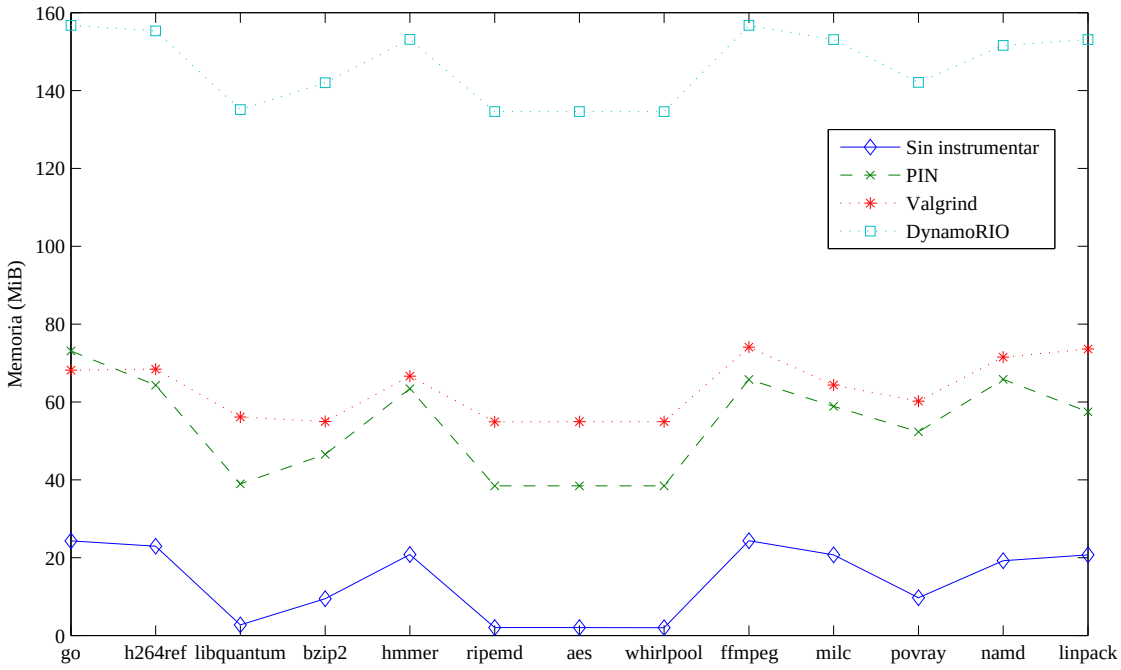

Figura 6.4: Consumo medio de memoria de las aplicaciones.

Framework	Media consumo (kiB)	Desviación media (kiB)
Pin -O0	39885	3113
Valgrind -O0	53033	3736
DynamoRIO -O0	132478	73
Pin -O3	40871	4006
Valgrind -O3	52789	3535
DynamoRIO -O3	132475	72

Tabla 6.3: Consumo medio de memoria por Framework.

la mejor también será una buena opción debido a que el consumo es inferior a la mitad de DynamoRIO.

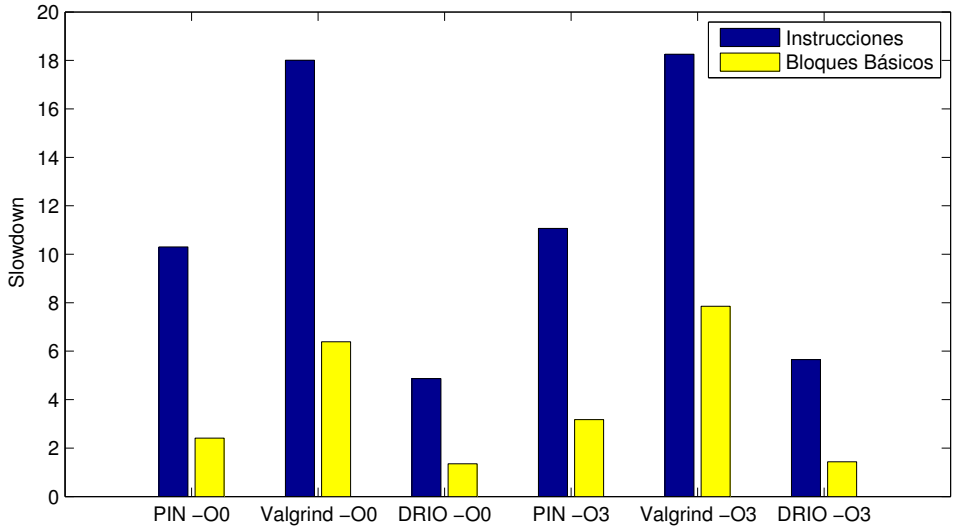

Figura 6.5: *Slowdown* en instrumentación por instrucciones y por bloques básicos por frameworks y optimizaciones.

En la Figura 6.5 se comparan los *slowdown* que aparecen en los tres frameworks de DBI, para todas las optimizaciones y para las instrumentaciones de instrucciones y de bloques básicos. Se puede comprobar que una instrumentación más ligera produce menos *slowdown* en cualquier tipo de aplicación.

A partir de lo visto en la Figura 6.5 se ha decidido hacer la Tabla 6.4 con los *slowdowns* relativos entre instrumentación por instrucciones y bloques básicos. Se muestra que el menor *slowdown* aparece en Valgrind en ambas optimizaciones, y los peores resultados se obtienen en Pin con optimización -O0 y en DynamoRIO con optimización -O3.

	Pin -O0	**Valgrind -O0**	**DynamoRIO -O0**	**Pin -O3**	**Valgrind -O3**	**DynamoRIO -O3**
Slowdown	4.29	2.81	3.76	3.54	2.29	4.03

Tabla 6.4: *Slowdown* relativo entre instrucciones y bloques básicos.

Y lo más importante respecto a rendimiento que también se puede comprobar en la Figura 6.5 es que la opción más eficiente al instrumentar siempre es DynamoRIO.

Capítulo 7

Conclusiones y trabajo futuro

Este capítulo presenta algunas conclusiones obtenidas de la elaboración de este PFC. Además, plantea el trabajo futuro.

La instrumentación dinámica de ejecutables ofrece soluciones necesarias y muy interesantes pero ofrece un rendimiento muy bajo, en comparación con la ejecución nativa. Cada framework intenta resolver este problema para venderse a sí mismo como la mejor opción de todas.

Este proyecto ha buscado unos criterios para la selección de diferentes frameworks de DBI y ha hecho una comparativa a nivel de impacto en rendimiento. Los tres frameworks que se han seleccionado han sido Pin, Valgrind y DynamoRIO. Para evaluar el rendimiento, se han seleccionado un conjunto de 15 aplicaciones, 10 de cálculo entero y 5 de cálculo real, para la creación de un benchmark. Para este benchmark se han programado dos herramientas con diferente granularidad de instrumentación: a nivel de instrucción y a nivel de bloque básico. Se han hecho tres versiones de cada herramienta programadas con cada uno de los frameworks de DBI.

Entre los datos más importantes, se ha evaluado el tiempo de ejecución de la aplicación en dos versiones diferentes (sin optimizar, -O0, y optimizada, -O3), y para cada una de estas dos versiones ejecutándose de forma nativa e instrumentada bajo los tres frameworks con las herramientas programadas.

Según los resultados obtenidos, la peor opción en rendimiento en todos los casos es Valgrind, y además con un *slowdown*, la relación entre el tiempo de ejecución instrumentado y sin instrumentar, muy superior a los otros dos frameworks de DBI. Según se ha podido comprobar en el trabajo relacionado desde las primeras versiones de Valgrind, su rendimiento es bastante malo y no lo están mejorando.

Sobre las optimizaciones, se ha comprobado que sin optimizar el que mejor rendimiento tiene es DynamoRIO, y una vez que se optimiza sigue siendo la mejor opción. También independientemente del tipo de cálculo principal realizado por el ejecutable, el que mejor rendimiento ofrece es DynamoRIO.

Sin embargo, un dato importante que no aparece en los trabajos relacionados es el consumo de memoria de la aplicación una vez instrumentada. Indirectamente está relacionado con el rendimiento, porque la falta de memoria física e incremento de la paginación

a disco puede suponer un detrimento del rendimiento. En la plataforma en la que se realiza el estudio, x86 bajo GNU/Linux, no es únicamente de servidores, ni equipos de escritorio, sino que engloba a multitud de equipos empotrados, que suelen incorporar una cantidad de memoria pequeña. En los resultados se ha visto que el framework de DBI más eficiente, DynamoRIO, también es el que más memoria consume, pero según la desviación media vista en la Tabla 6.3, la memoria que consume de más se podría llegar a estimar. El framework de DBI que menos memoria consume y es la opción más recomendada para un equipo con poca memoria es Pin.

Otra de las conclusiones es que no se instrumentan la misma cantidad de instrucciones en los tres frameworks de DBI seleccionados. Se ha estudiado el funcionamiento paso a paso de la carga de un ejecutable por los frameworks de DBI en los problemas encontrados. Es importante conocerlo para comprobar la correcta funcionalidad de las herramientas programadas.

A continuación se exponen algunos trabajos futuros que completarían o ampliarían el trabajo desarrollado en este PFC:

- **Uso de Linux Performance-Monitoring Driver** para la monitorización de los ejecutables del benchmark, como en [RAH08] y [WM08]. Utilizando este *driver* del *kernel* se podría obtener información del número de instrucciones ejecutadas por aplicación sin necesidad de instrumentarlas. De esta forma se podría comparar el número de instrucciones ejecutadas por framework de DBI con las de una aplicación ejecutada de forma nativa.
- **Uso de Intel® VTune™ Amplifier XE** [Int12] para la medida de tiempo e instrucciones. Utilizando esta aplicación, que también modifica el *kernel*, se obtendría de manera fácil el tiempo de ejecución de cada aplicación, las instrucciones ejecutadas y el consumo de memoria, tanto para aplicaciones nativas como instrumentadas.
- **Ampliar el estudio a más frameworks de DBI**. Una vez que se ha definido un método para comparar el rendimiento de frameworks de DBI, se puede ampliar a otros para tener un estudio más completo.
- **Comparar en más detalle las APIs y las herramientas que llevan los frameworks de DBI**. Como dicen Nethercote et al. en [NS07] sobre Valgrind, a pesar de que es el framework con menos rendimiento, no sólo hay que fijarse en el rendimiento sino en las capacidades de la instrumentación y las herramientas que han sido programadas con ellos.
- **Ampliar las aplicaciones del benchmark**. Para comprobar el rendimiento de los frameworks de DBI en un mayor número de aplicaciones.

