# Performance Evaluation of Dynamic Binary Instrumentation Frameworks

Ricardo J. Rodríguez, Juan A. Artal, and J. Merseguer

***Resumen*—Code analysis, static or dynamic, is a primary mean for improving correctness and performance of software applications. Dynamic binary analysis (DBA) refers the methods to analyse runtime behaviour of binary code. Nowadays, DBA tools are implemented using dynamic binary instrumentation (DBI) frameworks, which can add arbitrary code into the execution flow of the binary. Unfortunately, a DBA tool increases the execution time of the analysed binary dramatically, as extra code is being executed. Experiments got figures with increments of $26x$. Therefore, it is crucial for DBA tool construction to know exact figures about such penalties and their roots. Hence, we conduct a performance evaluation of leading DBI frameworks, namely Pin, Valgrind, and DynamoRIO, for which we have built a benchmark selecting a bunch of representative tools. The evaluation that we procure here provides guidance to choose the best DBI framework suited for different needs. Moreover, the benchmark by itself is a tool ready to be eventually used for performance evaluation of future DBI frameworks.**

***Index Terms*—Performance evaluation, Benchmark, Testing, Software Tools.**

## I. Introducción

La vulnerabilidad de las aplicaciones es a día de hoy uno de los problemas más importantes que debe abordarse durante el diseño, construcción y prueba del software. La gestión del riesgo y el *análisis de código* son probablemente las dos técnicas más importantes en el desarrollo de software seguro [1].

El análisis de código verifica que no existan errores que atenten contra la funcionalidad esperada de la propia aplicación analizada. Existen dos métodos: el análisis *estático*, que analiza el código fuente de la aplicación; y el *dinámico*, que analiza la aplicación durante su ejecución. Ambos métodos presentan ventajas e inconvenientes. Mientras el análisis estático ofrece potencialmente la posibilidad de explorar todos los caminos posibles de ejecución (como las técnicas de cobertura de código), el dinámico tiene en cuenta un único camino de ejecución cada vez. Sin embargo, el dinámico puede detectar situaciones que permanecen indetectables con un análisis estático, como por ejemplo variables puntero sin inicializar o de tipos no esperados. Por tanto ambos métodos se complementan y necesitan de herramientas para llevarlos a cabo.

R.J. Rodríguez is with Dpto. de Lenguajes y Sistemas Informáticos e Ingeniería de Software, Universidad Politécnica de Madrid, 28660 Boadilla del Monte (Madrid), Spain . E-mail: rjrodriguez@fi.upm.es.

J.A. Artal and J. Merseguer are with Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, María de Luna 1, 50018 Zaragoza, Spain. E-mail: jaartal@gmail.com, jmerse@unizar.es. This work was done while R.J. Rodríguez was at Universidad de Zaragoza.

En este artículo nos centramos en las herramientas de análisis dinámico que instrumentan binarios (Dynamic Binary Analysis (DBA) tools). Se distinguen dos tipos de herramientas DBA: las que instrumentan el binario generando otro binario que contiene las funciones de instrumentación y el propio binario a analizar, llamadas herramientas de *instrumentación estática de binarios*; y las que instrumentan el binario durante su ejecución, sin necesidad de crear otro binario intermedio, llamadas herramientas de *instrumentación dinámica de binarios*.

La instrumentación de código binario (Dynamic Binary Instrumentation, DBI) es una técnica de análisis dinámico de binarios que permite la inserción de código arbitrario en la aplicación durante su ejecución. Este tipo de instrumentación resulta especialmente útil cuando cuando no se dispone del código fuente. Presentando las siguientes ventajas: independencia del lenguaje de programación con que se desarrolló la aplicación, una visión modo máquina del ejecutable, no es necesario recompilar las aplicaciones si se desean cambiar las funciones de instrumentación y permiten el descubrimiento y análisis de código generado dinámicamente. Sin embargo, su principal desventaja es la sobrecarga que inducen en el tiempo de ejecución de la aplicación que instrumentan.

Actualmente existen sistemas DBI (*frameworks* DBI) optimizados para minimizar la sobrecarga y que cuentan con librerías de desarrollo que facilitan la creación de herramientas DBA. Dependiendo del *framework* DBI se pueden crear dos tipos de herramientas DBA: las "pesadas", aquellas que realizan una conversión del binario a un código intermedio guardando meta-información (por ejemplo, de los registros y de la memoria); y las que no necesitan de código intermedio, "ligeras".

Los *frameworks* DBI permiten al programador definir las funciones de instrumentación y los puntos de instrumentación de interés, creando herramientas DBA para analizar sus aplicaciones. Estas herramientas ayudan a: descubrir los recursos consumidos por la aplicación (p.e., memoria usada o número de consultas ejecutadas), calcular correctamente su complejidad ciclomática, conseguir trazas de ejecuciones legítimas (a fin de compararlas y buscar posibles comportamientos no deseados) e incluso encontrar sus errores (punteros nulos, condiciones de carrera, fugas de memoria o divisiones por cero). Por tanto son herramientas necesarias para desarrollar un software optimizado (por ejemplo, detectando múltiples accesos a memoria que se pueden evitar) y para verificar la corrección del software (es decir, detectar posibles errores). Ejemplos de estas herramientas son los *profilers* y los *checkers* o visualizadores de la ejecución para análisis de

registros contaminados (*taint analysis* [2]). Son por tanto herramientas fundamentales no sólo para la construcción de software seguro sino también en otros campos de las ciencias de la computación como la construcción de compiladores, el procesamiento de lenguajes o los tests de arquitecturas hardware.

En cuanto al rendimiento de los *frameworks* DBI, aun habiendo sido estudiado de manera particular en algunos casos [3]–[6], no existe ningún trabajo en la literatura que realice comparativas de rendimiento entre ellos. Por ello creemos necesario llevar a cabo un estudio con el objetivo de conocer el *framework* DBI que mejor se ajustaría a las necesidades del programador en diferentes situaciones. Es decir, con qué *framework* deberá analizar sus aplicaciones, en función por ejemplo de si estas tienen códigos optimizados, con alta complejidad ciclomática, o grandes requerimientos de memoria.

