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

Análisis dinámico de dependencias: en busca del paralelismo

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

José F. Cacherosky

Director: Diego Garbervetsky

Buenos Aires, 2026

ANÁLISIS DINÁMICO DE DEPENDENCIAS: EN BUSCA DE PARALELISMO

El análisis de dependencias es un componente fundamental para la identificación de oportunidades de paralelización en programas. Sin embargo, los enfoques estáticos suelen ser conservadores y no logran capturar con precisión el comportamiento real de ejecución, especialmente en presencia de estructuras dinámicas y dependencias dependientes de los datos de entrada.

En esta tesis se presenta un enfoque de análisis dinámico a nivel de bytecode para programas Java, junto con una herramienta que lo implementa. La herramienta instrumenta programas en tiempo de ejecución mediante un Java Agent, generando *traces* que registran accesos a memoria y eventos de control.

A partir de estos *traces*, se detectan y clasifican dependencias de datos (RAW, WAR, WAW, RAR), las cuales se asocian a estructuras de alto nivel como loops, permitiendo asistir al programador en la interpretación de dependencias dinámicas relevantes para potenciales oportunidades de paralelización.

El enfoque es evaluado mediante benchmarks representativos, analizando tanto la capacidad del sistema para capturar dependencias dinámicas como el overhead introducido por la instrumentación. Los resultados muestran que es posible identificar dependencias efectivas y vincularlas con decisiones de paralelización, manteniendo un balance razonable entre precisión y costo.

Dado que el análisis se basa en ejecuciones concretas, los resultados dependen de los datos de entrada y deben interpretarse como evidencia empírica del comportamiento observado. En conjunto, el trabajo contribuye a reducir la brecha entre el análisis de bajo nivel y la toma de decisiones de paralelización a nivel de código fuente.

Palabras clave: análisis dinámico, dependencias de datos, paralelización, bytecode Java, instrumentación, loops

DYNAMIC DEPENDENCE ANALYSIS: IN SEARCH OF PARALLELISM

Dependence analysis is a fundamental component for identifying parallelization opportunities in programs. However, static approaches are often conservative and fail to accurately capture actual runtime behavior, especially in the presence of dynamic structures and input-dependent behavior.
This thesis presents a dynamic dependence analysis approach at the bytecode level for Java programs, along with a tool that implements it. The tool instruments programs at runtime using a Java Agent, generating *traces* that record memory accesses and control-flow events. Based on these *traces*, data dependences (RAW, WAR, WAW, RAR) are detected and classified, and then associated with high-level structures such as loops, assisting programmers in the preliminary interpretation of dynamic dependencies under the observed execution paths.
The approach is evaluated using representative benchmarks, analyzing both its ability to capture dynamic dependences and the overhead introduced by instrumentation. The results show that it is possible to identify effective dependences and relate them to parallelization decisions, while maintaining a reasonable balance between precision and cost.
Since the analysis is based on concrete executions, the results depend on the input data and should be interpreted as empirical evidence of the observed behavior.
Overall, this work contributes to reducing the gap between low-level analysis and parallelization decisions at the source-code level.

Keywords: dynamic analysis, data dependences, parallelization, Java bytecode, instrumentation, loops

AGRADECIMIENTOS

Quiero agradecer a mi director de tesis por su guía y acompañamiento durante el desarrollo de este trabajo, y principalmente por su insistencia y perseverancia para que esta tesis se haga realidad.

También agradezco a mi familia, mi novia y amigos por el apoyo a lo largo de todo este proceso.

A mi hija y a mi madre. Y a toda mi familia, en el sentido amplio de la palabra.

Nada cansa tanto como la eterna sensación de tener colgada una tarea incompleta.
–William James

Índice general

1. INTRODUCCIÓN

We can only see a short distance ahead, but we can see plenty there that needs to be done.
— Alan Turing

1.1. Motivación

En los últimos años, la programación paralela se ha vuelto fundamental para el desarrollo de software de alto rendimiento. El desarrollo tecnológico de los microprocesadores ha alcanzado límites físicos en cuanto a la velocidad de procesamiento, y los nuevos procesadores incorporan cada vez mayor cantidad de núcleos. En este contexto, el aprovechamiento del paralelismo se vuelve fundamental para mejorar el rendimiento de las aplicaciones.

Sin embargo, el desarrollo de programas paralelos presenta desafíos importantes. Es considerablemente más complejo que la programación secuencial, tanto desde el punto de vista conceptual como en términos de correctitud y eficiencia.

