Evaluación del uso de Streams y el estilo Pipe and Filter en el Análisis de Código Fuente

Evaluation of using Streams and the Pipe and Filter style on Source Code Analysis

Paul Mendoza del Carpio *

RESUMEN

La extracción de modelos desde código fuente requiere un análisis exhaustivo de los elementos de código de una aplicación. Mediante el uso de un framework para la extracción mencionada, el presente trabajo evalúa el efecto de aplicar streams y el estilo arquitectónico *Pipe and filter*. Las implementaciones desarrolladas con el framework, aplican el estilo *Pipe and filter* mediante pipelines con streams de datos durante la ejecución de los pasos necesarios para el análisis de código fuente. Las pruebas se realizaron sobre proyectos de aplicación presentes en repositorios de GitHub. Las implementaciones que aplicaron paralelismo, mostraron una notoria y estable mejora sobre aquellas secuenciales cuando se analizaban mayores cantidades de elementos de código fuente. En consecuencia, la capacidad de ejecución paralela e incremental que permite el uso de streams y el estilo *Pipe and filter*, hacen de ellos una alternativa importante para la mejora de tiempos de ejecución en el campo de análisis de código fuente.

Palabras clave: *streams, pipe, filter, paralelismo.*

ABSTRACT

The extraction of models from source code requires an exhaustive analysis of code elements in an application. Through the use of a framework for the mentioned extraction, this work evaluates the effect of applying streams and the architectural style *Pipe and filter*. The implementations developed with the framework, apply the *Pipe and filter* style through pipelines with data streams during the execution of the necessary steps for the source code analysis. The tests were performed on application projects gotten from GitHub repositories. The implementations which applied parallelism, showed a notorious and stable improvement in comparison with sequential executions when a higher quantity of source code elements was analyzed. In consequence, the capacity of parallel and incremental execution that allows the use of streams and the *Pipe and filter* style, make them an important alternative for the improvement of execution time in the field of source code analysis.

Key words: *streams, pipe, filter, parallelism.*

* Magister en Ingeniería de Software y Candidato a Doctor en Ciencias de la Computación. Profesional IBM Certificado. Docente de la UAP.
E-mail: pmendozadelcarpio@gmail.com
Twitter: @paulnetedu

INTRODUCCIÓN

El uso de hardware multicore permite incrementar la capacidad de procesamiento en aplicaciones hoy en día. Java 8, la última versión recientemente disponible en la plataforma, incorpora mecanismos de manejo de concurrencia, facilitando la tarea de implementación de aplicaciones multicore. El uso de las expresiones Lambda presentes en Java 8, proporciona importantes elementos para los desarrolladores de aplicaciones: streams, paralelismo, iteradores internos, operaciones perezosas (lazy operation), interfaces funcionales, operaciones MapReduce.

Las aplicaciones de streaming son programas que procesan datos en forma continua tan pronto como los resultados están disponibles. Muchas implementaciones de streaming son empleadas en aplicaciones que precisan de alto rendimiento. Cada stream es una secuencia de datos, y cada operador (filter, map, reduce, peek) consume elementos de streams entrantes, produciendo otros hacia streams de salida. Puesto que los operadores pueden ser ejecutados en forma concurrente, ellos exponen paralelismo en forma inherente.

Asimismo, el estilo arquitectónico *Pipe and filter* tiene entre sus ventajas más importantes el soporte de ejecución concurrente. En el presente trabajo se hace uso de pipes y filters encadenados en pipelines, los cuales son recorridos por streams de datos para el análisis de código fuente, tomando como fin el generar modelos representativos del código analizado, para ello se emplea un framework desarrollado por el autor.

HIPÓTESIS

El uso de *Streams* y el estilo arquitectónico *Pipe and filter*, puede reducir el tiempo empleado en el análisis de código fuente.

MATERIALES Y MÉTODOS

Material

A. *Framework Java*: framework implementado para la extracción de modelos desde código fuente, el cual emplea Java en su versión 8 haciendo uso de colecciones, streams, y expresiones Lambda.