En un ámbito más personal, mi valoración de este PFC es muy positiva, puesto que además de alcanzarse con éxito los objetivos del proyecto, me ha permitido conseguir una enriquecedora experiencia sobre la instrumentación y frameworks de DBI, además de mejorar mi conocimiento sobre la depuración y el formato de los ficheros ejecutables.

Bibliografía

[BDCW91] Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical report, 1991.

[Bed90] Robert Bedichek. Some efficient architectures simulation techniques. In *Winter 1990 USENIX Conference*, 1990.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[Bru04] Derek Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[BZA12] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM.

[CKS+08] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. *SIGARCH Comput. Archit. News*, 36(3):377–388, jun 2008.

[Cor06] Standard Performance Evaluation Corporation. SPECCPU 2006 benchmarks. http://www.spec.org, 2006.

[Fut10] Futuremark Corporation Benchmarks. PCMark 7, 3DMark 11. http://www.futuremark.com/benchmarks/, 2010.

[GH93] Stephen R. Goldschmidt and John L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '93, pages 146–157, New York, NY, USA, 1993. ACM.

[Int86] Intel. *80386 programmer's reference manual*. Intel Corporation, 1986.

[Int12] Intel. Intel® VTune™ Amplifier XE , 2012.

[Jon08] M. Tim Jones. Anatomy of linux dynamic libraries. 2008.

[LB94] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Softw. Pract. Exper.*, 24(2):197–218, February 1994.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[NEC08] NEC Corporation. SPEC® CINT2006 Result - Intel Core 2 Duo T7400. http://www.spec.org/cpu2006/results/res2008q2/cpu2006-20080316-03692.html, 2008.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[PA06] G.-R. Uh; R. Cohn; B. Yadavalli; R. Peri and R. Ayyagari. Analyzing dynamic binary instrumentation overhead. In *Workshop on Binary Instrumentation and Applications*, WBIA, 2006.

[RAH08] Arkaitz Ruiz-Alvarez and Kim M. Hazelwood. Evaluating the impact of dynamic binary translation systems on hardware cache performance. In *IISWC*, pages 131–140, 2008.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. *SIGPLAN Not.*, 29(6):196–205, June 1994.

[SG92] Inc. Silicon Graphics. *MIPS Assembly Language Programmer's Guide*. Silicon Graphics, Inc., 1992.

[Sof07] A. Guha; J.D. Hiser; Naveen Kumar; J. Yang; M. Zhao; S. Zhou; B.R. Childers; J.W. Davidson; K. Hazelwood; M.L. Soffa. Virtual execution environments: Support and tools. In *International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Dept. of Comput. Sci., Virginia Univ., Charlottesville, VA, March 2007.

[SSNB06] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 175–185, New York, NY, USA, 2006. ACM.

[Wal91] David W. Wall. Systems for late code modification. In *WRL Research Report 91/5*, pages 275–293. Springer-Verlag, 1991.

[WM08] Vincent M. Weaver and Sally A. McKee. Using dynamic binary instrumentation to generate multi-platform simpoints: methodology and accuracy. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers*, HiPEAC'08, pages 305–319, Berlin, Heidelberg, 2008. Springer-Verlag.

Apéndice A

Fases de Desarrollo

A.1. Diagrama de *Gantt*

En este apéndice se describe la distribución temporal de cada una de las etapas de este proyecto. La figura A.1 muestra el Diagrama de Gantt donde se reflejan las diferentes tareas realizadas y la figura A.2 el tiempo dedicado a su desarrollo, que ha sido de 410 horas y se puede ver detallado en la Tabla A.1.

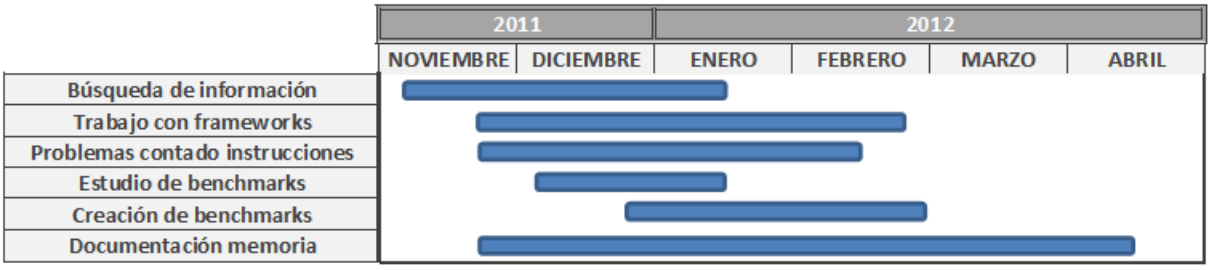

Figura A.1: Diagrama de gantt.

Primero se realizó una búsqueda de información sobre DBI, ya que no se poseían conocimientos previos. Se buscó información sobre la instrumentación y sus diferentes tipos, análisis estático, dinámico, DBI, frameworks de DBI y trabajos previos relacionados con rendimiento para frameworks de DBI.

Una vez que se encontró información suficiente, se procedió a la selección de frameworks de DBI, donde el procedimiento que se realizaba para cada uno era: descarga del código fuente del framework, compilación, instalación y prueba de funcionamiento. Una vez que se llegaba a este punto, se estudiaba el API y se procedía a programar una herramienta básica.

Se hizo un estudio sobre los benchmarks existentes y de cómo se podían llegar a usar en el desarrollo de este PFC. Se comprobó que no había ninguno que encajara y se desarrolló uno nuevo.

Para la creación de este benchmark primero se buscaron aplicaciones y se desarrollaron metodologías para la ejecución de las aplicaciones de prueba. Se buscaron diferentes métodos para medir el tiempo de ejecución de las aplicaciones y el consumo de memoria de estas. Una vez finalizado este proceso se realizaron las pruebas con el benchmark para obtener los resultados para este PFC.

La memoria del proyecto se ha ido realizando poco a poco, desde el principio del proyecto. Se fue completando al final con los resultados y conclusiones de los experimientos realizados.

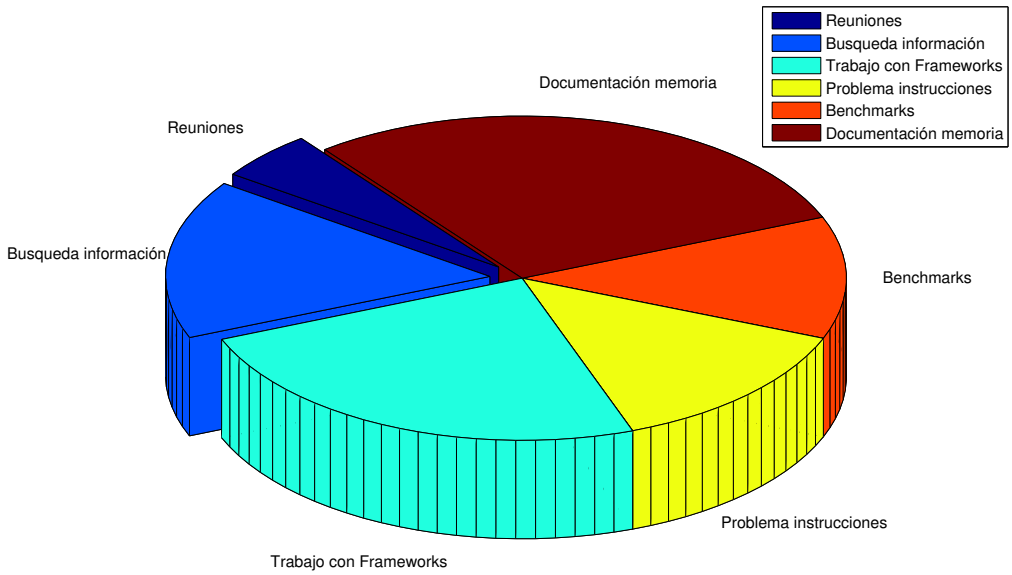

Figura A.2: Horas dedicadas.

Tareas	Horas
Reuniones	20
Búsqueda de información	75
Trabajo con frameworks	95
Problema contado de instrucciones	55
Estudio y creación de benchmark	60
Documentación memoria	105
Total	**410**

Tabla A.1: Horas dedicadas.

Apéndice B

Problemas encontrados

En este capítulo se resumen las dificultades más destacables que han ido apareciendo durante la realización de este proyecto.

B.1. Cuenta de instrucciones

Al principio del estudio se hicieron pruebas para verificar el funcionamiento de los frameworks de DBI. La primera herramienta se dedicaba a contar instrucciones ejecutadas en Pin, Valgrind y DinamoRIO. Se esperaba que el resultado de las tres instrumentaciones fuera exactamente el mismo número de instrucciones. Para esto se creó un programa en C que calculaba el factorial de un número. Los resultados fueron que, para instrumentar el comando `$ factorial >/dev/null`, se obtenían 99094 instrucciones en Pin, 120014 en Valgrind y 10150 en DynamoRIO.

Debido a esto, se hicieron pruebas con más ejecutables para ver si existía algún tipo de correlación y los resultados son los que aparecen en la Tabla B.1. Son datos que no tienen ninguna relación entre ellos, por lo que se procedió a estudiar paso a paso el funcionamiento de las herramientas programadas.

	Pin	**Valgrind**	**DynamoRIO**
`xfsinfo`	269699	298295	25581
`ls`	443104	474249	103596
`xeyes`	875730	920633	223912
`cat`	219189	242386	63354

Tabla B.1: Instrucciones contadas por framework de DBI.

El primer paso fue sacar las primeras instrucciones de la aplicación instrumentada, junto con su traza, para ver que instrumenta y cuenta la herramienta, para después compararlas con las que aparecen en un *debugger*. Para esto, primero se desensambló el ejecutable con `$ objdump -d ./factorial` y así ver cuáles son las instrucciones en

ensamblador y sus direcciones. Aunque en Pin y Valgrind las primeras instrucciones coincidían, en DynamoRIO no era así.