Para llevar a cabo nuestro objetivo primero crearemos un *benchmark* para comparar el rendimiento de los *frameworks* DBI; y segundo, presentamos un estudio comparativo de los *frameworks* más relevantes aplicando este benchmark. En concreto, en este artículo se comparan `Pin` [7], [8], `Valgrind` [9] y `DynamoRIO` [10]. Otros sistemas, como `Bochs` [11] o `QEMU` [6] no se han considerado al no estar diseñados específicamente para análisis dinámico de binarios.

El artículo se estructura como sigue. La sección II revisa los conceptos fundamentales de la instrumentación dinámica de ejecutables. En la sección III se presenta el *benchmark* desarrollado y en la sección IV la evaluación de los *frameworks* DBI. La sección V hace una exhaustiva revisión de la literatura relacionada. La sección VI concluye el trabajo.

## II. Instrumentación dinámica de ejecutables

El análisis dinámico ofrece la ventaja de estudiar qué es lo que está ocurriendo durante la ejecución del programa, en lugar de lo que podría estar ocurriendo. En particular, la instrumentación dinámica de ejecutables (DBI) permite analizar el comportamiento de los ejecutables mientras se mantiene un control absoluto de su ejecución. Se aplica principalmente para realizar tareas de simulación de fallos de memoria cache, explotación de paralelismo u optimizaciones de código.

### *II-A. Granularidad*

La instrumentación puede tener diferentes granularidades según el objetivo del análisis a realizar:

- **Instrucción**, es la unidad mínima que se puede instrumentar. Son instrucciones en ensamblador de la arquitectura en la que se compiló el binario.
- **Bloque básico**, es una secuencia de instrucciones que finaliza con una instrucción de control de transferencia. Por ejemplo, estas instrucciones en ensamblador Intel x86 [12] son el salto condicional (`JZ`, salto si la flag *zero* es 1), incondicional (`JMP`), repeticiones (instrucciones con el prefijo `REP`), llamada o retorno de procedimiento (`CALL` o `RET`).
- **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 por ejemplo C.
- **Imagen**, que representa a todas las secciones de un ejecutable, es decir, las partes en las que se divide (por ejemplo, `.init`, `.text` o `.fini` en un ejecutable PE de Windows [13].

### *II-B. Frameworks DBI*

Como hemos dicho anteriormente un *framework* DBI permite crear herramientas DBA de instrumentación de código binario. Un *framework* DBI tiene dos componentes principales: el **núcleo** y las **herramientas DBA desarrolladas** con él.

El **núcleo** puede verse como un compilador *just-in-time*, donde la entrada es un ejecutable, que intercepta la ejecución de la primera instrucción del ejecutable y genera un nuevo código o secuencia, en función de la granularidad escogida. Esta secuencia es prácticamente idéntica a la original, pero el núcleo se asegura el retorno del control cuando finalice la ejecución de la misma. Además, el núcleo da opción a la **herramienta DBA** para ejecutar su propio código, o sea, el *código de instrumentación*. La secuencia se guarda en memoria, por lo que puede ser reutilizada sin necesidad de regenerarla cada vez, disminuyendo así el impacto en el tiempo de ejecución.

Según lo anterior una herramienta DBA es un *plug-in* o librería que añade código al ya generado por el núcleo. Se entienden dos fases de actuación:

- **Instrumentación**, se define en qué puntos de la ejecución del código binario se ha de realizar la inserción y qué código se inserta en cada punto;
- **Análisis**, es la ejecución del código tal como se describió en el párrafo anterior.

La fase de instrumentación se ejecuta una única vez durante el análisis, sin embargo, el código inyectado por la herramienta se puede llegar a ejecutar múltiples veces -obviamente dependiendo de la lógica del código binario-. Por ello el código inyectado debe estar muy afinado, además hay que destacar que éste se ejecuta de forma transparente [14] de tal forma que no pueda interferir en el comportamiento del código binario original.

Por último, para que el funcionamiento de un código binario instrumentado sea correcto tanto el núcleo como la herramienta tienen que estar trabajando en el mismo espacio de direcciones que el binario instrumentado. Es decir, deben estar o bien en "espacio de usuario", donde residen las aplicaciones. o bien en "espacio de *kernel*", donde residen los módulos o *drivers* del sistema operativo.

Para profundizar en el funcionamiento de *frameworks* DBI concretos, se recomienda el lector visitar las referencias de `Pin` [7], [8], `Valgrind` [9] o `DynamoRIO` [10].

## III. Benchmark para frameworks DBI

Esta sección presenta el *benchmark* que hemos desarrollado para evaluar *frameworks* DBI.

Un *benchmark* realiza un conjunto de operaciones definidas por: una **carga de trabajo**, o conjunto de programas que comprende el *benchmark*, que devuelve algún tipo de resultado; y una **métrica**, que describe la metodología para las pruebas a realizar. En lo que respecta a la carga de trabajo, el criterio más importante fue la selección de programas que hiciesen un uso intensivo de la CPU. La sección III-A describe los aspectos más importantes. Respecto a la métrica, los datos que esperamos obtener están relacionados con el tiempo de ejecución de los programas y los requisitos de memoria necesarios. Lo más importante será conocer cuánto tiempo más cuesta ejecutar una aplicación instrumentada con respecto a la versión no instrumentada. La sección III-B describe los aspectos más importantes a este respecto.

Previamente a la creación del nuevo *benchmark* se estudió la posibilidad de utilizar *benchmarks* ya existentes. La mayoría de ellos fueron descartados ya que sólo funcionaban para el sistema operativo Windows, lo que limitaba el espectro de la comparativa. Este fue el caso de *benchmarks* como Futuremark [15] (por ejemplo, 3DMark y PCMark) o SysMark 2012 [16]. El *benchmark* que más se acercaba a los propósitos planteados era SPEC CPU2006 [17], ya que dispone de una versión para Windows y Linux y además ofrece dos cargas de trabajo de CPU diferentes, una para enteros (CINT2006) y otra para punto flotante (CFP2006). Sin embargo, este *benchmark* usa un hardware de referencia (concretamente, Sun Ultra Enterprise 2 de 1997), con lo que el resultado obtenido está normalizado con respecto a la máquina. Además, otra característica de SPEC CPU2006 es que es una solución comercial, no gratuita, sin embargo un trabajo científico debe ofrecer la posibilidad de replicar los experimentos por terceros, y en nuestro caso particular dar la posibilidad de utilizar el *benchmark* en futuros trabajos -comparar nuevas versiones de los *frameworks* o incluso nuevos *frameworks*-. Las razones expuestas nos llevaron a descartar SPEC CPU2006.