Se han realizado numerosos intentos para automatizar la explotación del paralelismo. Existen dos enfoques principales. El primero consiste en el desarrollo de compiladores capaces de paralelizar programas automáticamente. Si bien se han propuesto técnicas sofisticadas, su aplicabilidad suele estar limitada por restricciones sobre los programas que pueden analizar.

El segundo enfoque delega la responsabilidad en el entorno de ejecución, en algunos casos con asistencia del compilador. En este caso, la eficiencia no está garantizada completamente y, en muchos casos, requiere soporte de hardware específico.

En este contexto, resulta razonable asumir que los programas paralelos siguen requiriendo en muchos casos intervención y adaptación manual por parte del programador. Por lo tanto, es fundamental contar con herramientas que asistan al programador en la identificación de oportunidades de paralelización.

En particular, este trabajo se enfoca en asistir al programador en la identificación de dependencias de datos dinámicas que limitan la paralelización de loops en programas Java.

En este contexto, surge la siguiente pregunta de investigación:

> ¿Es posible utilizar análisis dinámico de dependencias a nivel de bytecode para identificar oportunidades de paralelización en loops de programas Java de una manera útil e interpretable para el programador?

1.2. Descripción general

Esta tesis presenta un enfoque de análisis dinámico de dependencias a nivel de bytecode, junto con una herramienta que implementa dicho enfoque, con el objetivo de identificar oportunidades de paralelización en programas Java.

El enfoque consiste en analizar ejecuciones concretas de un programa, recolectando información sobre dependencias de datos dinámicas. Para ello, la herramienta instrumenta el bytecode de Java en tiempo de ejecución, generando *traces* que registran accesos a memoria y eventos relevantes.

Esto permite no solo detectar dependencias a bajo nivel, sino también interpretarlas en términos de estructuras del programa que resultan comprensibles y accionables para el programador.

El proceso completo puede resumirse conceptualmente en las siguientes etapas: instrumentación del programa, ejecución con generación de *traces*, y análisis posterior de dichos *traces* para la detección de dependencias.

Este pipeline se materializa en una arquitectura compuesta por dos componentes principales: un módulo de instrumentación encargado de recolectar información durante la ejecución, y un módulo de análisis que procesa los *traces* generados para detectar dependencias dinámicas. La organización de estos componentes y su interacción se describen en detalle en el Capítulo 3.

Si bien existen herramientas de análisis dinámico orientadas a profiling y detección de errores de concurrencia, la identificación de oportunidades de paralelización a partir de dependencias dinámicas continúa siendo un desafío.

1.2.1. Pipeline del enfoque

El proceso completo se divide en dos etapas claramente diferenciadas: una etapa online de instrumentación y ejecución, y una etapa offline de análisis de los *traces* generados.

- **Instrumentación:** el programa Java es instrumentado a nivel de bytecode utilizando un Java Agent, con el objetivo de recolectar eventos relevantes durante su ejecución (accesos a memoria, entradas/salidas de métodos, comportamiento de loops).
- **Análisis:** a partir de los *traces* generados, se realiza un análisis dinámico para detectar dependencias entre accesos a memoria y asociarlas a estructuras de control como loops.

Esta separación permite desacoplar la recolección de información del análisis, facilitando la extensibilidad del enfoque y la reutilización de los *traces* generados.

A diferencia de enfoques estáticos tradicionales, este trabajo se basa en el análisis dinámico de ejecuciones concretas, permitiendo capturar comportamientos reales del programa que no siempre pueden ser inferidos mediante análisis estático.

A partir de estos *traces*, se detectan dependencias entre instrucciones, permitiendo identificar porciones del código que podrían ser paralelizadas, así como aquellas que no lo son debido a la presencia de dependencias.

Una característica distintiva de este trabajo es la combinación de información de bajo nivel (accesos a memoria a nivel de bytecode) con un análisis a nivel de estructuras de alto nivel como loops. Esta integración no es habitual en herramientas de análisis, y permite mapear dependencias dinámicas sobre estructuras comprensibles para el programador.

Dado que el análisis se basa en ejecuciones concretas, los resultados dependen de los datos de entrada utilizados. Por lo tanto, las sugerencias generadas por la herramienta deben ser interpretadas y validadas por el programador.