Buscando más información, se obtuvo que DynamoRIO no soporta *early injection* (no instrumenta desde la primera instrucción). Además DynamoRIO necesita que el ejecutable esté enlazado dinámicamente, y cambia el proceso de carga de la aplicación: en vez de utilizar el *loader* del sistema utiliza uno privado y carga sus librerías después de las de la aplicación. En Linux comienza la instrumentación cuando DynamoRIO vuelve de la inicialización de las librerías dinámicas (exactamente cuando deja la dirección de retorno en la pila). Si el ejecutable no está enlazado dinámicamente, no puede cargar sus librerías y no puede instrumentar. Debido a esta limitación, en aplicaciones pequeñas, el número de instrucciones que instrumenta es de un orden de magnitud menor. En la versión para Windows sí soporta *early injection*.

El segundo problema que apareció fue que en la traza de direcciones las instrucciones eran las mismas, pero cada vez que se ejecutaba, las direcciones eran diferentes. Esto es debido a que a partir de la versión 2.6 del *kernel* de Linux tiene activado por defecto y por motivos de seguridad *Virtual Address Space Randomization*. Se soluciona deshabilitándolo con el siguiente comando:

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

En el momento que las direcciones eran las mismas, se estudió el proceso de carga de un ejecutable bajo GNU/Linux. Cuando se invoca un ejecutable, el *kernel* lo carga en memoria virtual del espacio de usuario, después en la sección `.interp` busca cúal es el cargador dinámico a utilizar (p.e., `/lib/ld-linux.so.2`) y lo inicializa, cargando después todas las bibliotecas dependientes (p.e., `libc`), resuelve los símbolos, y finalmente transfiere la ejecución al ejecutable original para comenzar su ejecución [Jon08]. Debido a esto, cada vez se instrumenta un ejecutable con la opción para dar la traza de las instrucciones, en vez de comenzar la ejecución en la dirección de inicio (`start address`) que se obtiene con `objdump` comienza en otra previa.

```
$ objdump -f factorial
prx2: file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048310
```

Para poder hacer una comparación más sencilla, se decide compilar el programa de prueba sin enlazar dinámicamente. Además, para tener más información de lo que se está ejecutando antes del inicio se compila con las librerías estándar de C estáticas con información de depuración (paquete `glibc-debuginfo-2.13-2`). Los parámetros a utilizar en la compilación son:

- Con información de depuración del programa (`-g`)
- Uso de glibc con información de depuración(`-L/usr/lib/debug/usr/lib`)

- En estático (`-static`) para que no incluya librerías externas.

Una vez realizada esta operación sobre el ejecutable, cuando se instrumenta, las instrucciones ejecutadas bajo Pin son 14223 y bajo Valgrind 14572. Hay 349 instrucciones de diferencia que se va a averiguar qué hacen y por qué se ejecutan en uno y no en el otro.

Siguiendo la traza de instrucciones, se comprueba que la diferencia de éstas corresponde a que el valor de los vectores de entorno (`envp`) y de los vectores auxiliares (`auxv`) son diferentes en ambas ejecuciones. En `envp` aparecen variables de entorno diferentes que son añadidas por los scripts de inicio de Pin y Valgrind, cada variable de entorno es evaluada individualmente en el proceso de carga del ejecutable, por lo que afecta el tener más o menos variables de entorno a las instrucciones ejecutadas. En `auxv` se encuentra la diferencia: en Pin se pasan valores para `AT_SYSINFO` y `AT_SYSINFO_EHDR`, pero para Valgrind no, por lo que primeramente afecta al procesamiento del vector ya que son de tamaño diferentes y después en esos valores está el puntero al *Virtual Dynamically-linked Shared Objects* (VDSO), que es la forma actual de hacer llamadas al sistema. En este método de llamadas al sistema se utilizan las instrucciones `sysenter/sysexit` y previamente se utilizaban interrupciones software con `int 0x80`.

La **primera conclusión** que se obtiene es que con un ejecutable enlazado dinámicamente, el proceso que realiza el cargador dinámico, incluyendo la carga de librerías, se realiza de forma diferente entre Pin y Valgrind. La **segunda conclusión** es que las variables de entorno afectan directamente al número de instrucciones ejecutadas. Y la **tercera conclusión** y más importante de todas es que las llamadas al sistema son realizadas de forma diferente entre Pin y Valgrind. Por este motivo, nunca se ejecutará el mismo número de instrucciones para el mismo programa instrumentado por diferentes frameworks.

La única forma que se ha encontrado para que se cuenten el mismo número de instrucciones entre distintos frameworks ha sido programándolo directamente en ensamblador. Compilando un programa que sólo tenía 12 instrucciones, el funcionamiento en Pin y Valgrind era correcto y contaban sin ningún problema. Sin embargo, DynamoRIO no podía instrumentar este ejecutable en ensamblador porque no era un ejecutable enlazado dinámicamente.

En aplicaciones con un número alto de instrucciones puede llegar a haber muy poca diferencia, como se ha comprobado en este PFC y se puede ver en la Tabla B.2, que muestra el número de instrucciones ejecutadas para las aplicaciones `whirlpool` y `memtester` optimizadas.

	Pin	**Valgrind**	**DynamoRIO**
`whirlpool -O3`	74222379297	74222385145	74222287028
`memtester -O3`	241201530296	241201536828	241201371000

Tabla B.2: Instrucciones contadas para las aplicaciones whirlpool y memtester

B.2. Fallo en ejecución

Durante la ejecución del benchmark ha aparecido un fallo con la aplicación `mlucas`. Como de todas las aplicaciones que hay en el benchmark se dispone del código fuente, a partir de los mensajes de salida del fallo y de todo el código que se encontraba a su alrededor, se ha podido hacer un pequeño programa de prueba capaz de repetir el fallo.

```
#include <stdio.h>

int main(int argc, char **argv)
{
 double RND_A,RND_B,prueba;
 prueba= 5.4321;
 RND_A = 3.0*0x4000000*0x2000000*0x800;
 RND_B =12.0*0x2000000*0x1000000*0x800;
 printf("INFO: using 80-bit-double form of rounding constant\n");
 printf("prueba: %20.15f RND_A: %20.15f RND_B: %20.15f\n",prueba,
                          RND_A, RND_B);
 if( ((prueba+RND_A)-RND_B) != 5.0 )
 {
  printf("INFO:prueba=%20.15f, rnd(prueba)=%20.15f\n",prueba,
                          (prueba+RND_A)-RND_B);
  printf("ERROR 30 in util.c\n"); return(1);
 }
return 0;
}
```

Según el programa original para hacer el redondeo de una variable usa la función `rnd()`. Lo que hace esta función es, a partir de un número real sumar y restar dos constantes de redondeo, suma el valor `RND_A` y resta el valor `RND_B`, ambos valores son iguales pero calculados de forma diferente. Y el comportamiento esperado es que el resultado final sea la parte entera del número real a redondear. El número que se va a redondear es 5.4321. El programa se comporta de esta manera sin instrumentar y bajo la instrumentación de Pin y DynamoRIO.

```
$ pruebafallo
INFO: using 80-bit-double form of rounding constant
prueba:    5.432100000000000 RND_A: 13835058055282163712.000000000000000
RND_B: 13835058055282163712.000000000000000
```

Sin embargo, bajo la ejecución en Valgrind el resultado no es el esperado:

```
$ valgrind pruebafallo
==2503== Memcheck, a memory error detector
==2503== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
```

```
==2503== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==2503== Command: ./pruebafallo
==2503==
INFO: using 80-bit-double form of rounding constant
prueba:    5.432100000000000 RND_A: 13835058055282163712.000000000000000
RND_B: 13835058055282163712.000000000000000
INFO: prueba =    5.432100000000000,  rnd(prueba) =    0.000000000000000
ERROR 30 in util.c
```

La condición del IF ya no falla, no se ha hecho bien el redondeo. Por este motivo falla `mlucas` en la prueba inicial de comprobación de funcionamiento y ya no sigue ejecutándose. En el programa original los dos valores reales con los que se hacía esta comprobación eran $\frac{1}{\sqrt{2}}$ y $2 \times \pi$, cuyas partes enteras son 1 y 6 respectivamente.

Al ir a abrir un bug sobre este fallo, éste ya estaba abierto, y reportado múltiples veces por diferentes usuarios. El problema es que Valgrind no puede trabajar con números de 80 bits en las arquitecturas x32 y x64.

El bug se puede ver en `https://bugs.kde.org/show_bug.cgi?id=197915`

Apéndice C

Aplicaciones usadas en el benchmark

En este apéndice se muestra todo el software que ha sido utilizado para la creación del benchmark, junto con detalles de cada uno, los ficheros de entrada y de salida generados.

C.1. bzip2

Versión: 1.0.6
Categoría: Compresión de datos
Tipo de cálculo realizado: Entero
Descripción: Es un compresor de datos de alta calidad, libre de patentes y libremente disponible. Típicamente comprime ficheros entre un 10 % y un 15 % más que con otros tipos de compresores, y es alrededor de dos veces más rápido en compresión y seis veces más rápido descomprimiendo.
Entrada: Se comprimirán de una única ejecución los siguientes ficheros: texto.txt, con texto en plano; libre1.odt y libre2.ods, ficheros de libreoffice; audio1.mp3 y audio1.wav, ficheros de audio; video1.mpg y video1.mkv, ficheros de video en SD y HD; comprimido1.gz y comprimido1bz2, ficheros comprimidos con gzip y bzip2.
Salida: Como resultado en vez de guardarse en uno o varios ficheros se ha redirigido a la salida estándar `stdout` y esta a `\dev\null` para que no escribiera en disco y obtener mayor eficiencia.
Página web: bzip.org
Autor: Julian Seward
Usada en: SPEC CINT 2006

C.2. GNU go

Versión: 3.8
Categoría: Inteligencia Artificial - Juegos
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Programa que analiza jugadas del juego Go.
Entrada: Fichero de jugadas del tipo *"SmartGo Format"* (.sgf) con el fichero `game001.sgf` que contiene una partida jugada en un tablero de 9x9, por Miyamoto Naoki y Go Seigen en 1968 .
Salida: Descripción en texto plano de las jugadas realizadas y el porcentaje con que la aplicación hubiera realizado esa jugada.
Página web: `http://www.gnu.org/software/gnugo/gnugo.html`
Autor: Man Lung Li et Al.
Usada en: SPEC CINT 2006