### *III-A. Carga de trabajo*

La Tabla I presenta las características mas importantes de las aplicaciones incluidas en el *benchmark*. Se han considerado algunos de los programas usados por SPEC CPU2006, aquellos para los que el código fuente está disponible con licencia libre. Se seleccionaron 15 aplicaciones de uso intensivo de CPU, clasificadas en cuatro grandes grupos que cubren todos los aspectos del análisis[1] : *cálculo entero*, *cálculo real*, *gran demanda de entrada/salida* (E/S) y *acceso reiterado a memoria*. La carga de trabajo en nuestro *benchmark* será siempre la misma para cada ejecutable.

### *III-B. Métrica*

La metodología utilizada está inspirada en SPEC CPU2006. Así, cada aplicación analizada se ejecuta con diferentes argumentos y nivel de optimización (desde `-O0` a `-O3`). Además, cada aplicación se ejecuta varias veces para obtener una media significativa de su tiempo de ejecución. Para aproximar la ejecución del *benchmark* a un entorno real donde diferentes aplicaciones concurren por el uso de los recursos del computador, las ejecuciones no se realizan de forma consecutiva sino intercalada. A diferencia de SPEC CPU2006 y con el fin de disminuir el tiempo de ejecución de las pruebas, nuestra metodología usa ficheros de entrada que ofrecen menos carga de proceso a las aplicaciones. La ejecución de las aplicaciones se realiza en primer lugar sin instrumentación, y después instrumentadas, siendo los resultados recogidos los siguientes:

- 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; y el
- número de instrucciones o bloques básicos ejecutados.

Se definen además intervalos de confianza con el fin de descartar y repetir el análisis para aquellas aplicaciones que o bien hayan utilizado un tiempo de ejecución excesivo o bien hayan sido paginadas muchas veces en comparación con las otras. Esto es así porque las aplicaciones están preparadas para que su resultado de ejecución sea siempre el mismo, por lo que cabe esperar que su ejecución dure, aproximadamente, lo mismo.

Para las mediciones de tiempo se usa el programa `/usr/bin/time` que se diferencia de la orden `times` integrada en el intérprete de órdenes de UNIX, esta última ofrece mucha menos información. El programa utilizado 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` deriva de las llamadas al sistema `wait3` o `wait4`, así los datos obtenidos serán tan fiables como los que pueda ofrecer la llamada correspondiente. En los sistemas dónde no estén disponibles `wait3` o `wait4` se usará la órden `times`. En el sistema utilizado para la evaluación la llamada a sistema usada fue `wait4`.

Para medir el consumo de memoria, el *benchmark* ejecuta un proceso en segundo plano (*background*) después de ejecutar la aplicación a evaluar. Este proceso se encarga de revisar en `/proc/<pid de la aplicación>/status` el consumo en el campo `VmPeak`, que muestra el pico de memoria utilizada por la aplicación. Para definir el intervalo de consulta de consumo se establecieron inicialmente valores inferiores a 1 segundo. Este intervalo se fue aumentando y ajustando mediante la comprobación de que el consumo de memoria no superaba el valor de pico del último intervalo.

*III-B1. Herramientas DBA:* Para la ejecución instrumentada de la carga de trabajo se han creado dos herramientas DBA: una con granularidad a nivel de instrucción y otra a nivel de bloque básico.

Con la primera herramienta conseguimos el máximo *slowdown*, ya que se instrumentan todas las instrucciones de la

[1]En general 15 aplicaciones en tres lenguajes diferentes es un diseño aceptado entre los *benchmark* más representativos, por ejemplo SPEC CPU2006 consta de entre 12 y 17 aplicaciones en 3 lenguajes.

| Nombre aplicación | Versión | Lenguaje | Palabras clave | Origen |
|---|---|---|---|---|
| **Cálculo entero** | | | | |
| `bzip2` | 1.0.6 | C | Compresión | SPEC CINT2006 |
| `GNU go` | 3.8 | C | Inteligencia artificial - Juegos | SPEC CINT2006 |
| `hmmer` | 3.0 | C | Genética | SPEC CINT2006 |
| `h264ref` | 18.2 | C | Compresión video | SPEC CINT2006 |
| `libquantum` | 1.0.0 | C | Física, computación cuántica | SPEC CINT2006 |
| **Cálculo real** | | | | |
| `namd` | 2.8 | C++ | Biología, simulación de moléculas | SPEC CFP2006 |
| `povray` | 3.0 | C | Renderización | SPEC CFP2006 |
| `milc` | v6 | C | Física, Cromodinámica cuántica | SPEC CFP2006 |
| `mlucas` | 2.8x | C | Numérica, cálculo de números primos | SPEC CFP2000 |
| `linpack` | 29.5.04 | Fortran | Numérica, multiplicación de matrices | Linpack benchmark |
| **Gran demanda de E/S** | | | | |
| `whirlpool` | 2ª Rev | C | Criptografía, Hash | (n/a) |
| `ripemd` | 160 | C | Criptografía, Hash | (n/a) |
| `aes` | 1 | C | Criptografía, cifrado | (n/a) |
| `ffmpeg` | 0.10 | C | Conversión de formatos video/audio | Phoronix Test Suite |
| **Acceso reiterado a memoria** | | | | |
| `memtester` | 4.0.5 | C | Comprobación de memoria defectuosa | (n/a) |

(n/a): Programa no incluido en ningún otro *benchmark*.