B. *Eclipse Java Development Tools*: conjunto de bibliotecas que brindan soporte a IDEs en el desarrollo de cualquier aplicación Java. Entre estas bibliotecas se encuentran implementaciones para el análisis de código fuente a través de AST (*Abstract Syntax Tree*).

C. *GitHub*: servicio de hosting Web basado en Git, el cual cuenta con una considerable cantidad de proyectos open source. Utilizado como fuente de repositorios donde se alojan aplicaciones de diversos dominios.

D. *Eclipse Luna*: entorno de desarrollo que soporta la programación de aplicaciones en la versión 8 de Java.

E. *Hojas de cálculo*: hojas de cálculo Excel con el registro de características y tiempos empleados en las aplicaciones Java. También son empleados para la generación de gráficas de resultados.

METODOLOGÍA

A. Implementación de un framework que realice análisis de código fuente basado en streams y el estilo Pipe and Filter

Se implementa un framework en Java 8 enfocado en realizar análisis de código fuente para la extracción de modelos desde código fuente, dicho framework emplea streams para el procesamiento de datos, grafos para la representación de los modelos, y un DSL (*Domain Specific Language*) para la definición de reglas. El uso de streams permite abordar el escenario de aplicación con eficiencia, los siguientes operadores son aplicados sobre los streams de objetos:

- Collect: Operador De Reducción Que Acumula Los Elementos De Entrada En Un Contenedor.
- Filter: Operador Que Retorna Un Stream Con Los Elementos Del Stream Original Que Cumplan Un Predicado Determinado.
- Map: Operador Que Retorna Un Stream Conformado Por Los Resultados De Ejecutar Una Función Sobre Los Elementos Del Stream Original.
- Peek: Operador Que Ejecuta Una Acción Determinada Sobre Cada Elemento Del Stream, Retornando El Stream Original.
- Reduce: Operador Que Toma Los Elementos De Entrada Y Produce Un Resultado Único Resumido.

Cabe señalar que los operadores pueden ser ejecutados en forma concurrente, de esta forma ellos permiten ejecutar operaciones en forma paralela.

El siguiente diagrama muestra los principales paquetes presentes en el framework:

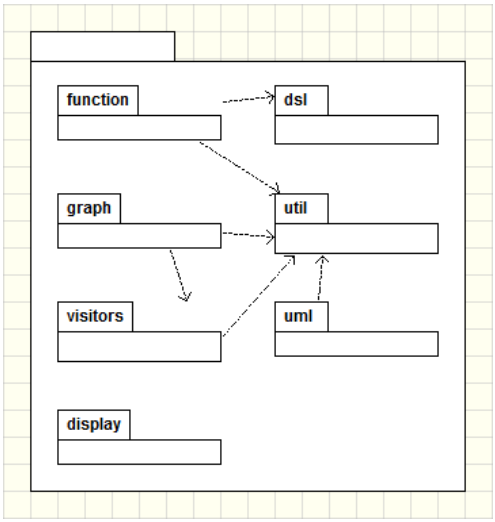

Figura 1. Diagrama de paquetes del framework.

A continuación se da una descripción de cada paquete:

- Function: Funciones Y Predicados Para La Creación Y Evaluación De Elementos Del Modelo Extraído.
- Graph: Funciones Para La Creación De Nodos Y Relaciones.
- Util: Utilitarios Para El Manejo De Archivos, Grafos, Streams, Y Código Fuente.

- Visitors: Elementos Visitor (Patrón De Diseño Gof) Basados En Árboles Ast Para La Identificación De Elementos Y Relaciones En Los Archivos De Código Fuente.
- Uml: Funciones Para La Creación De Elementos Clasificadores De Uml, Incluye Operadores Sobre Los Mismos.
- Display: Funciones Y Operadores Para La Visualización De Elementos Del Modelo. Incluye La Generación De Formatos Estándar.
- dsl: clases para la lectura y obtención de información contenida en los archivos XML de reglas. Tales reglas son empleadas para identificar elementos de relevancia a formar parte del modelo final extraído.