C.3. hmmer

Versión: 3.0
Categoría: Genética
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Busca una proteína en una base de datos. Usa el modelo de perfiles de Markov ocultos (HMMs) como modelos estadísticos, que es usado en biología computacional para buscar patrones en secuencias de ADN.
Entrada: Se indica la proteína a buscar, goblins50 de 143 moléculas en el fichero `globin.hmm` y la base de datos donde tiene que hacerlo `sprot41.dat`
Salida: Genera cuatro ficheros donde se indican las coincidencias encontradas.
Página web: `http://hmmer.janelia.org/`
Autor: Sean Eddy et Al.
Usada en: SPEC CINT 2006

C.4. libquantum

Versión: 1.0.0
Categoría: Física, computación cuántica
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Es una librería en C para simulación de mecánica cuántica, especializada en computación cuántica. Comenzó por ser un simulador de un computador cuántico puro, y se le ha añadido recientemente simulación cuántica genérica. Basado en los

principios de la mecánica cuántica, ofrece una implementación de un registro cuántico. Están disponibles las operaciones básicas para manipulación de registros como la puerta Hadamard o la puerta *Controlled-NOT* mediante un sencillo interfaz. Las medidas se pueden realizar en qubits sencillos o en un registro cuántico completo. Se ejecutará una aplicación que desarrolla el algoritmo de grover.
Entrada: Se le indica un número a buscar y el número de qubits a utilizar
Salida: Después de un número de iteraciones, devuelve la probabilidad con la que ha encontrado el número.
Página web: `http://www.libquantum.de/`
Autor: Björn Butscher y Hendrik Weimer.
Usada en: SPEC CINT 2006

C.5. h264ref

Versión: 18.2
Categoría: Compresión y conversión de video y audio.
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Software de conversión de formatos de video y audio. Utiliza los *codecs* H.264/AVC.
Entrada: Se codifica el fichero `foreman_cif.yuv`, una secuencia de imágenes en formato YUV420, a una resolución de 352x288.
Salida: Se genera el fichero `foreman_cif.264` en formato H.264.
Página web: `http://iphome.hhi.de/suehring/tml/`
Autor: Karsten Sühring et Al.
Usada en: SPEC CINT 2006

C.6. ripemd

Versión: 160
Categoría: Criptografía, Hash
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: *RACE Integrity Primitives Evaluation Message Digest* (RIPEMD-160) es un algoritmo de *hash* o resumen de mensaje desarrollado en la Universidad Católica de Lovaina (Bélgica).
Entrada: Se usa el fichero `sprot.rmd` que contiene la suma de los ficheros `sprot41.dat` usado para la aplicación **hmmer** de 419MB y `sprot.whi` usado para la aplicación **whirlpool** de 209MB.
Salida: Se obtienen 40 dígitos en hexadecimal correspondientes al resumen del mensaje.
Página web: `http://homes.esat.kuleuven.be/~bosselae/ripemd160.html`
Autor: Hans Dobbertin, Antoon Bosselaers y Bart Preneel

Usada en: Origen propio, no es usado en benchmarks conocidos.

C.7. aes

Versión: 1
Categoría: Criptografía, cifrado.
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: *Advanced Encryption Standard* AES es un algoritmo de cifrado simétrico. Fue adoptado por el gobierno de los Estados Unidos de América en 2001, después de un proceso en el que se evaluaron diferentes algoritmos de cifrado que duró 5 años. Esta evaluación se hizo para sustituir al algoritmo *Data Encryption Standard* DES.
Entrada: Se indica la contraseña para cifrar y el fichero a utilizar es `sprot41.dat`, que tiene un tamaño de 419MB, usado para la aplicación **hmmer**
Salida: Se obtiene el fichero cifrado, pero es redirigido a la salida estándar `stdout` y esta a `\dev\null` para que no escribiera en disco y obtener mayor eficiencia.
Página web: `http://csrc.nist.gov/archive/aes/index.html`
Autor: Vincent Rijmen, Joan Daemen
Usada en: Origen propio, no es usado en benchmarks conocidos.

C.8. whirlpool

Versión: 2ª Revisión
Categoría: Criptografía, hash
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: es un algoritmo de *hash* o resumen de mensaje desarrollado por uno de los autores de AES, y de este algoritmo toma ciertos detalles para su funcionamiento.
Entrada: Se usa el fichero `sprot.whi` que contiene los primeros 209MB del fichero `sprot41.dat` usado para la aplicación **hmmer**
Salida: Proporciona un *hash* de 512-bit y se representa con 128 dígitos hexadecimales.
Página web: `http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html`
Autor: Vincent Rijmen y Paulo S. L. M. Barreto,
Usada en: Origen propio, no es usado en benchmarks conocidos.

C.9. memtester

Versión: 4.0.5
Categoría: Comprobación de memoria

Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Realiza operaciones en memoria como XOR, AND, SUB y MUL para encontrar fallos en el subsistema de memoria.
Entrada: Se le indica la cantidad de memoria a comprobar y el número de repeticiones que se tienen que hacer las pruebas.
Salida: Visualiza por pantalla las operaciones que realiza, la parte de memoria donde se está realizando la prueba e indica si ha habido cualquier problema durante la ejecución.
Página web: `http://pyropus.ca/software/memtester/`
Autor: Charles Cazabon
Usada en: Origen propio, no es usado en benchmarks conocidos.

C.10. ffmpeg

Versión: 0.10
Categoría: Conversión de formatos de video/audio.
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Es un framework multimedia que permite codificar, decodificar, transcodificar, multiplexar, demultiplexar, filtrar y reproducir la mayoría de los formatos de audio y video. Genera una librería para poder ser utilizada por cualquier software para reproducción o conversión de formatos.
Entrada: Se usa el fichero `parapara.mpg` de formato YUV420, a una resolución de 352x240, con el codec de audio mp3 a 48Khz.
Salida: Se genera el ficher `parapara.avi` con el formato MPEG-4, a una resolución de 640x480 con el codec de audio Dolby AC3 en stereo a 48Khz.
Página web: `http://ffmpeg.org`
Autor: Fabrice Bellard et al.
Usada en: Phoronix Test Suite (PTS)

C.11. milc

Versión: v6
Categoría: Física, cromodinámica cuántica
Tipo de cálculo realizado: Real
Lenguaje de programación: C
Descripción: MILC es un conjunto de aplicaciones desarrolladas por "MIMD Lattice Computation", donde lo utilizan para hacer grandes simulaciones numéricas para estudiar la cromodinámica cuántica (QCD), que es, la teoría de la interacción fuerte en la física subatómica. Se usará el ejecutable `su3_rmd`.
Entrada: Se usa el fichero `sm2` que contiene parámetros de configuración de *gauge* para

la simulación a realizar a `su3_rmd`
Salida: Muestra por la salida estándar el resultado de la simulación.
Página web: `http://physics.indiana.edu/~sg/milc.html`
Autor: Steven Gottlieb et al.
Usada en: SPEC CFP 2006

C.12. povray

Versión: 3.0
Categoría: Renderización
Tipo de cálculo realizado: Real
Lenguaje de programación: C
Descripción: Es una herramienta que produce gráficos por ordenador de muy alta calidad. Utiliza el algoritmo de trazado de rayos para generar imágenes tridimensionales.
Entrada: Se usa uno de los ficheros de ejemplo que incluye povray: `radio-patio.pov`
Salida: Se obtiene el fichero `patio-radio.png` con la imagen
Página web: `http://www.povray.org/`
Autor: David Buck et al.
Usada en: SPEC CFP 2006

C.13. mlucas

Versión: 2.8x
Categoría: Numérica, búsqueda de números primos
Tipo de cálculo realizado: Real
Lenguaje de programación: C
Descripción: Realiza la búsqueda de números primos de Mersenne, que tienen la forma de $M_p = 2^p - 1$ utilizando el algoritmo de Lucas-Lehmer.
Entrada: Se le indica el rango de exponentes para buscar números primos de Mersenne
Salida: Muestra los números primos que se hayan encontrado.
Página web: `http://hogranch.com/mayer/README.html`
Autor: Ernst Mayer et al.
Usada en: SPEC CFP 2000

C.14. namd

Versión: 2.8
Categoría: Biología, simulación de moléculas
Tipo de cálculo realizado: Real

Lenguaje de programación: C++
Descripción: Es un software de dinámica molecular paralela diseñado para simulación de alto rendimiento de grandes sistemas biomoleculares.
Entrada: Se utiliza un fichero de prueba, que ofrecen los autores de namd, que se llama `tiny.namd` preparado para un benchmark, ya que es la carga significativa para un único procesador en una gran simulación.
Salida: Genera 3 ficheros `tiny.coor`, `tiny.vel` y `tiny.xsc` con los resultados de la simulación.
Página web: `http://www.ks.uiuc.edu/Research/namd/`
Autor: Jim Phillips et al.
Usada en: SPEC CFP 2006

C.15. linpack

Versión: 25.5.04
Categoría: Numérica, multiplicación de matrices.
Tipo de cálculo realizado: Real
Lenguaje de programación: Fortran
Descripción: Es una librería software que se usa para resolver sistemas de ecuaciones. A partir de aquí nació el benchmark linpack, que resuelve sistemas de ecuaciones haciendo uso intensivo de operaciones en cálculo real.
Entrada: Se le indica el tamaño de la matriz, 1500x1500.
Salida: Ofrece información del tiempo que le ha costado realizar las operaciones.
Página web: `http://www.netlib.org/linpack/`
Autor: Jack Dongarra et al.
Usada en: Linpack benchmark

Apéndice D

Resultados del benchmark

En este apéndice se muestran los resultados de la ejecución del benchmark en todas las aplicaciones.

Por cada aplicación se presenta una tabla dividida en tres partes:

- Ejecuciones nativas, sin instrumentar.
- Ejecuciones instrumentadas por instrucciones.
- Ejecuciones instrumentadas por bloques básicos.

Por cada parte se presentan tres ejecuciones sin optimizar (-O0) y tres optimizadas (-O3), además, en las instrumentadas se muestran para los tres frameworks. En todas las tablas se presenta el consumo de memoria de las aplicaciones y para las instrumentadas también se presenta el número de instrucciones o bloques básicos ejecutados.