Tabla I
APLICACIONES INCLUIDAS EN EL *benchmark* PARA EVALUACIÓN DE FRAMEWORKS DBI.

aplicación analizada. De esta forma, por cada instrucción ejecutada, también se ejecuta el código añadido. En este caso, ese código incrementa un contador. Al finalizar la ejecución se devuelve como resultado el número total de instrucciones ejecutadas.

Con la segunda herramienta conseguimos una sobrecarga media, por ello la mayoría de trabajos que evalúan *frameworks* DBI usan este nivel de granularidad [3], [4], [18]. En este caso, se instrumentan todos los bloques básicos y por cada bloque básico ejecutado, se ejecuta el código añadido, que es idéntico al anteriormente descrito, es decir, incremento de un contador. Al finalizar la ejecución de la aplicación se obtiene la suma del número total de bloques básicos ejecutados.

### *III-C. Obtención del* benchmark

El *benchmark* se distribuye con licencia GNU GPL version 3 y está disponible para su descarga gratuita en la página web http://webdiis.unizar.es/GISED/?q=tool/benchdbi

La citada web proporciona una breve guía para su correcta utilización y diversas gráficas que muestran resultados comparativos de diferentes *frameworks* DBI en plataformas de 32 y 64 bits. El nombre que se ha dado al *benchmark* es `benchDBI`.

## IV. EVALUACIÓN DE FRAMEWORKS DBI

Esta sección resume los aspectos más interesantes de los experimentos realizados para evaluar los *frameworks* DBI seleccionados. En primer lugar, se define el entorno de pruebas usado. Por último, se muestran los resultados y su análisis crítico.

### *IV-A. Entorno de pruebas*

La Tabla II resume el hardware y software utilizados en los experimentos. Las versiones de los *frameworks* DBI utilizadas han sido `Pin` v2.10, `Valgrind` v3.7.0 y `DynamoRIO` v2.2.0-2. En particular se ha deshabilitado uno de los núcleos del procesador, a fin de evitar que las aplicaciones pudieran ejecutar más de un proceso a la vez, y que el porcentaje de uso de CPU superara el $100\,\%$.

| Especificaciones del hardware | |
|---|---|
| **Procesador** | Intel Core 2 Duo CPU T7300 |
| **Características** | 2.00 GHz, 667 MHz bus |
| **CPU(s)** | 2 cores (un core desactivado) |
| **Caché primer nivel** | 32 KiB I + 32 KiB D (por core) |
| **Caché segundo nivel** | 4 MiB I+D |
| **Memoria RAM** | 2 GiB DDR2 SODIMM 667 Mhz |
| **Disco Duro** | 120 GB HITACHI HTS54161 |
| **Especificaciones del software** | |
| **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 II
ESPECIFICACIONES DEL HARDWARE Y SOFTWARE UTILIZADO EN LA EVALUACIÓN.

Dado que los experimentos requieren que el resto de procesos en ejecución afecten lo mínimo posible, el computador se inicia en nivel 3 de ejecución (multiusuario con red). En este nivel no se tiene ningún proceso relacionado con el gestor de escritorio o salvapantallas que pueda influir durante la ejecución de las pruebas, que se lanzarán de manera remota mediante conexión `ssh`.

La ejecución del benchmark genera los resultados que se especificaron en la sección III-B y lo hace en dos ficheros: un fichero de registro y un fichero de valores. El fichero de registro recoge el número de instrucciones o de bloques básicos ejecutados por aplicación instrumentada y el consumo de memoria por proceso. El fichero con valores recoge información sobre 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 órdenes con que se

invoca a la aplicación.

### *IV-B. Discusión de resultados*

Tal y como se describe en [7], [9], la instrumentación dinámica de ejecutables provoca una ralentización de la ejecución del programa instrumentado. En este trabajo queremos analizar la eficiencia de cada *framework* DBI, en la creación de herramientas DBA, e intentar inferir cuál es el más adecuado según sea el tipo de aplicación que se va a instrumentar. En primer lugar se muestran los resultados del slowdown medio usando granulidad de instrucciones y bloques básicos, y después se discute el consumo de memoria.

La Tabla III recoge el *slowdown* medio para cada aplicación de cálculo entero y cálculo real, con granularidad de instrucción, con diferentes grados de optimización y para cada uno de los *frameworks* DBI considerados en el análisis. Nótese que no se han obtenido resultados de la aplicación `mlucas` con `Valgrind` debido a que éste provoca un fallo en su ejecución[2] Las Figuras 1(a) y (b) resumen los datos de la tabla anterior mostrando el *slowdown* medio para cada *framework* discriminando por tipo de aplicación y nivel de optimización. Las medias se han obtenido como la media aritmética de los *slowdown* de la tabla anterior. Al estilo de otros resultados comparativos, se ha optado por no incluir los resultados de E/S ni memoria ya que se considerar menos significativos. Como primera conclusión, se observa que, tanto individualmente como en las medias, `Valgrind` es siempre la opción que introduce más *slowdown*. Entendemos que este resultado es derivado del hecho de que `Valgrind` crea herramientas "pesadas", generando código intermedio con penalización respecto al resto de *frameworks* DBI que crean herramientas "ligeras". Por otro lado, las aplicaciones instrumentadas con `DynamoRIO` son las más rápidas, tanto en aplicaciones de cálculo entero como real e independientemente de la opción de optimización utilizada durante la compilación. Es interesante destacar que la optimización no mejora el *slowdown* sino que lo empeora, exceptuando las aplicaciones de cálculo real donde se observa una ligera mejoría en los casos de `Valgrind` y `DynamoRIO`.

La Figura 2 muestra el consumo medio de memoria, considerando tanto las aplicaciones no instrumentadas como sus versiones instrumentadas con cada uno de los *frameworks* DBI y usando una granularidad de bloque básico. Los resultados indican que el *framework* que menor consumo de memoria requiere es `Pin`, independientemente del tipo de aplicación (cálculo real, cálculo entero, E/S, memoria). Otra conclusión que se extrae es que el consumo de memoria de la aplicación instrumentada sufre un incremento proporcional al consumo de memoria de la aplicación nativa. La Tabla IV resume los datos ofrecidos en la Figura 2 mostrando el consumo medio de memoria para todas las aplicaciones en función de cada *framework* DBI, así como la desviación media. El consumo medio de memoria tiene en cuenta exactamente la memoria adicional empleada por la aplicación instrumentada.