B. Recolección de código fuente de proyectos de aplicación desde una fuente global

Se buscan y recolectan proyectos de aplicación desde la plataforma GitHub, plataforma Web donde grandes cantidades de proyectos albergan su código fuente, archivos y recursos que también son frecuentemente empleados como fuentes de datos para diversas investigaciones. Desde los repositorios de GitHub se seleccionan aplicaciones Web implementadas en lenguaje Java, eligiéndose 17 aplicaciones.

La siguiente figura muestra el número de archivos presente en cada aplicación tomada en consideración:

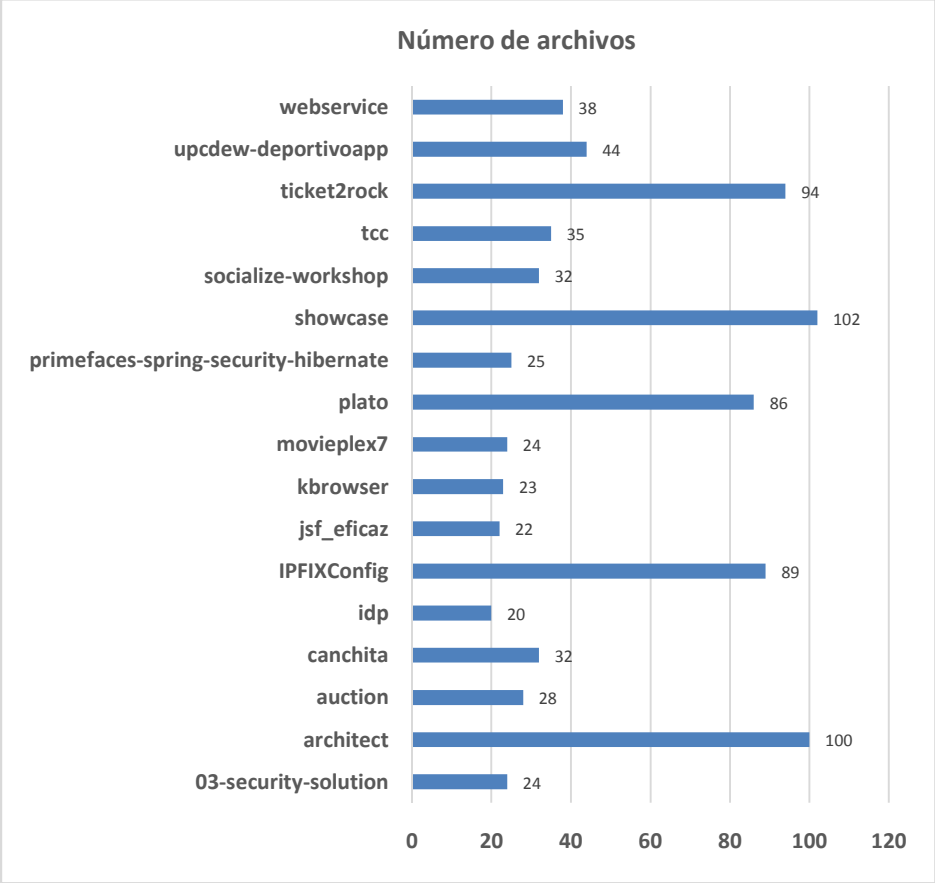

Figura 2. Número de archivos por proyecto de aplicación.

Con fin de proporcionar una medición adicional sobre el tamaño de las aplicaciones empleadas, en la figura 3 se muestra el LOC (*Lines of Code*) de cada proyecto de aplicación considerado.

C. Implementación de aplicaciones para el análisis de código fuente de los proyectos obtenidos

Se implementan aplicaciones para el análisis de código de cada proyecto de aplicación. Se hace uso del estilo arquitectónico *Pipe and Filter* mediante pipelines recorridos por streams de datos sobre los cuales se ejecutan operadores filter, map, peek, reduce, collect. En los siguientes párrafos se presentan segmentos de código que muestran el uso de pipelines, los ejemplares mostrados realizan ejecución en paralelo (nótese el uso del método parallelStream).