Finalmente, en la sección D.16 se mostrará el tiempo total de ejecución del benchmark y la comparación en tiempo con SPEC.

D.1. bzip2

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	20.6	20.51	20.58	15.03	15.06	15.03
Mem. (kiB)	9440	9440	9440	9432	9432	9432
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	277.58	273.7	273.67	185.71	185.36	185.58
Mem. (kiB)	45816	45816	45816	46524	46524	46524
Slowdown	13,45	13,33	13,27	12,32	12,29	12,32
Instrucc.	73656398144	73656398144	73656398144	48475694952	48475694952	48475694952
Valgrind						
Tiempo (s)	414.49	412.92	414.91	278.01	276.94	277.34
Mem. (kiB)	58328	58328	58328	58320	58320	58320
Slowdown	20,09	20,11	20,13	18,45	18,36	18,41
Instrucc.	73656441642	73656441646	73656441638	48475738454	48475738460	48475738454
DynamoRIO						
Tiempo (s)	174.64	174.7	174.7	153.26	153.44	153.22
Mem. (kiB)	142036	142036	142036	142028	142028	142028
Slowdown	8,46	8,50	8,47	10,15	10,16	10,16
Instrucc.	73637943900	73637943900	73637943900	48457240420	48457240420	48457240420
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	72,31	72,49	72,94	66,47	66,98	66,91
Mem. (kiB)	45952	45952	45952	46564	46564	46564
Slowdown	3,88	3,91	3,93	4,86	4,90	5,38
Bloques	7063386170	7063386170	7063386170	6488305984	6488305984	6488305984
Valgrind						
Tiempo (s)	109,47	110,02	109,99	104,88	104,57	104,55
Mem. (kiB)	54996	54996	54996	54988	54988	54988
Slowdown	5,88	5,94	5,93	7,67	7,65	8,40
Bloques	4472395968	4472395967	4472395967	3998502071	3998502072	3998502072
DynamoRIO						
Tiempo (s)	29,18	29,09	29,09	21,93	21,87	21,73
Mem. (kiB)	142036	142036	142036	142028	142028	142028
Slowdown	1,56	1,57	1,57	1,60	1,60	1,74
Bloques	2139535991	2139535991	2139535991	2609502570	2609502570	2382277616

D.2. GNU go

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	133,3	133,3	133,42	82,98	82,97	82,96
Mem. (kiB)	24120	24120	24116	24328	24332	24332
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	1118,01	1117,59	1116	732,8	732,44	729,12
Mem. (kiB)	69340	69356	69352	75380	75392	75380
Slowdown	8,39	8,39	8,37	8,85	8,84	8,80
Instrucc.	247750494248	247750666655	247750666655	153754482838	153754535528	153754683960
Valgrind						
Tiempo (s)	2168,5	2165,89	2162,98	1542,52	1544,43	1546,33
Mem. (kiB)	71288	71288	71288	71496	71496	71496
Slowdown	16,29	16,27	16,23	18,62	18,64	18,66
Instrucc.	247750711832	247752391901	247785718265	153746809732	153746892452	153770256115
DynamoRIO						
Tiempo (s)	1028,65	1027,37	1032,39	789,55	790,33	789
Mem. (kiB)	156544	156548	156544	156756	156756	156756
Slowdown	7,72	7,71	7,74	9,51	9,52	9,51
Instrucc.	247745640933	247782861334	247748101772	150904441919	150904369639	150904369639
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	419,82	419,94	419,79	337,57	337,19	337,56
Mem. (kiB)	66880	66900	66900	73112	73108	73108
Slowdown	3,48	3,48	3,48	4,49	4,49	4,49
Bloques	36090511045	36095502484	36095454987	32464100442	32464133005	32464163434
Valgrind						
Tiempo (s)	847,94	849,49	847,13	765,86	760,61	767,98
Mem. (kiB)	67956	67956	67956	68164	68164	68164
Slowdown	7,02	7,04	7,02	10,19	10,12	10,22
Bloques	31790795376	31790783660	31795085040	27064807604	27064795737	27064795727
DynamoRIO						
Tiempo (s)	199,54	200,21	199,81	144,59	144,66	144,46
Mem. (kiB)	156544	156548	156544	156752	156756	156752
Slowdown	1,65	1,65	1,65	1,92	1,92	1,92
Bloques	1655477351	1655477351	1650396069	3809286521	3805093463	3805095450

D.3. hmmer

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	20,33	20,34	20,34	4,47	4,47	4,52
Mem. (kiB)	20892	20912	20892	20912	20832	20780
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	202,42	202,28	202,58	70,54	70,68	70,61
Mem. (kiB)	63660	63624	63660	63696	63688	63684
Slowdown	9,71	9,88	9,90	15,24	15,24	15,15
Instrucc.	54297654013	54297720973	54297720440	16649566301	16649736886	16649643995
Valgrind						
Tiempo (s)	376,88	375,67	375,65	120,75	121,4	121,51
Mem. (kiB)	69892	70020	69896	69880	69960	69884
Slowdown	18,06	18,33	18,34	26,03	26,10	26,02
Instrucc.	54298001231	54297950064	54297734601	16649955696	16649786684	16649862949
DynamoRIO						
Tiempo (s)	72,47	72,53	72,51	51	50,91	51,07
Mem. (kiB)	153392	153260	153392	153252	153208	153248
Slowdown	3,48	3,55	3,54	10,98	10,93	10,92
Instrucc.	53478979044	49205120807	53353754824	15030755553	15131107542	15001172531
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	45,69	45,74	45,71	27,87	28,03	27,9
Mem. (kiB)	63064	63020	63020	63380	63348	63440
Slowdown	2,22	2,26	2,26	6,12	6,14	6,07
Bloques	2563838468	2563770093	2563756783	2167186708	2167136122	2167187211
Valgrind						
Tiempo (s)	160,47	159,82	159,71	63,5	63,65	63,7
Mem. (kiB)	66596	66608	66632	66548	66548	66640
Slowdown	7,71	7,82	7,81	13,75	13,79	13,70
Bloques	2398092822	2398029615	2398077405	1766363309	1766390390	1766397846
DynamoRIO						
Tiempo (s)	25,26	25,09	25,14	9,19	9,23	9,14
Mem. (kiB)	153312	153316	153304	153216	153204	153188
Slowdown	1,22	1,23	1,23	2,02	2,03	2,00
Bloques	2382277616	2382274878	2382282498	2144631367	2144645182	2144660538

D.4. libquantum

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	2,97	2,97	2,97	1,14	1,17	1,14
Mem. (kiB)	2712	2712	2712	2716	2716	2716
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	41,32	42,5	41,3	18,56	18,9	18,44
Mem. (kiB)	38964	38964	38964	39068	39068	39068
Slowdown	13,85	14,24	13,84	16,05	15,94	15,95
Instrucc.	9486298035	9486298035	9486298035	4172204824	4172204824	4172204824
Valgrind						
Tiempo (s)	52,24	52,29	52,56	25,82	25,8	25,21
Mem. (kiB)	59456	59456	59456	59460	59460	59460
Slowdown	17,46	17,48	17,57	22,16	21,60	21,64
Instrucc.	9486307171	9486307163	9486307159	4172213780	4172213780	4172213784
DynamoRIO						
Tiempo (s)	23,89	23,72	23,81	14,3	14,25	14,21
Mem. (kiB)	135136	135136	135136	135140	135140	135140
Slowdown	7,98	7,92	7,95	12,26	11,92	12,19
Instrucc.	9483103387	9483103387	9483103387	4169009342	4169009342	4169009342
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	45,69	45,74	45,71	27,87	28,03	27,9
Mem. (kiB)	63064	63020	63020	63380	63348	63440
Slowdown	2,22	2,26	2,26	6,12	6,14	6,07
Bloques	2563838468	2563770093	2563756783	2167186708	2167136122	2167187211
Valgrind						
Tiempo (s)	160,47	159,82	159,71	63,5	63,65	63,7
Mem. (kiB)	66596	66608	66632	66548	66548	66640
Slowdown	7,71	7,82	7,81	13,75	13,79	13,70
Bloques	2398092822	2398029615	2398077405	1766363309	1766390390	1766397846
DynamoRIO						
Tiempo (s)	25,26	25,09	25,14	9,19	9,23	9,14
Mem. (kiB)	153312	153316	153304	153216	153204	153188
Slowdown	1,22	1,23	1,23	2,02	2,03	2,00
Bloques	2382277616	2382274878	2382282498	2144631367	2144645182	2144660538

D.5. h264ref

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	175,56	175,51	175,52	52,2	52,23	52,23
Mem. (kiB)	22800	22800	22800	22924	22924	22924
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	1969,15	1976,8	1979,48	731,61	731,16	730,83
Mem. (kiB)	63640	63624	63640	66536	66368	66368
Slowdown	11,20	11,25	11,27	13,96	13,96	13,95
Instrucc.	468222306245	468222571081	468222609480	151093059135	151094273120	151092430740
Valgrind						
Tiempo (s)	2615,92	2613,49	2610,26	913,42	913,12	918,55
Mem. (kiB)	71628	71628	71628	71752	71752	71752
Slowdown	14,89	14,89	14,87	17,43	17,42	17,52
Instrucc.	468602886301	468602886398	468602886324	151469116709	151469116744	151469117073
DynamoRIO						
Tiempo (s)	1384,98	1379,37	1391,31	263,49	258,38	258,62
Mem. (kiB)	155224	155224	155224	155348	155348	155348
Slowdown	7,88	7,85	7,92	5,03	4,93	4,93
Instrucc.	468099926570	468100703832	468100510812	150796835198	150798200840	150798661928
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	399,97	399,48	397,53	142,96	142,73	143,14
Mem. (kiB)	62044	62076	62068	64324	64324	64324
Slowdown	2,50	2,51	2,49	3,02	3,02	3,03
Bloques	41664395913	41664395918	41664395932	10246885976	10246885990	10246885958
Valgrind						
Tiempo (s)	842,34	849,58	847,54	317,07	316,91	317,07
Mem. (kiB)	68296	68296	68296	68420	68420	68420
Slowdown	5,28	5,34	5,31	6,68	6,68	6,68
Bloques	31254620209	31254620206	31254620232	9761665724	9761665724	9761665716
DynamoRIO						
Tiempo (s)	321,68	315,51	319,49	59,37	59,33	58,61
Mem. (kiB)	155224	155224	155224	155348	155348	155348
Slowdown	2,01	1,98	2,00	1,25	1,25	1,24
Bloques	2590913223	2590913138	2590913183	760355482	760355502	760355525