[2]`Valgrind` no puede trabajar con números de 80 bits en arquitecturas de 32 ó 64 bits. Este fallo está documentado en https://bugs.kde.org/show_bug.cgi?id=197915.

| Framework | Media consumo (kiB) | Desviación media (kiB) |
|---|---|---|
| *Aplicaciones con nivel de optimización -O0* | | |
| `Pin` | 39885 | 3113 |
| `Valgrind` | 53033 | 3736 |
| `DynamoRIO` | 132478 | 73 |
| *Aplicaciones con nivel de optimización -O3* | | |
| `Pin` | 40871 | 4006 |
| `Valgrind` | 52789 | 3535 |
| `DynamoRIO` | 132475 | 72 |

Tabla IV
CONSUMO MEDIO DE MEMORIA POR FRAMEWORK DBI.

La desviación media de `DynamoRIO` infiere que el consumo de memoria de este *framework*, a pesar de ser el más elevado de todos, es prácticamente fijo e independiente de la aplicación instrumentada, con lo que es un dato muy importante a tener en cuenta en casos en los que se requiera analizar aplicaciones con restricciones de memoria (como por ejemplo, en un sistema empotrado). Sin embargo, la opción de `Pin` aparece como la más eficiente respecto al consumo de memoria, quedando `Valgrind` como una opción alternativa ya que su consumo es inferior a la mitad de `DynamoRIO`.

La Figura 3 compara el *slowdown* de las aplicaciones con diferentes niveles de optimización, e instrumentadas a nivel de instrucción y de bloques básicos con los tres *frameworks* DBI usados en la comparación. Los resultados muestran, como era de esperar, que una instrumentación más fina produce un *slowdown* superior en cualquier tipo de aplicación. Otra conclusión que se extrae de esta figura es que la opción más eficiente al instrumentar siempre es `DynamoRIO`.

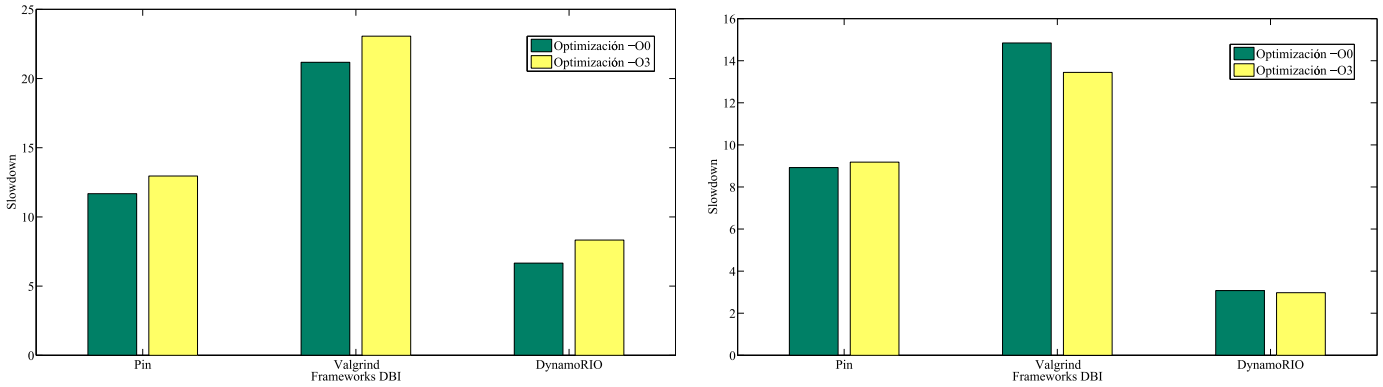
En resumen, cuando la aplicación a analizar es de cálculo intensivo (real o entero) nuestro *benchmark* dice que `DynamoRIO` es la mejor opción. En el caso de ser además una aplicación optimizada, `DynamoRIO` también es el recomendado al presentar una mayor velocidad de ejecución durante el análisis. Sin embargo `DynamoRIO` es también el que más memoria consume, así en un sistema con recursos limitados puede imponer un detrimento en el rendimiento consecuencia del incremento de la paginación a disco por falta de memoria física. La recomendación para aplicaciones con uso intensivo de memoria es `Pin`, al ser el *framework* que menos sobrecarga la memoria. A pesar de que `Valgrind` obtiene un rendimiento más modesto creemos que es debido a la generación de herramientas "pesadas" para añadir meta-información. Sin embargo, debemos destacar que precisamente la utilidad de esta meta-información es la que ha hecho de `Valgrind` uno de los *frameworks* más usados por la comunidad de desarrolladores software.

## V. TRABAJO RELACIONADO

La revisión de la literatura nos muestra diferentes trabajos que comparan el rendimiento de *frameworks* DBI, a saber [3]–[6]. Con respecto a ellos nuestro trabajo presenta diferencias y aporta contribuciones, a la vez que los complementa. En primer lugar, todos estos trabajos utilizan benchmarks comerciales de propósito general, sin embargo nuestro trabajo desarrolla un benchmark específico para la evaluación de

| Aplicaciones | Optimización -O0 | | | Optimización -O3 | | |
|---|---|---|---|---|---|---|
| | Pin | Valgrind | DynamoRIO | Pin | Valgrind | DynamoRIO |
| Cálculo entero | | | | | | |
| bzip2 | 13,35$x$ | 20,11$x$ | 8,48$x$ | 12,31$x$ | 18,41$x$ | 10,16$x$ |
| GNU go | 8,38$x$ | 16,26$x$ | 7,72$x$ | 8,83$x$ | 18,64$x$ | 9,51$x$ |
| hmmer | 9,83$x$ | 18,24$x$ | 3,52$x$ | 15,21$x$ | 26,05$x$ | 10,94$x$ |
| h264ref | 11,24$x$ | 14,88$x$ | 7,88$x$ | 13,96$x$ | 17,46$x$ | 4,96$x$ |
| libquantum | 13,98$x$ | 17,50$x$ | 7,95$x$ | 15,98$x$ | 21,80$x$ | 12,12$x$ |
| Cálculo real | | | | | | |
| povray | 8,34$x$ | 17,23$x$ | 7,20$x$ | 7,62$x$ | 16,03$x$ | 6,95$x$ |
| milc | 7,55$x$ | 14,18$x$ | 1,76$x$ | 7,18$x$ | 14,28$x$ | 1,67$x$ |
| mlucas | 8,81$x$ | (n/a) | 1,29$x$ | 13,62$x$ | (n/a) | 1,54$x$ |
| linpack | 9,44$x$ | 13,67$x$ | 2,82$x$ | 5,10$x$ | 7,45$x$ | 1,98$x$ |