Un aspecto relevante del diseño es la consideración de la performance. La instrumentación y el análisis dinámico generan un volumen considerable de información, por lo que se tomaron decisiones específicas para manejar eficientemente la generación, almacenamiento y procesamiento de los *traces*.

En este sentido, el objetivo no es reemplazar al programador, sino asistirlo en la toma de decisiones informadas sobre paralelización.

1.2.2. Contribuciones

Este trabajo propone un enfoque de análisis dinámico de dependencias a nivel de bytecode orientado a asistir al programador en la identificación de oportunidades de paralelización, mediante la asociación de dependencias dinámicas con estructuras de alto nivel como loops.

Las contribuciones principales incluyen:

- Un enfoque de instrumentación y análisis dinámico aplicado a programas Java
- Un modelo que permite relacionar dependencias dinámicas con estructuras de loops
- Una herramienta funcional para la detección de dependencias en ejecución
- Una evaluación experimental mediante benchmarks representativos, analizando la capacidad del enfoque para capturar dependencias dinámicas y su impacto en términos de overhead

1.3. Estado del arte

El análisis de dependencias ha sido ampliamente estudiado en el contexto de la paralelización de programas. Tradicionalmente, este problema ha sido abordado mediante técnicas estáticas, que intentan inferir dependencias a partir del código fuente. Sin embargo, estos enfoques suelen ser conservadores, especialmente en presencia de aliasing, estructuras dinámicas y dependencias dependientes de los datos de entrada.

Como alternativa, el análisis dinámico permite observar el comportamiento real del programa durante su ejecución, capturando dependencias efectivas a partir de accesos a memoria. Este enfoque ha sido explorado en distintos trabajos, tanto en el contexto de paralelización como en profiling y análisis de concurrencia.

A pesar de estos avances, la mayoría de las herramientas actuales no vincula de forma directa las dependencias detectadas con estructuras de alto nivel del programa, como los loops, lo cual limita su utilidad para asistir al programador en la identificación de oportunidades de paralelización.

Un análisis detallado del estado del arte y de la evolución de estas técnicas se presenta en el Capítulo 6.

1.4. Estructura de la tesis

Esta tesis se organiza de la siguiente manera.

El Capítulo 2 presenta la arquitectura general de la herramienta, describiendo sus principales componentes y el flujo de procesamiento desde la ejecución del programa hasta la detección de dependencias dinámicas. Se introduce la separación entre la etapa de instrumentación y ejecución (online) y la etapa de análisis (offline), así como la interacción entre ambos componentes.

El Capítulo 3 describe el proceso de recolección de información de ejecución mediante instrumentación de bytecode. Se detalla el diseño e implementación del *TracerTool*, encargado de generar los *traces* de ejecución a partir de eventos relevantes como accesos a memoria, entradas y salidas de métodos y ejecución de loops.

El Capítulo 4 presenta el análisis de los *traces* generados. Se introduce el modelo de análisis dinámico de dependencias, incluyendo las estructuras principales utilizadas —como el árbol de ejecución, la tabla de accesos a objetos y las tablas de dependencias— y los algoritmos empleados para detectar dependencias y evaluar oportunidades de paralelización.

El Capítulo 5 valida el enfoque propuesto mediante ejemplos representativos y benchmarks, analizando tanto la correctitud del análisis como su costo computacional en términos de tiempo, memoria y volumen de datos generados.

El Capítulo 6 presenta una revisión del estado del arte en análisis dinámico de dependencias y paralelización automática, contextualizando el enfoque propuesto dentro de la evolución del área.

Finalmente, el Capítulo 7 expone las conclusiones del trabajo y propone posibles líneas de investigación futura.

En síntesis, esta tesis propone un enfoque de análisis dinámico de dependencias a nivel de bytecode que busca asistir al programador en la identificación de oportunidades de paralelización, combinando precisión a bajo nivel con una interpretación estructural a nivel de loops.

Para llevar este enfoque a la práctica, es necesario contar con una arquitectura que permita recolectar y procesar información de ejecución de manera sistemática. En el siguiente capítulo se presenta dicha arquitectura y sus componentes principales.

Este trabajo muestra que es posible vincular el análisis dinámico de bajo nivel con decisiones concretas de paralelización, contribuyendo a reducir la brecha entre la ejecución real del programa y su optimización a nivel de código fuente.

2. ARQUITECTURA DE LA HERRAMIENTA

Abstraction is the key to building complex systems.
— Barbara Liskov

En este trabajo, se parte de la ejecución del programa para construir abstracciones que permiten interpretar dependencias de datos en términos de estructuras del programa como loops.