D.6. ripemd

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	15,92	15,92	15,95	5,79	5,81	5,88
Mem. (kiB)	2012	2012	2012	2016	2016	2016
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	154,98	155,11	154,59	93,59	93,53	93,97
Mem. (kiB)	38608	38608	38608	38612	38612	38612
Slowdown	9,64	9,63	9,61	15,37	15,33	15,45
Instrucc.	41946768879	41946768879	41946768879	22330342413	22330342413	22330342413
Valgrind						
Tiempo (s)	393,34	392,59	388,46	209,88	210,94	211,39
Mem. (kiB)	58244	58244	58244	58248	58248	58248
Slowdown	24,43	24,35	24,10	34,42	34,50	34,73
Instrucc.	41947389200	41947389190	41947389190	22330962488	22330962492	22330962488
DynamoRIO						
Tiempo (s)	21,89	21,83	21,86	7,8	7,82	7,8
Mem. (kiB)	134604	134604	134604	134608	134608	134608
Slowdown	1,38	1,37	1,37	1,33	1,33	1,34
Instrucc.	41789987035	41789987035	41789987035	22173559980	22173559980	22173559980
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	19,73	19,75	19,8	10,14	10,15	11,12
Mem. (kiB)	38440	38440	38440	38444	38444	38444
Slowdown	1,33	1,39	1,39	1,92	1,92	2,10
Bloques	873132829	873132829	873132829	518737174	518737174	518737174
Valgrind						
Tiempo (s)	38,91	38,81	40,75	23,09	23,22	22,9
Mem. (kiB)	54912	54912	54912	54916	54916	54916
Slowdown	2,60	2,70	2,83	4,27	4,31	4,27
Bloques	810173395	810173395	810173396	446564696	446564696	446564697
DynamoRIO						
Tiempo (s)	14,85	14,85	15,83	5,72	5,61	5,68
Mem. (kiB)	134604	134604	134604	134608	134608	134608
Slowdown	0,99	1,03	1,10	1,06	1,06	1,07
Bloques	263197444	263197444	263197444	75864982	75864982	75864982

D.7. aes

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	18,54	18,61	18,46	14,36	14,32	14,24
Mem. (kiB)	2024	2024	2024	2024	2024	2024
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	284,16	283,98	283,77	215,66	216,17	216,01
Mem. (kiB)	38620	38620	38620	38620	38620	38620
Slowdown	15,14	15,09	15,20	14,57	14,92	14,97
Instrucc.	74291586364	74291586364	74291586364	57453607695	57453607695	57453607695
Valgrind						
Tiempo (s)	535,29	540,04	539,4	396,95	399,45	399,78
Mem. (kiB)	58256	58256	58256	58256	58256	58256
Slowdown	28,58	28,74	28,93	26,85	27,60	27,76
Instrucc.	74291592035	74291592031	74291592035	57453613370	57453613366	57453613366
DynamoRIO						
Tiempo (s)	233,55	233,04	233,14	217,62	217,72	217,8
Mem. (kiB)	134616	134616	134616	134616	134616	134616
Slowdown	12,44	12,37	12,47	14,67	14,99	15,07
Instrucc.	74291424373	74291424373	74291424373	57453445704	57453445704	57453445704
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	68,34	68,27	68,02	64,82	64,45	64,87
Mem. (kiB)	38452	38452	38452	38452	38452	38452
Slowdown	3,96	4,03	4,03	4,97	4,90	4,94
Bloques	8821962398	8821962398	8821962398	8113823083	8113823083	8113823083
Valgrind						
Tiempo (s)	144,64	145,58	148,45	144,47	147,57	142,88
Mem. (kiB)	54924	54924	54924	54924	54924	54924
Slowdown	8,42	8,63	8,81	11,09	11,24	10,90
Bloques	7771123026	7771123026	7771123026	7089211083	7089211083	7089211083
DynamoRIO						
Tiempo (s)	32,92	32,91	32,83	33,31	34,03	33,35
Mem. (kiB)	134616	134616	134616	134616	134616	134616
Slowdown	1,91	1,94	1,94	2,55	2,58	2,53
Bloques	4002415509	4002415509	4002415509	3451640469	3451640469	3451640469

D.8. whirlpool

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	21,64	21,64	21,68	15,87	15,87	15,85
Mem. (kiB)	1988	1988	1988	1984	1984	1984
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	275,87	276,52	276,62	140,94	137,48	136,36
Mem. (kiB)	38636	38636	38636	38632	38632	38632
Slowdown	12,69	12,73	12,72	8,85	8,63	8,56
Instrucc.	74222379295	74222379297	74222379295	37982311186	37982311186	37982311186
Valgrind						
Tiempo (s)	498,15	485,03	490,35	229,05	233,33	229,3
Mem. (kiB)	58256	58256	58256	58252	58252	58252
Slowdown	22,93	22,33	22,55	14,38	14,66	14,40
Instrucc.	74222385145	74222385145	74222385149	37982316916	37982316920	37982316920
DynamoRIO						
Tiempo (s)	33,78	33,54	33,7	26,81	26,66	26,7
Mem. (kiB)	134612	134612	134612	134608	134608	134608
Slowdown	1,56	1,55	1,55	1,69	1,68	1,68
Instrucc.	74222287028	74222287028	74222287028	37982218617	37982218617	37982218617
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	26,05	25,86	25,91	19,48	19,62	19,62
Mem. (kiB)	38436	38436	38436	38432	38432	38432
Slowdown	1,32	1,32	1,32	1,36	1,37	1,37
Bloques	1358316929	1358316929	1358316929	906899343	906899343	906899343
Valgrind						
Tiempo (s)	62,77	61,8	62,95	48,26	48,92	48,92
Mem. (kiB)	54924	54924	54924	54920	54920	54920
Slowdown	3,17	3,14	3,21	3,35	3,40	3,40
Bloques	1485706427	1485706428	1485706427	962288944	962288944	962288944
DynamoRIO						
Tiempo (s)	20,03	20,05	20,03	14,77	14,81	14,83
Mem. (kiB)	134612	134612	134612	134608	134608	134608
Slowdown	1,01	1,02	1,02	1,02	1,03	1,03
Bloques	530888668	530888668	530888668	465378216	465378216	465378216

D.9. memtester

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	96,37	96,67	96,23	45,89	46,21	46,39
Mem. (kiB)	67408	67408	67408	67408	67408	67408
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	853,55	868,61	854,01	509,59	509,63	509,22
Mem. (kiB)	103176	103176	103176	103244	103244	103244
Slowdown	8,85	8,98	8,87	11,09	11,01	10,96
Instrucc.	241201530296	241201530296	241201530296	138709496251	138709496251	138709496251
Valgrind						
Tiempo (s)	1394,89	1387,66	1386,66	746,37	745,31	746,73
Mem. (kiB)	114396	114396	114396	114396	114396	114396
Slowdown	14,48	14,36	14,42	16,26	16,12	16,10
Instrucc.	241201536832	241201536828	241201536828	138709502897	138709502901	138709502905
DynamoRIO						
Tiempo (s)	810,44	813,83	811,57	491	491,13	491,03
Mem. (kiB)	200132	200132	200132	200132	200132	200132
Slowdown	8,40	8,41	8,42	10,68	10,60	10,56
Instrucc.	241201371000	241201371000	241201371000	138709337257	138709337257	138709337257
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	262,92	257,74	257,73	180,18	180,21	180,15
Mem. (kiB)	103076	103076	103076	103172	103172	103172
Slowdown	3,03	3,01	3,00	3,96	3,95	3,95
Bloques	31173422341	31173422341	31173422341	22306660323	22306660323	22306660323
Valgrind						
Tiempo (s)	412,74	411,55	407,47	288,89	288,91	289,47
Mem. (kiB)	111064	111064	111064	111064	111064	111064
Slowdown	4,78	4,82	4,76	6,38	6,36	6,36
Bloques	29075595091	29075595091	29075595091	18573054766	18573054765	18573054765
DynamoRIO						
Tiempo (s)	109,11	104,79	105,83	74,67	74,48	74,6
Mem. (kiB)	200132	200132	200132	200132	200132	200132
Slowdown	1,26	1,22	1,23	1,64	1,63	1,63
Bloques	1108619690	1108619690	1108619690	831792336	831792336	831792336

D.10. ffmpeg

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	4,56	4,56	4,52	4,44	4,41	4,42
Mem. (kiB)	22956	22956	22956	24360	24360	24360
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	61,53	61,59	62,16	59,01	59,01	58,99
Mem. (kiB)	65184	65184	65184	66996	66996	66996
Slowdown	13,55	13,59	13,76	13,35	13,43	13,44
Instrucc.	14181309912	14181311778	14181329060	13651672255	13651668591	13651685335
Valgrind						
Tiempo (s)	158,34	158,17	158,01	157,18	157,67	157,27
Mem. (kiB)	75996	75996	75996	77400	77400	77400
Slowdown	34,49	34,52	34,62	35,13	35,47	35,38
Instrucc.	14184459886	14184458954	14184460118	13654922353	13654922421	13654922244
DynamoRIO						
Tiempo (s)	32,77	32,78	32,83	30,77	30,73	30,73
Mem. (kiB)	155360	155360	155360	156764	156764	156764
Slowdown	7,14	7,16	7,20	6,89	6,92	6,92
Instrucc.	14166867955	14166867702	14166867805	13637273113	13637274631	13637272906
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	20,11	20,07	20	19,47	19,38	19,5
Mem. (kiB)	63860	63860	63860	65768	65784	65768
Slowdown	4,51	4,50	4,53	4,51	4,50	4,52
Bloques	1231713961	1231710900	1231711011	1156607294	1156607439	1156607440
Valgrind						
Tiempo (s)	93,4	93	93,04	91,77	91,87	91,62
Mem. (kiB)	72664	72664	72664	74068	74068	74068
Slowdown	20,26	20,18	20,36	20,52	20,50	20,50
Bloques	1078267036	1078264642	1078267216	1010901183	1010901248	1010898795
DynamoRIO						
Tiempo (s)	7,56	7,55	7,57	7,3	7,32	7,33
Mem. (kiB)	155360	155360	155360	156764	156764	156764
Slowdown	1,65	1,65	1,66	1,64	1,64	1,65
Bloques	1198264965	1198264859	1198261935	1120973424	1120973305	1120973266