Tabla III
*Slowdown* DE LAS APLICACIONES DE CÁLCULO ENTERO Y CÁLCULO FLOTANTE (GRANULARIDAD DE INSTRUCCIÓN).




(a) Aplicaciones de cálculo entero

(a) Aplicaciones de cálculo real

Figura 1. *Slowdown* medio en el benchmark (granularidad de instrucción).

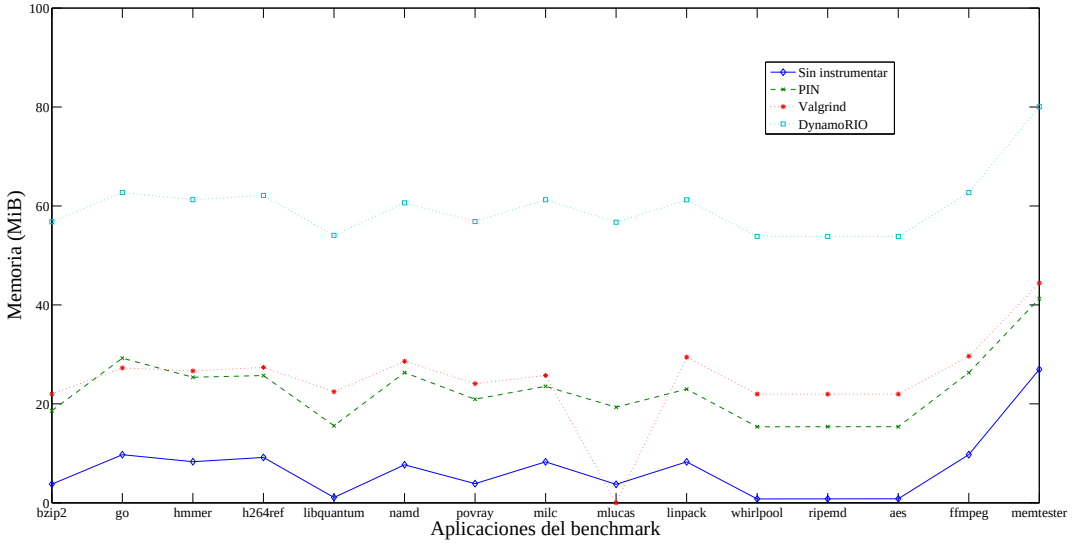



Figura 2. Consumo medio de memoria de las aplicaciones optimizadas (granularidad de bloque básico).

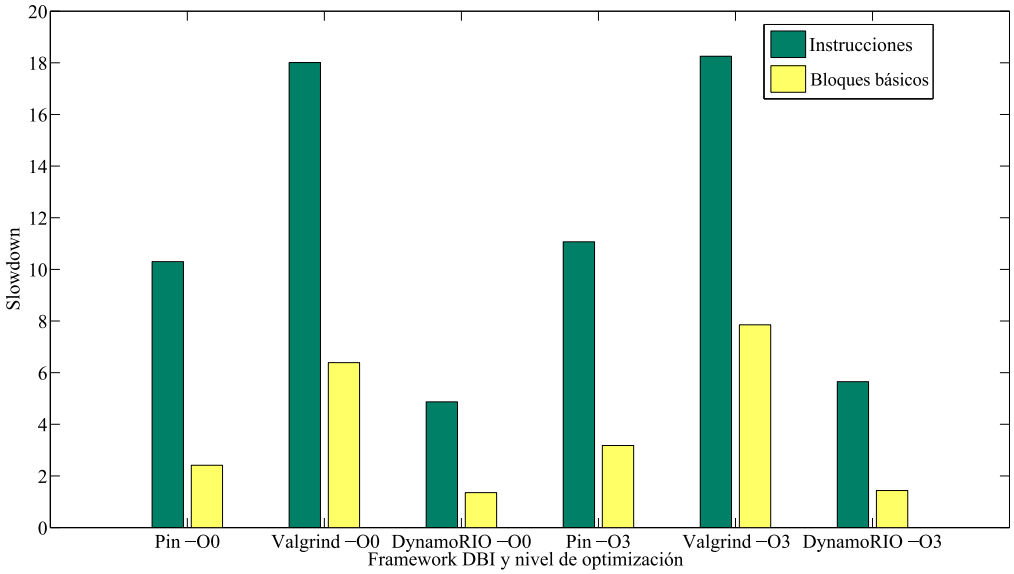



Figura 3. *Slowdown* en instrumentación con granularidad de instrucciones y bloques básicos, por *frameworks* y optimizaciones.

*frameworks* DBI y lo pone en manos de los interesados para futuras evaluaciones y para la validación de los resultados aquí presentados. La evaluación que presenta nuestro trabajo analiza el consumo de memoria, dato no contemplado en ningún otro de los trabajos relacionados. Por otro lado, nuestro *benchmark* es el único apropiado para evaluar el desempeño de los *frameworks* DBI en dos sistemas operativos diferentes (Linux y Windows) y en dos arquitecturas diferentes (32 y 64 bits). Nuestra evaluación es también la única que instrumenta las aplicaciones con dos granularidades diferentes para el análisis: por instrucciones, de manera que se consigue el *slowdown* máximo; y por bloques básicos, para conseguir una sobrecarga media. Sin embargo, los trabajos relacionados presentan resultados muy interesantes en su contexto que destacamos en el siguiente párrafo. Es interesante también destacar el trabajo de [19], donde examinan la sobrecarga introducida por el framework de Pin en algunas de las aplicaciones más comunes de Windows, además de algunas aplicaciones de SPEC CPU 2000 (INT Y FP, de 32 y 64 bits).