Este capítulo presenta una visión integrada de la arquitectura de la herramienta desarrollada, describiendo sus principales componentes y el flujo de procesamiento desde la ejecución del programa hasta la detección de dependencias dinámicas.

A diferencia de los capítulos siguientes, donde se detallan los aspectos de implementación, aquí se adopta una perspectiva de alto nivel con el objetivo de facilitar la comprensión global del sistema y de las decisiones de diseño adoptadas.

2.1. Visión general

El enfoque propuesto se basa en un análisis dinámico de dependencias a partir de ejecuciones concretas de programas Java. Para ello, la herramienta se organiza como un pipeline de procesamiento dividido en dos etapas principales:

- **Etapa de instrumentación y ejecución (online):** el programa es instrumentado a nivel de bytecode y ejecutado, generando un *trace* de eventos que registra accesos a memoria y estructuras de control.
- **Etapa de análisis (offline):** el *trace* generado es procesado posteriormente para reconstruir la ejecución del programa y detectar dependencias dinámicas.

Esta separación permite desacoplar la recolección de información del análisis, facilitando la reutilización de *traces* y la extensibilidad del sistema.

2.2. Pipeline de procesamiento

La Figura 2.1 muestra el flujo general del sistema.
El proceso completo se compone de las siguientes etapas:

1. **Programa Java:** punto de partida del análisis.
2. **Instrumentación:** el bytecode es modificado dinámicamente mediante un Java Agent, insertando instrucciones que registran eventos relevantes durante la ejecución.
3. **Ejecución y generación de trace:** el programa instrumentado se ejecuta, generando un archivo de *trace* (`tracer.out`) que contiene la secuencia de eventos observados.