D.11. milc

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	13,07	13,1	13,11	12,57	12,56	12,55
Mem. (kiB)	20688	20688	20688	20716	20716	20716
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	98,63	98,63	98,59	90,25	89,83	89,84
Mem. (kiB)	59016	59016	59008	59232	59240	59240
Slowdown	7,57	7,54	7,53	7,20	7,17	7,17
Instrucc.	25332642996	25332642951	25332642997	23054066769	23054066698	23054066650
Valgrind						
Tiempo (s)	186,44	185,7	185,08	179,4	179,62	179,13
Mem. (kiB)	67672	67672	67672	67700	67700	67700
Slowdown	14,27	14,17	14,11	14,27	14,30	14,26
Instrucc.	25332664956	25332664946	25332665003	23054088185	23054088433	23054088218
DynamoRIO						
Tiempo (s)	23,12	23,05	23,03	20,92	20,95	20,96
Mem. (kiB)	153112	153112	153112	153140	153140	153140
Slowdown	1,77	1,76	1,76	1,66	1,67	1,67
Instrucc.	25332217058	25332217130	25332217063	23053639394	23053639519	23053639522
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	16,95	16,84	16,84	15,55	15,61	15,58
Mem. (kiB)	58684	58676	58668	58864	58864	58864
Slowdown	1,37	1,36	1,36	1,31	1,32	1,31
Bloques	642302583	642302584	642302542	562161334	562161276	562161300
Valgrind						
Tiempo (s)	67,58	67,82	67,77	67,71	67,8	67,58
Mem. (kiB)	64340	64340	64340	64368	64368	64368
Slowdown	5,40	5,43	5,42	5,64	5,63	5,60
Bloques	618954025	618954016	618953981	533178012	533177998	533177989
DynamoRIO						
Tiempo (s)	13,44	13,41	13,46	12,91	12,93	12,96
Mem. (kiB)	153112	153112	153112	153140	153140	153140
Slowdown	1,07	1,07	1,07	1,07	1,07	1,07
Bloques	365223774	365223790	365223740	296635117	296635077	296635074

D.12. povray

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	20,67	20,59	20,59	11,42	11,36	11,36
Mem. (kiB)	9672	9672	9672	9708	9708	9708
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	171,1	171,3	171,4	86,05	86,17	85,88
Mem. (kiB)	52544	52576	52592	53516	53516	53524
Slowdown	8,31	8,35	8,35	7,60	7,64	7,62
Instrucc.	40880091223	40880040916	40879930936	19478328487	19478328874	19478329037
Valgrind						
Tiempo (s)	354,7	353,49	356,36	182,58	182,58	181,37
Mem. (kiB)	63524	63524	63524	63560	63560	63560
Slowdown	17,18	17,18	17,32	16,01	16,09	15,98
Instrucc.	40890689362	40890689308	40890689996	19490557438	19490557442	19490554679
DynamoRIO						
Tiempo (s)	148,48	148,66	148,36	79,15	79,14	79,08
Mem. (kiB)	142108	142108	142108	142128	142128	142128
Slowdown	7,18	7,22	7,20	6,93	6,96	6,96
Instrucc.	40437582205	40437582770	40437582385	19035068653	19035068123	19035068105
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	49,3	49,27	48,98	33,18	33,03	33,03
Mem. (kiB)	51152	51124	51136	52340	52340	52340
Slowdown	2,67	2,67	2,65	3,28	3,27	3,27
Bloques	4992500785	4992512652	4992474673	3208282021	3208263741	3208264584
Valgrind						
Tiempo (s)	142,08	142,4	142,76	82,08	82,5	81,81
Mem. (kiB)	60192	60192	60192	60228	60228	60228
Slowdown	7,62	7,64	7,65	7,98	8,03	7,97
Bloques	3762148897	3762127642	3762141997	2494999759	2494986930	2494983484
DynamoRIO						
Tiempo (s)	33,48	33,02	32,96	15,18	15,13	15,12
Mem. (kiB)	142108	142092	142092	142144	142128	142128
Slowdown	1,79	1,77	1,76	1,47	1,47	1,47
Bloques	244200167	244167577	244162585	2756061306	2756061265	2756073299

D.13. mlucas

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	35,05	34,87	35,03	14,6	14,65	14,62
Mem. (kiB)	10004	10004	10004	9340	9340	9340
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	307,11	309,24	307,2	199,01	199,2	199,01
Mem. (kiB)	49560	49560	49560	49648	49648	49648
Slowdown	8,78	8,88	8,78	13,64	13,61	13,62
Instrucc.	85256941803	85256941829	85256941799	53554796686	53554796681	53554796686
Valgrind						
Tiempo (s)	✗	✗	✗	✗	✗	✗
Mem. (kiB)	✗	✗	✗	✗	✗	✗
Slowdown	✗	✗	✗	✗	✗	✗
Instrucc.	✗	✗	✗	✗	✗	✗
DynamoRIO						
Tiempo (s)	45,12	45,07	45,02	22,5	22,48	22,58
Mem. (kiB)	142428	142428	142428	141764	141764	141764
Slowdown	1,29	1,29	1,29	1,54	1,53	1,54
Instrucc.	85256744807	85256744807	85256744807	53554599416	53554599416	53554599416
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	37,91	37,99	38	21,16	21,15	21,19
Mem. (kiB)	48244	48244	48244	48324	48324	48324
Slowdown	1,21	1,21	1,21	1,62	1,62	1,62
Bloques	1387505307	1387505303	1387505307	909951883	909951875	909951875
Valgrind						
Tiempo (s)	✗	✗	✗	✗	✗	✗
Mem. (kiB)	✗	✗	✗	✗	✗	✗
Slowdown	✗	✗	✗	✗	✗	✗
Bloques	✗	✗	✗	✗	✗	✗
DynamoRIO						
Tiempo (s)	32,48	32,48	32,48	14,39	14,42	14,4
Mem. (kiB)	142428	142428	142428	141764	141764	141764
Slowdown	1,03	1,03	1,03	1,09	1,09	1,09
Bloques	278148461	278148462	278148462	227746162	227746162	227746162

D.14. namd

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	14,95	15,06	15,04	7,12	7,1	7,11
Mem. (kiB)	19744	19744	19744	19208	19208	19208
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	155,66	156,39	155,38	87,18	87,14	87,21
Mem. (kiB)	66980	66992	66988	69096	69020	68628
Slowdown	10,49	10,47	10,41	12,37	12,34	12,40
Instrucc.	38252761764	38252801255	38252759804	19832746065	19832741407	19832743883
Valgrind						
Tiempo (s)	214,64	213,66	213,89	113,25	113,22	114,94
Mem. (kiB)	78660	78660	78660	74860	74860	74860
Slowdown	14,32	14,16	14,20	15,85	15,77	16,10
Instrucc.	38253097254	38253096968	38253096409	19833493146	19833491949	19833488839
DynamoRIO						
Tiempo (s)	34,43	34,43	34,51	19,43	19,44	19,44
Mem. (kiB)	152144	152144	152144	151608	151608	151608
Slowdown	2,30	2,29	2,29	2,72	2,71	2,73
Instrucc.	38250283476	38250318179	38250282771	19814092244	19814078844	19814080525
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	26,68	26,49	26,75	18,73	18,68	18,78
Mem. (kiB)	64552	63992	64540	66636	65696	65820
Slowdown	2,04	2,02	2,06	3,08	3,07	3,08
Bloques	971692620	971695374	971691998	549464758	549461424	549464261
Valgrind						
Tiempo (s)	65	65	65,15	44,12	44,24	44,12
Mem. (kiB)	75328	75328	75328	71528	71528	71528
Slowdown	4,75	4,73	4,79	6,81	6,84	6,83
Bloques	949810174	949810463	949810293	511693175	511691568	511692316
DynamoRIO						
Tiempo (s)	17,09	17,17	17,03	8,55	8,54	8,54
Mem. (kiB)	152144	152144	152144	151608	151608	151608
Slowdown	1,25	1,25	1,25	1,32	1,32	1,32
Bloques	540152965	540159401	540149109	344577457	344575920	344575061

D.15. linpack

	Sin optimizar			Optimizado		
	Ejec.1	**Ejec.2**	**Ejec.3**	**Ejec.1**	**Ejec.2**	**Ejec.3**
Ejecución nativa						
Tiempo (s)	6,95	7,15	6,96	4,33	4,35	4,34
Mem. (kiB)	20692	20692	20692	20692	20692	20692
Instrumentación a nivel de instrucciones						
Pin						
Tiempo (s)	66,54	66,07	66,28	22,04	22,02	22
Mem. (kiB)	57848	57448	57496	57496	57496	57496
Slowdown	9,59	9,22	9,52	5,13	5,09	5,09
Instrucc.	19647034175	19647034027	19647034197	6246035342	6246035106	6246035254
Valgrind						
Tiempo (s)	96,45	96,04	95,93	32,47	32,28	32,36
Mem. (kiB)	77956	77956	77956	77956	77956	77956
Slowdown	13,88	13,39	13,75	7,51	7,41	7,44
Instrucc.	19647045951	19647045432	19647045951	6246046953	6246046921	6246046849
DynamoRIO						
Tiempo (s)	19,83	19,8	19,8	8,6	8,6	8,6
Mem. (kiB)	153120	153120	153120	153120	153120	153120
Slowdown	2,85	2,76	2,84	1,99	1,97	1,98
Instrucc.	19646781826	19646781851	19646781804	6245779182	6245778866	6245779042
Instrumentación a nivel de bloques básicos						
Pin						
Tiempo (s)	9,39	9,35	9,37	5,98	5,86	5,84
Mem. (kiB)	57400	57396	57400	57456	57456	57456
Slowdown	1,38	1,33	1,37	1,40	1,39	1,40
Bloques	615171369	615171418	615171345	330477783	330477785	330477792
Valgrind						
Tiempo (s)	28,86	28,26	28,85	18,05	18,03	18,14
Mem. (kiB)	73600	73600	73600	73600	73600	73600
Slowdown	4,15	3,95	4,15	4,11	4,15	4,22
Bloques	608174575	608174559	608174609	316652503	316652492	316652465
DynamoRIO						
Tiempo (s)	6,96	6,92	6,92	4,55	4,47	4,5
Mem. (kiB)	153120	153120	153120	153120	153120	153120
Slowdown	1,00	0,97	1,00	1,03	1,03	1,05
Bloques	609917486	609917509	609917517	325785898	325785905	325785922

D.16. Tiempo de ejecución de los benchmarks

En esta sección se va a hacer una comparación entre el tiempo que hubiera durado la ejecución del benchmark propio desarrollado para este PFC y el que hubiera durado SPEC 2006. Se compararán los resultados con un equipo similar [NEC08].