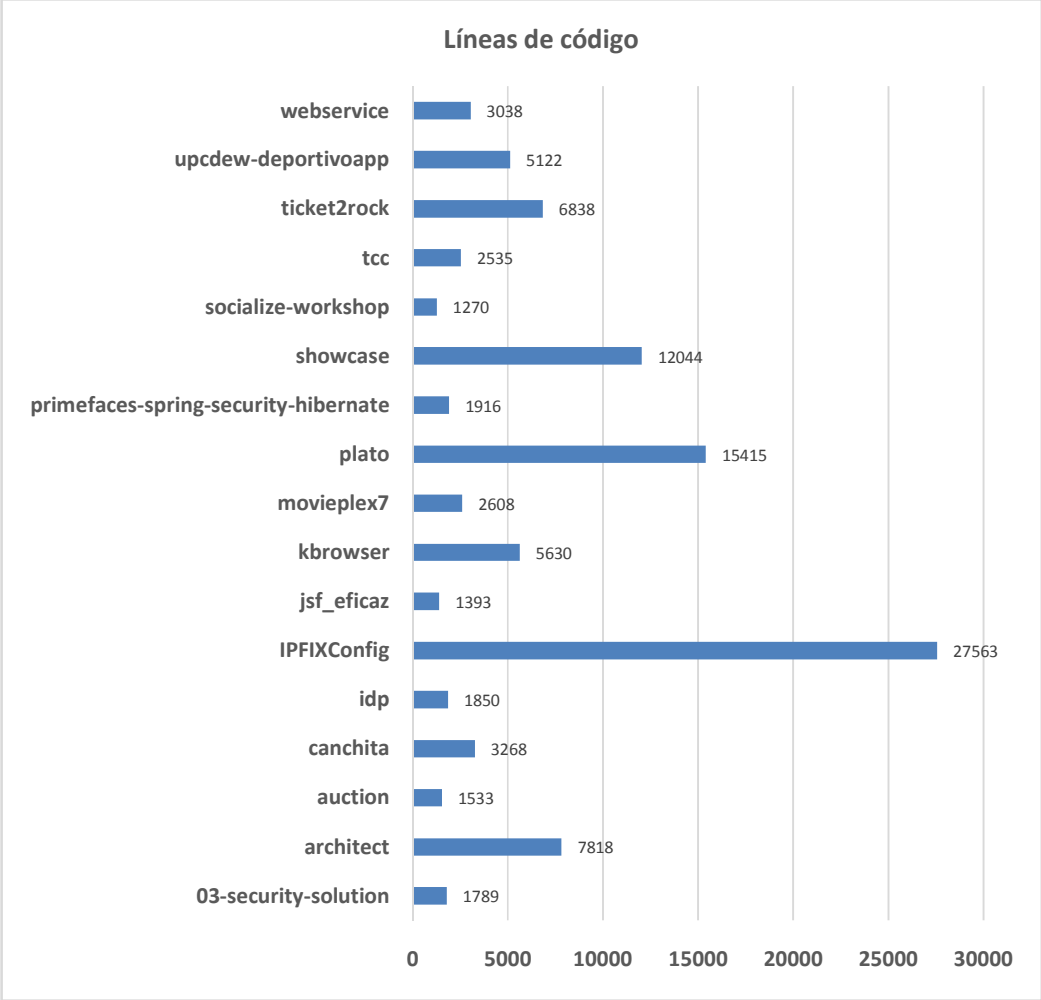

Figura 3. Número de líneas de código por proyecto de aplicación.

La siguiente figura muestra pipelines conformados por operadores filter y collect para el filtrado y colección de directorios y archivos:

```
List<Path> lstPathSource
        = FileUtil.generateListPath(pathSource);
// Filtrado de directorios y archivos
List<Path> lstSourceDir = lstPathSource.parallelStream()
        .filter(p -> (Files.isDirectory(p)))
        .collect(Collectors.toList());
List<Path> lstSourceFile = lstPathSource.parallelStream()
        .filter(p -> (!Files.isDirectory(p)))
        .collect(Collectors.toList());
```

Figura 4. Pipelines con operadores filter y collect.