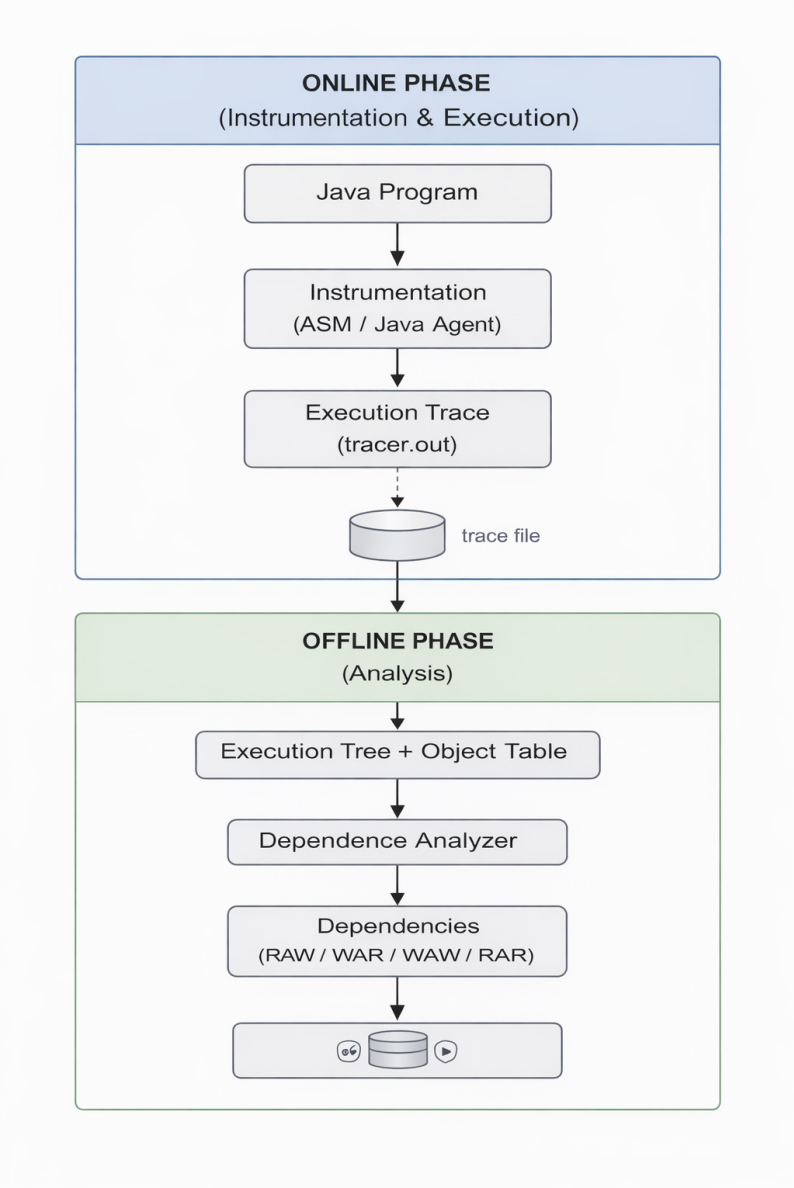

Fig. 2.1: Pipeline de la herramienta, dividido en etapa online (instrumentación y ejecución) y etapa offline (análisis de dependencias).

4. **Reconstrucción de estructuras dinámicas:** a partir del *trace*, se construyen estructuras como el *execution tree* y la *object table*, que representan el contexto de ejecución y los accesos a memoria.

5. **Análisis de dependencias:** se procesan las estructuras reconstruidas para detectar y clasificar dependencias dinámicas.

2.3. Separación entre etapas online y offline

Una de las decisiones de diseño más relevantes del enfoque es la separación entre la etapa de ejecución instrumentada y el análisis posterior.

En la etapa online se prioriza la recolección eficiente de información, registrando eventos relevantes con el menor overhead posible. En esta etapa no se realiza análisis complejo, sino que se genera un *trace* que captura el comportamiento dinámico del programa.

En la etapa offline, en cambio, se realiza el análisis intensivo sobre los datos recolectados. Esto incluye la reconstrucción del contexto de ejecución y la detección de dependencias.

Esta separación presenta múltiples ventajas:

- Permite reutilizar el mismo *trace* para distintos análisis
- Reduce la complejidad de la instrumentación
- Facilita la evolución independiente de cada componente
- Permite trasladar el costo computacional a una etapa posterior

Sin embargo, también introduce desafíos asociados al volumen de datos generado, como se discute en el Capítulo 5.

2.4. Componentes principales

La arquitectura de la herramienta se organiza en dos componentes principales, alineados con las etapas del pipeline.

2.4.1. TracerTool

El *TracerTool* es responsable de la instrumentación del programa y de la generación del *trace* de ejecución.

Este componente opera como un Java Agent que intercepta la carga de clases en la JVM y modifica su bytecode utilizando el framework ASM. Durante este proceso, se insertan instrucciones que permiten registrar eventos como accesos a memoria, entradas y salidas de métodos y comportamiento de loops.

El resultado de esta etapa es un archivo de *trace* que contiene una secuencia ordenada de eventos que reflejan la ejecución del programa.

Los detalles de implementación de este componente se presentan en el Capítulo 3.

2.4.2. Dependence Analyzer

El *Dependence Analyzer* es el encargado de procesar el *trace* generado y detectar dependencias dinámicas.

Este componente reconstruye estructuras de ejecución, como el *execution tree*, que permite representar el contexto dinámico del programa, y la *object table*, que almacena información sobre accesos recientes a memoria.

A partir de estas estructuras, el analizador identifica dependencias entre accesos a memoria y las clasifica en categorías como RAW (Read-After-Write), WAR (Write-After-Read), WAW (Write-After-Write) y RAR (Read-After-Read).

Los detalles de este proceso se describen en el Capítulo 4.

2.5. Flujo de datos

El flujo de datos entre los componentes puede resumirse de la siguiente manera:

- El programa instrumentado genera un *trace* de ejecución
- El *trace* es procesado para reconstruir el contexto dinámico
- Sobre esta representación se detectan dependencias
- Los resultados se asocian a estructuras del programa, como loops

Este flujo refleja la transformación progresiva de información desde eventos de bajo nivel (accesos a memoria) hacia conocimiento estructurado útil para el programador.

2.6. Discusión

La arquitectura propuesta refleja un diseño orientado a la separación de responsabilidades y a la extensibilidad del sistema.

Por un lado, el uso de instrumentación dinámica permite capturar el comportamiento real del programa con gran precisión. Por otro, la etapa de análisis permite interpretar esta información en términos de estructuras de alto nivel.

Esta combinación constituye uno de los aportes centrales del trabajo, ya que permite vincular información de bajo nivel con decisiones de paralelización a nivel de código fuente.

No obstante, esta arquitectura también implica costos significativos en términos de tiempo de ejecución, consumo de memoria y volumen de datos generados, aspectos que se analizan en detalle en el capítulo siguiente.