En el equipo similar, a la ejecución de la parte de cálculo entero (CINT) de SPEC le cuesta 13 horas y 59 minutos. En el benchmark propio la ejecución de cálculo entero sin instrumentar, dura 38 minutos y 40 segundos. Extrapolando el resultado de SPEC del equipo similar, con la media de instrumentación en instrucciones (13.97x) por cada uno de los tres frameworks, la ejecución de este hubiera durado 25 días.

La duración total del benchmark propio es de 32 horas y 20 minutos.

Apéndice E

Código fuente aplicaciones usadas en el benchmark

En el presente capítulo se muestra el código fuente de las herramientas creadas para instrumentar las aplicaciones en el benchmark. La primera herramienta es la que instrumenta por instrucciones y la segunda herramienta es la que instrumenta por bloques básicos.

E.1. Instrumentación por instrucciones

E.1.1. Pin

```
#include <stdio.h>
#include "pin.H"
#include <iostream>

UINT64 icuenta = 0;

VOID contar() { icuenta++; }

VOID Instruction(INS ins, VOID *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)contar, IARG_END);
}

VOID Fini(INT32 code, VOID *v)
{
    std::cerr << "Instrucciones: " << icuenta << endl;
}

int main(int argc, char * argv[])
```

```
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

E.1.2. DynamoRIO

```
#include "dr_api.h"

#define DISPLAY_STRING(msg) dr_printf("%s\n", msg);
#define DISPLAY_STRING_ERR(msg) dr_fprintf(STDERR,"%s\n", msg);
#define NULL_TERMINATE(buf) buf[(sizeof(buf)/sizeof(buf[0])) - 1] = '\0'

static uint64 icuenta=0; //Contador de instrucciones
static void cuenta(void) { icuenta++; } //Código a añadir

static void event_exit(void);
static dr_emit_flags_t event_basic_block(void *drcontext,
  void *tag, instrlist_t *bb, bool for_trace, bool translating);

DR_EXPORT void
dr_init(client_id_t id)
{
   dr_register_exit_event(event_exit);
   dr_register_bb_event(event_basic_block);
   dr_log(NULL, LOG_ALL, 1, "Inicializando cliente 'icuenta'\n");
}

static void event_exit(void)
{
   char msg[512];
   int len;
   len = dr_snprintf(msg, sizeof(msg)/sizeof(msg[0]),
                     "Instrucciones:  %llu \n", icuenta);
   DR_ASSERT(len > 0);
   NULL_TERMINATE(msg);
   DISPLAY_STRING_ERR(msg);
}

static dr_emit_flags_t
```

```
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating)
{
   instr_t *instr;
   int i;
   // Se recorre el bloque básico y se instrumentan todas las instrucciones
   for (instr  = instrlist_first(bb), num_instrs = 0;
         instr != NULL;
         instr = instr_get_next(instr)) {
   dr_insert_clean_call(drcontext, bb, instr),
                        (void *)cuenta, false, 0 );
   }
   return DR_EMIT_DEFAULT;
}
```

E.1.3. Valgrind

```
#include "pub_tool_basics.h"
#include "pub_tool_tooliface.h"
#include "pub_tool_options.h"
#include "pub_tool_libcbase.h"
#include "pub_tool_libcassert.h"
#include "pub_tool_machine.h"
#include "pub_tool_libcprint.h"
#include "pub_tool_debuginfo.h"

static ULong icuenta  = 0;

static void contar(void)  { icuenta++; }

static void ic_post_clo_init(void)  { }

static IRSB* ic_instrument ( VgCallbackClosure* closure,
                      IRSB* sbIn,
                      VexGuestLayout* layout,
                      VexGuestExtents* vge,
                      IRType gWordTy, IRType hWordTy )
{
   IRDirty*   di;
   Int        i;
   IRSB*      sbOut;

   sbOut = deepCopyIRSBExceptStmts(sbIn);
```

```
   i = 0;
   while (i < sbIn->stmts_used && sbIn->stmts[i]->tag != Ist_IMark) {
      addStmtToIRSB( sbOut, sbIn->stmts[i] );
      i++;
   }

   for (; i < sbIn->stmts_used; i++) {
      IRStmt* st = sbIn->stmts[i];
      if (!st || st->tag == Ist_NoOp) continue;

      switch (st->tag) {
         case Ist_IMark:
               di = unsafeIRDirty_0_N( 0, "contar",
                                       VG_(fnptr_to_fnentry)( &contar ),
                                       mkIRExprVec_0() );
               addStmtToIRSB( sbOut, IRStmt_Dirty(di) );
            break;

         default:
           tl_assert(0);
     }
   }
   return sbOut;
}

static void ic_fini(Int exitcode)
{
      VG_(umsg)("Instrucciones:  %'llu\n", icuenta);
      VG_(umsg)("Exit code:       %d\n", exitcode);
}

static void ic_pre_clo_init(void)
{
   VG_(details_name)            ("icuenta");
   VG_(details_version)         (NULL);
   VG_(details_description)     ("Contador de instrucciones");
   VG_(details_copyright_author)("GNU GPL");
   VG_(details_avg_translation_sizeB) ( 275 );
   VG_(basic_tool_funcs)        (ic_post_clo_init,
                                 ic_instrument,
                                 ic_fini);
}
```

E.2. Instrumentación por bloques básicos

E.2.1. Pin

```
#include <stdio.h>
#include "pin.H"
#include <iostream>

static UINT64 bcuenta = 0;

VOID contar() { bcuenta++; }

VOID Trace(TRACE trace, VOID *v)
{
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)contar, IARG_END);
    }
}

VOID Fini(INT32 code, VOID *v)
{
    std::cerr << "Bloques básicos: " << bcuenta << endl;
}

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

E.2.2. DynamoRIO

```
#include "dr_api.h"

#define DISPLAY_STRING(msg) dr_printf("%s\n", msg);
#define DISPLAY_STRING_ERR(msg) dr_fprintf(STDERR,"%s\n", msg);
#define NULL_TERMINATE(buf) buf[(sizeof(buf)/sizeof(buf[0])) - 1] = '\0'

static uint64 bcuenta=0; //Contador de bloques básicos
```

```
static void contar(void) { bcuenta++; } //Código a añadir

static void event_exit(void);
static dr_emit_flags_t event_basic_block(void *drcontext,
                  void *tag, instrlist_t *bb, bool for_trace, bool translating);

DR_EXPORT void
dr_init(client_id_t id)
{
    dr_register_exit_event(event_exit);
    dr_register_bb_event(event_basic_block);
    dr_log(NULL, LOG_ALL, 1, "Inicializando cliente 'bcuenta'\n");
}

static void event_exit(void)
{
    char msg[512];
    int len;
    len = dr_snprintf(msg, sizeof(msg)/sizeof(msg[0]),
                      "Bloques básicos: %llu\n", bcuenta);
    DR_ASSERT(len > 0);
    NULL_TERMINATE(msg);
    DISPLAY_STRING_ERR(msg);
}

static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating)
{

    dr_insert_clean_call(drcontext, bb, instrlist_first(bb) ),
                         (void *)contar, false, 0 );
    return DR_EMIT_DEFAULT;
}
```

E.2.3. Valgrind

```
#include "pub_tool_basics.h"
#include "pub_tool_tooliface.h"
#include "pub_tool_libcassert.h"
#include "pub_tool_libcprint.h"
#include "pub_tool_debuginfo.h"
```

```
#include "pub_tool_libcbase.h"
#include "pub_tool_options.h"
#include "pub_tool_machine.h"

static ULong bcuenta = 0;

static void contar(void) { bcuenta++; }

static void lk_post_clo_init(void) { }

static
IRSB* lk_instrument ( VgCallbackClosure* closure,
                      IRSB* sbIn,
                      VexGuestLayout* layout,
                      VexGuestExtents* vge,
                      IRType gWordTy, IRType hWordTy )
{
   IRDirty*   di;
   Int        i;
   IRSB*      sbOut;
   Char       fnname[100];

   sbOut = deepCopyIRSBExceptStmts(sbIn);

   i = 0;
   while (i < sbIn->stmts_used && sbIn->stmts[i]->tag != Ist_IMark) {
      addStmtToIRSB( sbOut, sbIn->stmts[i] );
      i++;
   }

   di = unsafeIRDirty_0_N( 0, "cuenta",
                    VG_(fnptr_to_fnentry)(
                   &cuenta ), mkIRExprVec_0() );
      addStmtToIRSB( sbOut, IRStmt_Dirty(di) );
   }

   for (; i < sbIn->stmts_used; i++) {
      IRStmt* st = sbIn->stmts[i];
      if (!st || st->tag == Ist_NoOp) continue;
      addStmtToIRSB( sbOut, st );
   }

   return sbOut;
```

```
}

static void lk_fini(Int exitcode)
{
      VG_(umsg)("Bloques básicos: %'llu\n", bcuenta);
      VG_(umsg)("Exit code:       %d\n", exitcode);
}

static void lk_pre_clo_init(void)
{
   VG_(details_name)            ("BBCount");
   VG_(details_version)         (NULL);
   VG_(details_description)     ("Cuenta de bloques basicos");
   VG_(details_copyright_author)("GNU GPL");
   VG_(details_avg_translation_sizeB) ( 200 );

   VG_(basic_tool_funcs)          (lk_post_clo_init,
                                   lk_instrument,
                                   lk_fini);
}

VG_DETERMINE_INTERFACE_VERSION(lk_pre_clo_init)
```