Seguido se muestra pipelines conformados por operadores map y collect para el mapeo y colección de nodos y relaciones de grafos generados por el framework:

```
// Creación de estructura de paquetes en nodos
FPackage fNodePackage = new FPackage(pathSource);
List<Node> lstNode = lstSourceDir.parallelStream()
        .map(fNodePackage)
        .collect(StreamUtil.<Node>collectorList());
FRelationshipContainPackage fRelationshipPackage
        = new FRelationshipContainPackage();
lstNode.stream().map(fRelationshipPackage)
        .collect(Collectors.toList());
```

Figura 5. Pipelines con operadores map y collect.

La siguiente figura muestra pipelines conformados por operadores peek y collect, operadores empleados para la ejecución de operaciones de relacionamiento entre elementos del modelo y colección, respectivamente:

```
// Creación de relaciones entre clasificadores
lstClass.parallelStream()
    .peek(new CGeneralization(mapClass))
    .collect(Collectors.toList());
lstClass.parallelStream()
    .peek(new CRealization(mapClass))
    .collect(Collectors.toList());
lstClass.parallelStream()
    .peek(new CAssociation(mapClass))
    .collect(Collectors.toList());
lstClass.parallelStream()
    .peek(new CDependency(mapClass))
    .collect(Collectors.toList());
```

Figura 6. Pipelines con operadores peek y collect.

RESULTADOS

Durante las pruebas de las aplicaciones, se han tomado tiempos de cada paso ejecutado durante el análisis del código fuente. Se realizaron ejecuciones sucesivas sobre cada proyecto para luego obtener tiempos promedio medidos en milisegundos. Los cálculos fueron realizados tanto para streams secuenciales como para streams paralelos sobre cada proyecto de aplicación. Luego, se comparó los tiempos de ejecución promedio de cada paso del análisis, teniéndose que la generación de las relaciones entre elementos del modelo a extraer era el proceso que consumía mayor tiempo de ejecución, los resultados mostrados a continuación corresponden a este proceso. La siguiente figura muestra los tiempos promedio tomados para cada proyecto de aplicación, tanto en la implementación con streams secuenciales como para streams paralelos.