En particular, el primer paso del pipeline consiste en la recolección de información de ejecución mediante instrumentación de bytecode. En el siguiente capítulo se describe en detalle el diseño e implementación del componente encargado de esta tarea, el *TracerTool*.

3. CÓMO OBTENER LA INFORMACIÓN

There is a difference between knowing the path and walking the path.
— Morpheus, The Matrix

Una vez obtenida la información de ejecución en forma de *traces*, el siguiente desafío consiste en interpretarla de manera estructurada para detectar dependencias dinámicas.

En este capítulo se describe cómo obtener la información necesaria a partir de la ejecución de un programa. En particular, se presenta el proceso de instrumentación de bytecode y la generación de *traces* que registran eventos relevantes durante la ejecución.

Esta información constituye la base para el análisis dinámico de dependencias que se presenta en el capítulo siguiente.

3.1. Instrumentación y generación de *traces*

El objetivo de esta etapa es instrumentar el programa Java de manera tal que, durante su ejecución, se genere un *trace* con información suficiente para reconstruir el contexto dinámico de ejecución y detectar dependencias entre accesos a memoria.

Esta información, correctamente recolectada, constituye la *materia prima* para el análisis de *traces* que se presenta en el capítulo siguiente.

A continuación se presenta la idea general del enfoque, junto con el marco teórico necesario. Luego se describen los detalles de implementación utilizados.

3.2. Instrumentación de código

La instrumentación de código consiste en modificar un programa con el objetivo de insertar instrucciones adicionales que permitan observar su comportamiento durante la ejecución.

En el caso de programas escritos en Java, el código fuente es compilado a *Java Bytecode*, que es el lenguaje de bajo nivel ejecutado por la Java Virtual Machine (JVM). Este mismo modelo es utilizado por otros lenguajes que corren sobre la JVM, como Scala, Kotlin o Groovy.

Cada programa Java tiene una representación equivalente en bytecode, lo que permite realizar transformaciones directamente sobre este nivel.

Sea un programa Java para el cual se desea registrar eventos durante su ejecución (al que llamaremos *Programa X*). Para lograrlo, se instrumenta su bytecode de manera tal que se inserten instrucciones adicionales exactamente en los puntos donde ocurren los eventos de interés.

De este modo, la instrumentación introduce instrucciones de bytecode dentro del código original del programa, con el objetivo de registrar información relacionada con:

- accesos a memoria (objetos y arrays)
- entradas y salidas de métodos

- comportamiento de loops (inicio, iteraciones y finalización)
- ejecución de bloques básicos

Estas modificaciones permiten obtener un registro detallado de la ejecución del programa sin alterar su lógica funcional.

3.2.1. Paquete de instrumentación de Java

La plataforma Java provee un mecanismo estándar para instrumentación mediante el paquete `java.lang.instrument`. Este permite interceptar la carga de clases en la JVM y modificar su bytecode antes de que sean ejecutadas.

A través de este mecanismo, es posible aplicar transformaciones dinámicas al programa, sin necesidad de modificar su código fuente ni recompilarlo.

3.2.2. Java Agent

Un *Java Agent* es un componente que se ejecuta junto con la JVM y que permite aplicar instrumentación sobre las clases en tiempo de carga.

El agente se registra al iniciar la aplicación y recibe cada clase antes de que sea definida por la JVM. En ese momento, puede modificar su bytecode para insertar las instrucciones necesarias para el trazado de eventos.

Este enfoque resulta especialmente útil ya que permite instrumentar programas existentes de manera transparente, sin necesidad de cambios en su código original.

3.2.3. Framework ASM

La instrumentación se realiza utilizando la librería ASM, que permite modificar el bytecode de clases Java en tiempo de carga, insertando instrucciones adicionales que registran eventos relevantes durante la ejecución.

ASM es una biblioteca ampliamente utilizada para la generación, transformación y análisis de bytecode Java. Se caracteriza por su eficiencia y por ofrecer un nivel de abstracción adecuado para trabajar directamente sobre instrucciones de bajo nivel.

Mediante ASM es posible recorrer y modificar la estructura de clases, métodos e instrucciones, lo cual resulta fundamental para insertar los puntos de instrumentación requeridos por el enfoque propuesto.

3.2.4. Eventos generados

Durante la ejecución del programa instrumentado, se generan distintos tipos de eventos que son registrados en el trace.