Guha et al. [3] usan SPEC CINT2000 para comparar `Strata` [20] con `Pin`, `Valgrind` y `DynamoRIO`, con granularidad a nivel de bloque básico. Los resultados muestran que `Pin` es más eficiente 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. Nuestros resultados son más críticos con `Pin` mientras no lo son tanto con `Valgrind`. En cualquier caso nótese que la comparativa en [3] usa las versiones de Pin 2, Valgrind 2.2.0 y DynamoRIO 0.9.3. Ruiz-Alvarez y Hazelwood [5] comparan `Pin` y `DynamoRIO` usando SPEC CINT2006, estas pruebas obtienen en promedio un *slowdown* de $1{,}22X$ en el caso de `DynamoRIO` y de $1{,}45x$ en el caso de `Pin`. Nuestros resultados siguen la misma línea, obteniendo mejor *slowdown* DynamoRIO que Pin para aplicaciones de cálculo entero. Nuestros resultados consideran además aplicaciones de cálculo real, gran demanda de E/S y memoria.

Los trabajos que describimos a continuación introducen `Bochs` [11] y/o `QEMU` [6] en la comparación. Como dijimos en la Introducción nuestra evaluación no contempla estos *frameworks* ya que se enfocan a la emulación del hardware y del sistema operativo estando nuestro trabajo enfocado al desarrollo de aplicaciones. Tampoco se ha considerado Strata al ser un *framework* para traducción dinámica de software [20]. Bellard [6] presenta `QEMU` y compara su *slowdown* con `Valgrind` y `Bochs` utilizando el benchmark BYTEmark [21]. El *slowdown* de `QEMU` es de $4x$ en operaciones de cálculo entero y de $10x$ en cálculo real, siendo la opción más rápida frente a `Bochs` y `Valgrind`, mejorándolos en un *slowdown* de $30x$ y $1{,}2x$, respectivamente. Cabe destacar también el trabajo de Weaver y McKee [4], donde comparan aplicaciones creadas con `Pin`, `Valgrind` y `QEMU` para abordar el problema del excesivo tiempo de ejecución de benchmarks como SPEC CINT2000 y SPEC CINT2006.

Por último, aunque no dedicados a la comparación de *frameworks* DBI, destacamos los trabajos en [22] y [18], ambos usan SPEC CINT2000 [17]. Chen et al. [22] buscan métodos para ejecutar más rápidamente aplicaciones instrumentadas a nivel de instrucción. Uh et al. [18] estudian el origen de la sobrecarga introducida por `Pin` así como posibles oportunidades de mejora del rendimiento de `Pin`.

## VI. Conclusiones y Trabajo Futuro

El análisis dinámico de binarios (DBA) se aplica en muy diferentes campos de las ciencias de la computación como construcción de compiladores, procesado del lenguaje o test de arquitecturas hardware. Además, es una técnica fundamental para el desarrollo de software seguro, tema transversal en la computación y actualmente objeto de investigación. Las herramientas DBA desarrolladas con *frameworks* DBI (Dynamic Binary Instrumentation) permiten analizar el comportamiento de las aplicaciones durante su ejecución mediante la

instrumentación del mismo binario, sin necesidad de recurrir al código fuente. Sin embargo, esta instrumentación conlleva una sobrecarga en el tiempo de ejecución de la aplicación instrumentada, sobrecarga introducida por el *framework* DBI. Consecuentemente ofrecen un rendimiento muy bajo en comparación con la ejecución nativa (no instrumentada) de la aplicación. Por ello actualmente es difícil justificar el uso de una aplicación instrumentada en entornos donde sería recomendable, como por ejemplo el seguimiento de trazas de ejecución en servidores virtuales para identificar riesgos en su seguridad. Existen diferentes *frameworks* DBI en el mercado, intentando solventar este problema de la mejor manera posible, recomendándose cada uno como la mejor opción. Sólo unos pocos trabajos en la literatura han intentado dar luz sobre este aspecto realizando comparativas entre los *frameworks*. Estos trabajos se han centrado en aspectos específicos pero sin tratar el problema de manera general. Nosotros hemos intentado complementar esos trabajos abordando el problema de manera exhaustiva.

En conclusión, nuestro trabajo aporta resultados en un entorno de evaluación completo y de interés para el desarrollo de herramientas DBA: considera para ello un amplio abanico de programas en diferentes dominios (cálculo intensivo entero y real, demanda de E/S y acceso a memoria) y realiza un estrés sobre ellos considerando dos granularidades representativas para los *frameworks* DBI. Para ello hemos desarrollado un *benchmark* formado por 15 aplicaciones con diferentes opciones de compilación, útil para diferentes sistemas operativos y capaz de comparar frameworks en cuanto al desempeño de las aplicaciones con ellos construidas, considerando también el uso de memoria. Los resultados obtenidos en la evaluación han puesto de manifiesto las bondades de los *frameworks* para construir herramientas DBA en función del rendimiento que se quiera obtener, de las características de las aplicaciones a analizar y del entorno de ejecución del propio *framework*.

Las posibles líneas de investigación que se abren a partir de este trabajo son varias. En primer lugar, puede ser interesante extender la comparativa de *frameworks* DBI a otros parámetros, como las APIs y herramientas que proporcionan cada uno de ellos. Por otro lado, desarrollar un controlador a nivel de núcleo del sistema operativo para una mejor monitorización las aplicaciones del benchmark, o el uso de aplicaciones específicas (a nivel de aplicación del sistema operativo) para la medición de tiempo y conteo de instrucciones, como Intel VTune Amplifier XE[3].