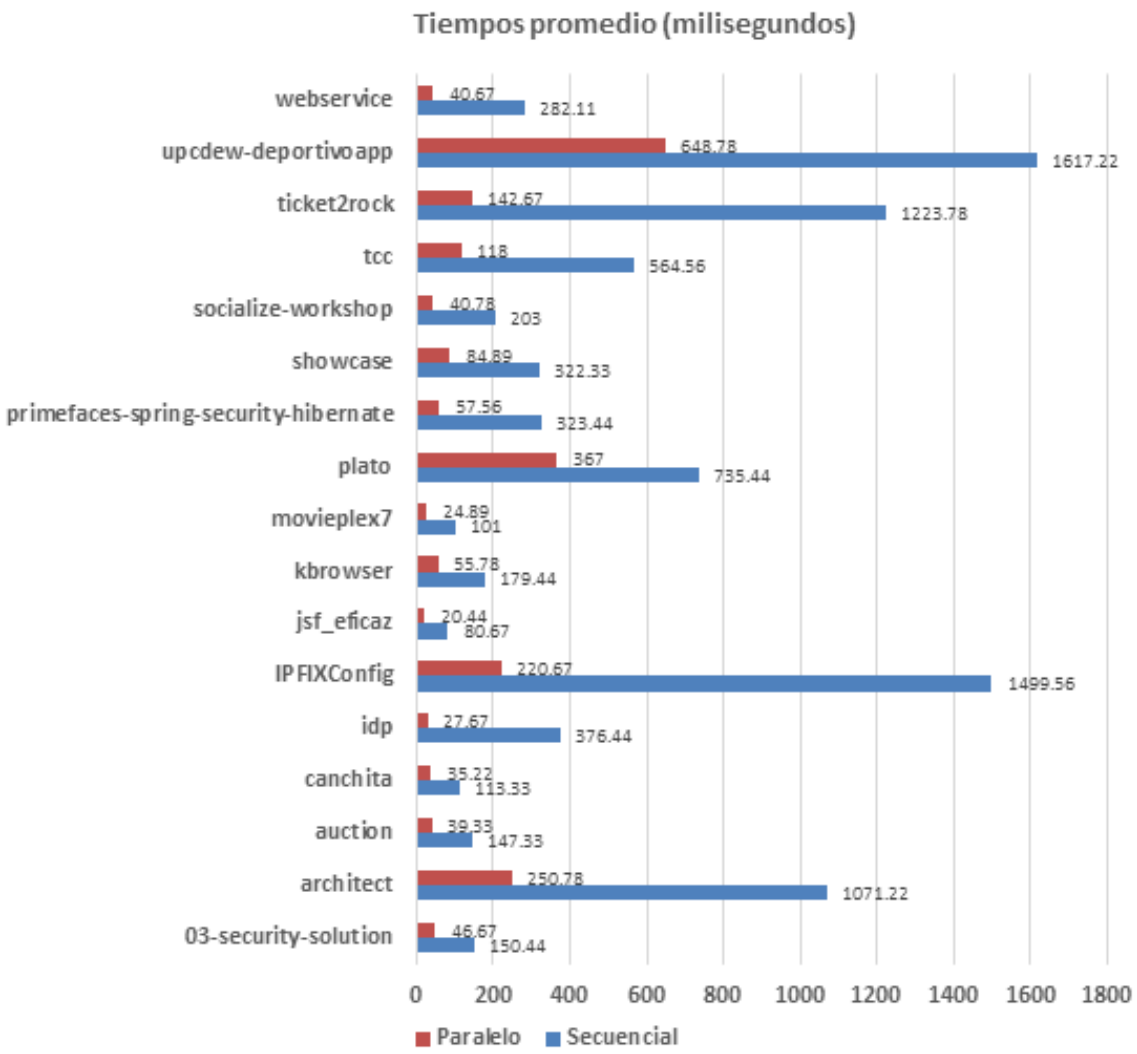

Figura 7. Tiempos para implementaciones secuenciales y paralelas.

ANÁLISIS Y DISCUSIÓN

Los resultados obtenidos muestran una marcada diferencia entre los tiempos de streams secuenciales y de streams paralelos, siendo estos últimos los que obtuvieron mejores tiempos. A fin de mostrar la mejora en forma uniforme sobre los proyectos, se calculó el porcentaje del tiempo de ejecución secuencial al que correspondería la diferencia entre el tiempo secuencial menos el tiempo de ejecución paralelo. Aquellos proyectos de aplicación que presentaron mejores resultados al emplear streams paralelos presentan un más alto porcentaje de diferencia en la siguiente figura.