Estos eventos representan los puntos relevantes de la ejecución y permiten reconstruir posteriormente el comportamiento dinámico del programa.

En particular, se registran:

- Entrada y salida de métodos
- Entrada, iteración y salida de loops
- Accesos a memoria (lectura y escritura)

Cada uno de estos eventos es codificado en el *trace* de ejecución (traza) mediante identificadores específicos, lo que permite reconstruir la estructura de ejecución del programa y asociar los accesos a memoria con su contexto dinámico.

Esta información constituye la base para el análisis posterior de los *traces*.

Una vez definidos los eventos que se registran durante la ejecución, es necesario establecer cómo se representan y almacenan en el *trace* generado.

3.3. Representación del trace

Los eventos generados durante la ejecución del programa instrumentado son registrados en un archivo de salida, típicamente denominado `tracer.out`.

Este archivo contiene una secuencia de eventos codificados que representan la ejecución del programa, incluyendo llamadas a métodos, ejecución de loops y accesos a memoria.

Cada línea del *trace* corresponde a un evento específico, identificado mediante un código abreviado.

Por ejemplo:

```
MEN 10
AAS 1163904012-0
AAL 1163904012-0
LIT L6
LEX L6
```

Donde:

- `MEN`: entrada a método (Method Entry)
- `AAS`: acceso a memoria de escritura (Array Store)
- `AAL`: acceso a memoria de lectura (Array Load)
- `LIT`: iteración de loop
- `LEX`: salida de loop

Este formato compacto permite registrar la ejecución del programa con bajo overhead, manteniendo la información necesaria para reconstruir la estructura dinámica de ejecución.

El procesamiento de este *trace* permite realizar el análisis de dependencias dinámicas a partir de la información recolectada.

Una vez definido el formato del *trace*, es posible reconstruir a partir de él una representación estructurada de la ejecución del programa.

3.4. Reconstrucción de la ejecución

A partir del *trace* generado durante la ejecución del programa instrumentado, es posible reconstruir una representación estructurada del comportamiento dinámico del programa.

Esta reconstrucción se basa en interpretar la secuencia de eventos registrados en el *trace* y organizarlos en una estructura jerárquica que refleje el contexto de ejecución.

En particular, los eventos de entrada y salida de métodos permiten reconstruir la pila de llamadas, mientras que los eventos asociados a loops permiten identificar su estructura y sus iteraciones.

Asimismo, los accesos a memoria registrados en el *trace* se asocian al contexto dinámico correspondiente, definido por el método activo y las iteraciones de loops en curso.

De esta manera, cada acceso a memoria puede ubicarse en una posición única dentro de la ejecución del programa.

Esta representación estructurada permite organizar la información del *trace* de manera coherente, preparando el análisis que se describe en el capítulo siguiente.

3.5. TracerTool

Para lograr el objetivo de generar un *trace* de los eventos de interés, respetando el orden en que ocurren durante la ejecución, se desarrolló un agente de Java encargado de instrumentar el bytecode de cualquier programa secuencial que se desee analizar. Este agente constituye la herramienta principal de este trabajo, denominada *TracerTool*.

TracerTool es un Java Agent que utiliza el paquete `java.lang.instrument` para interceptar la carga de clases en la JVM y aplicar transformaciones sobre su bytecode de manera dinámica, generalmente bajo demanda. En particular, la transformación se realiza inmediatamente antes de que cada clase sea definida por la JVM.

Es posible definir qué clases serán instrumentadas, lo cual permite evitar la modificación de clases internas de Java. Esta decisión depende del objetivo del análisis y del tipo de *trace* que se desea obtener.

Cabe destacar que existen clases que son cargadas en las etapas iniciales del arranque de la JVM. Estas clases no son interceptadas por el agente y, por lo tanto, no pueden ser instrumentadas mediante este enfoque. Si bien es posible extender *TracerTool* para cubrir estos casos (por ejemplo, redefiniendo el *ClassLoader*), esto queda fuera del alcance del presente trabajo, ya que el objetivo principal no es analizar clases internas de la JVM.

Durante el proceso de transformación, *TracerTool* utiliza el framework ASM (versión 3.3.1) para manipular el bytecode de cada clase. ASM es una biblioteca ampliamente utilizada para la generación, análisis y transformación de bytecode Java, destacándose por su eficiencia y simplicidad de uso. En particular, se emplea la API de tipo árbol (Tree API).