## AGRADECIMIENTOS

Este trabajo ha sido parcialmente financiado por el Instituto Nacional de Ciberseguridad (INCIBE) de acuerdo con la Regla 19 del Plan de Confianza Digital (Agenda Digital de España), la Universidad de León bajo el acuerdo X43 y el Grupo de Computación Distribuida (DisCo) del Gobierno de Aragón (Ref. T94). Parte de este trabajo se realizó mientras R. J. Rodríguez trabajaba para la Universidad de Zaragoza.

[3] http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe

## REFERENCIAS

[1] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[2] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. Internet Society, 2005.

[3] A. Guha, J. D. Hiser, N. Kumar, J. Yang, M. Zhao, S. Zhou, B. R. Childers, J. W. Davidson, K. Hazelwood, and M. L. Soffa, "Virtual Execution Environments: Support and Tools," in *21th International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 1–6.

[4] V. Weaver and S. McKee, "Using Dynamic Binary Instrumentation to Generate Multi-platform SimPoints: Methodology and Accuracy," in *High Performance Embedded Architectures and Compilers*, ser. Lecture Notes in Computer Science, P. Stenström, M. Dubois, M. Katevenis, R. Gupta, and T. Ungerer, Eds. Springer Berlin Heidelberg, 2008, vol. 4917, pp. 305–319.

[5] A. Ruiz-Alvarez and K. Hazelwood, "Evaluating the Impact of Dynamic Binary Translation Systems on Hardware Cache Performance," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2008, pp. 131–140.

[6] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–46.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[8] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal, "Analyzing Parallel Programs with Pin," *Computer*, vol. 43, pp. 34–41, 2010.

[9] N. Nethercote and J. Seward, "Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[10] D. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, Massachusetts Institute of Technology (MIT), 2004.

[11] Kevin Lawton et al., "BOCHS: the cross platform IA-32 emulator," Online, accedido el 31 de octubre de 2014. [Online]. Available: http://bochs.sourceforge.net/

[12] Intel, *80386 Programmer's Reference Manual*. Intel Corporation, 1986. [Online]. Available: http://books.google.es/books?id=yJCG7EjVvUQC

[13] M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format," http://msdn.microsoft.com/en-us/library/ms809762.aspx, [Online], accessed 31 de octubre de 2014.

[14] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent Dynamic Instrumentation," *SIGPLAN Not.*, vol. 47, no. 7, pp. 133–144, March 2012.

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

[16] Applications Performance Corporation, "SYSmark 2012," http://bapco.com/products/sysmark-2012, 2012.

[17] Standard Performance Evaluation Corporation, "SPEC CPU2006 benchmarks," http://www.spec.org, 2006.

[18] G.-R. Uh, R. Cohna, B. Yadavalli, R. Peri, and R. Ayyagari, "Analyzing Dynamic Binary Instrumentation Overhead," in *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, October 2006.

[19] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach, "Dynamic Program Analysis of Microsoft Windows Applications," in *IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 2–12.

[20] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. Soffa, "Retargetable and Reconfigurable Software Dynamic Translation," in *International Symposium on Code Generation and Optimization (CGO)*, 2003, pp. 36–47.

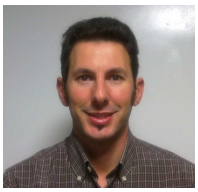
[21] Inquisitor, "BYTEmark benchmark suite," http://www.inquisitor.ru/doc/tests/bytemark.html.

[22] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 377–388, June 2008.

**Ricardo J. Rodríguez** received the M.S. and Ph.D. degrees in computer science engineering from the University of Zaragoza, Zaragoza, Spain, in 2010 and 2013, respectively. His Ph.D. study was related to the study of performance analysis and resource optimisation in large critical systems modelled with Petri nets. He is currently a Research Assistant with the Department of Computer Languages and Systems and Software Engineering, Technical University of Madrid, Madrid, Spain. His research interests include performance analysis and optimisation of large discrete event systems and computer security.

He was a Visiting Researcher with the School of Computer Science and Informatics, Cardiff University, Cardiff, U.K. He is currently a Research Assistant with the Department of Computer Languages and Systems and Software Engineering, Technical University of Madrid, Madrid, Spain. His research interests include performance analysis and optimization of large discrete event systems, secure software engineering, and fault-tolerant systems.

Dr. Rodríguez is a member of IEEE Computer Society. He has been serving as a Referee for international journals and for several international conferences and workshops.



**Juan Antonio Artal** received the B.S. degree in computer science engineering from the University of Zaragoza, Zaragoza, Spain, in 2012. He is working as a System Administrator at Indra since twelve years ago, mainly working with UNIX and Oracle. He has been in the IT Industry for 15 years and worked with several flavors of the UNIX operating system including Sun Solaris, Tru64 Unix, HP-UX, and Red Hat Enterprise Linux. He has been serving telecommunication business and government customers.

Mr. Artal holds several certifications, such as *Red Hat Certified System Administrator* (RHCSA), *Red Hat Certified Engineer* (RHCE), *Oracle Database 11g Administrator Certified Associate* (OCA) and *Oracle Certified Expert: Oracle Real Application Clusters 11g and Grid Infrastructure Administrator* (OCE). He is currently working toward the Cisco Certified Network Associate (CCNA) certification.

**José Merseguer** received the B.S. and M.S. degrees in computer science and software engineering from the Technical University of Valencia, Valencia, Spain, and the Ph.D. degree in computer science from the University of Zaragoza, Zaragoza, Spain.

He has been a Visiting Researcher at the Universities of Turin, Cagliari and Politecnico di Milano in Italy, Carleton University in Canada and Iowa State University in USA. His main research interests include performance and dependability analysis of software systems. He has published several papers in international journals and Conferences. He is co-author of the book "Model-driven Dependability Assessment of Software Systems" by Springer.

Dr. Merseguer currently serves as co-chair of the 5th International ACM/SPEC International Conference on Performance Engineering. He has also served as PC member in international conferences related to performance evaluation: ACM/SPEC ICPE, ACM QoSA, ACM WOSP, ValueTools and EPEW. He is the director of the Master of Systems Engineering and Informatics at the University of Zaragoza.