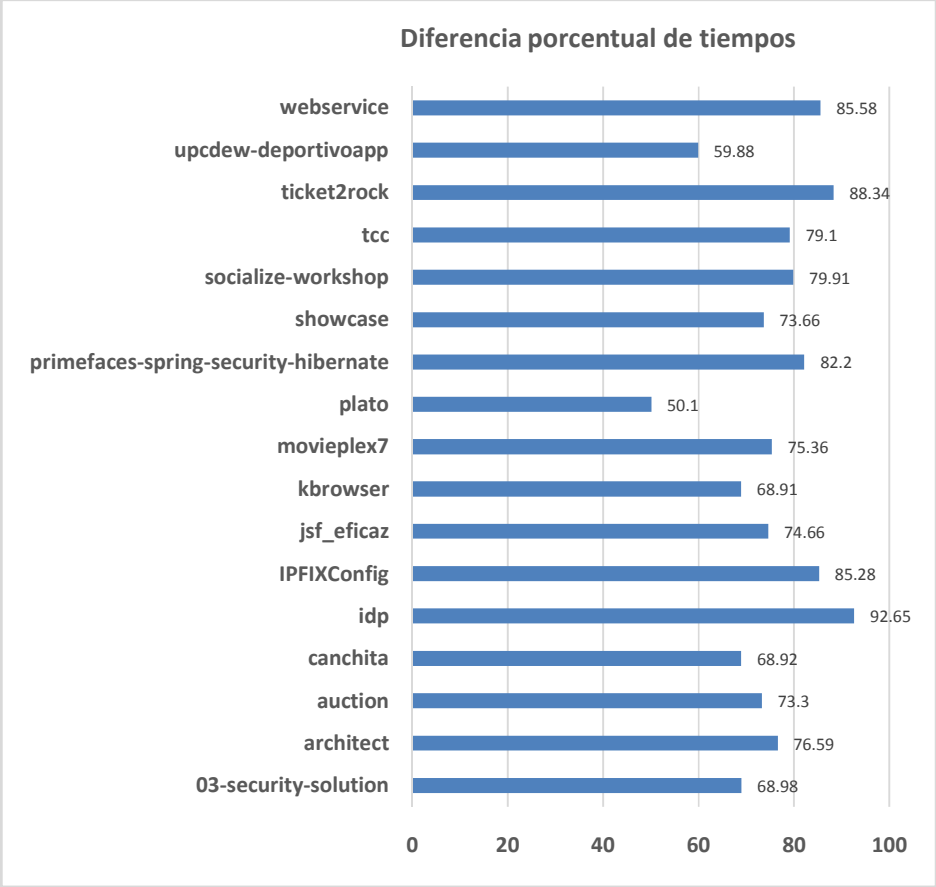

Figura 8. Diferencia porcentual entre tiempo secuencial y paralelo.

En general, en los diversos pasos de análisis de código las mayores diferencias de código se presentaron en proyectos de aplicación que tenían mayor cantidad de archivos y/o LOC. Para pasos del análisis que no consumían tiempo considerable, se encontró que el tiempo de ejecución secuencial podría ser mejor que el correspondiente en paralelo, lo cual se puede atribuir a la sobrecarga necesaria para aplicar paralelismo durante la ejecución, ello puede ser admisible y justificable al aplicar la ejecución en escenarios con mayor cantidad de archivos y LOC.

CONCLUSIONES

1. El uso de *Streams* y el estilo arquitectónico *Pipe and filter*, permitió reducir el tiempo empleado en el análisis de código fuente de volumen.
2. Por su capacidad de ejecución paralela e incremental, el uso de streams y el estilo *Pipe and Filter* puede ser determinante para la implementación de procesos eficientes en problemas inherentemente paralelizables.

REFERENCIAS BIBLIOGRÁFICAS

- Buğra Gedik. 2014. Partitioning functions for stateful data parallelism in stream processing. The VLDB Journal 23, 4 (August 2014), 517-539. DOI=10.1007/s00778-013-0335-9 **http://dx.doi.org/10.1007/s00778-013-0335-9**
- Kai Qian, Xiang Fu, LiXin Tao, Chong-wei Xu, and Jorge Diaz-Herrera. 2009. Software Architecture and Design Illuminated (1st ed.). Jones and Bartlett Publishers, Inc., USA.
- Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. ACM Comput. Surv. 46, 4, Article 46 (March 2014), 34 pages. DOI=10.1145/2528412 **http://doi.acm.org/10.1145/2528412**
- Richard Warburton. 2014. Java 8 lambdas. Sebastopol, CA: O'Reilly Media.
- Robert Liguori, and Patricia Liguori. 2014. Java 8 Pocket Guide. Sebastopol: O'Reilly & Associates.
- Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling techniques for elastic data stream processing. In Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14). ACM, New York, NY, USA, 318-321. DOI=10.1145/2611286.2611314 http://doi.acm.org/10.1145/2611286.2611314