Para cada clase, se construye una representación en forma de árbol a partir de su bytecode. Cada método es representado como una lista de nodos que incluyen instrucciones, etiquetas, números de línea, entre otros elementos. Para más detalles sobre esta estructura, puede consultarse la documentación oficial de ASM[1].

Internamente, *TracerTool* utiliza un esquema de identificación único para clases, métodos y campos. Cada clase posee un identificador numérico único (*classId*). De manera similar, cada par nombre de método–firma de método se identifica mediante un *methodId*, lo cual permite distinguir correctamente métodos sobrecargados. Asimismo, cada campo instrumentado posee un *fieldId* único.

Por ejemplo, una clase podría identificarse como C12, un método como M32 y un campo como F4. Este esquema es gestionado por un componente denominado *TraceManagerHelper*, el cual utiliza estructuras de tipo mapa para mantener estas correspondencias. Esta información forma parte de los archivos de salida generados por la herramienta.

[1] ASM Guide: http://download.forge.objectweb.org/asm/asm-guide.pdf

Durante la instrumentación de cada método, se analiza en detalle cada instrucción de bytecode con el objetivo de detectar:

- accesos a objetos
- entradas y salidas de métodos
- comportamiento de loops

Una vez detectados estos eventos, se insertan llamadas a métodos estáticos de la clase *TraceManager*, que es responsable de registrar los eventos en el *trace*.

Este proceso requiere un conocimiento detallado del conjunto de instrucciones de Java Bytecode[2], así como de su interacción con la pila de la JVM, ya que es necesario manipularla correctamente para insertar las llamadas adicionales sin alterar el comportamiento del programa original.

3.6. Entrada y salida de métodos

La detección de la entrada a un método es directa: antes de la primera instrucción de cada método, se inserta una llamada a *TraceManager.traceMethodEntry(...)*.

En cuanto a la salida de métodos, antes de cada instrucción de tipo `xRETURN` presente en el bytecode original, se inserta una llamada a *TraceManager.traceXReturn(...)*. La letra `x` representa los distintos tipos de retorno posibles (por ejemplo, `D` para *double*, `F` para *float*, `A` para referencias, etc.).

Los parámetros utilizados en estas llamadas incluyen el *classId* y el *methodId*, lo cual permite identificar de manera única el contexto de ejecución.

Adicionalmente, una excepción no capturada dentro del método también representa una salida del mismo. Para cubrir este caso, se instrumenta el mecanismo de manejo de excepciones, de modo que se registre la salida del método cuando una excepción es propagada. En este contexto, los eventos de entrada a bloques básicos (descritos más adelante) resultan fundamentales para detectar correctamente estas situaciones.

3.7. Acceso a objetos

El análisis de dependencias requiere registrar todos los accesos a memoria. En el modelo de ejecución de la JVM, esto se traduce en accesos a objetos y a arrays.

Para detectar estos accesos, se inspeccionan las siguientes instrucciones de bytecode:

- `GETFIELD`
- `PUTFIELD`
- `GETSTATIC`
- `PUTSTATIC`
- `xALOAD` (lectura desde arrays)

[2] Java Bytecode Instructions: `http://java.sun.com/docs/books/vmspec/2nd-edition/html/Instructions2.doc.html`

- `xASTORE` (escritura en arrays)

En cada uno de estos casos, se inserta una llamada al método correspondiente de *TraceManager* (por ejemplo, *traceGetField(...)* o *tracePutField(...)*).

Los parámetros incluyen, según corresponda, el *fieldId*, la referencia al objeto y el índice en el caso de arrays. Esta información es suficiente para identificar de manera única cada acceso a memoria.

3.8. Detección de loops y construcción del Loop Nesting Tree

La detección de loops requiere la construcción de diversas estructuras en múltiples etapas, para cada método analizado. Se sigue un enfoque clásico basado en la bibliografía[3], al cual se le incorporan extensiones para contemplar correctamente estructuras de control asociadas al manejo de excepciones.

El proceso se describe a continuación:

1. **Construcción del Control Flow Graph (CFG).** Para cada método se construye su correspondiente *Control Flow Graph* (CFG), el cual representa la estructura de control del programa.

   El CFG es un grafo dirigido con raíz, cuyos nodos son *Basic Blocks*. Un *Basic Block* es una secuencia de instrucciones con un único punto de entrada y un único punto de salida.

   Se agregan además nodos especiales: *Entry Block*, *Exit Block* y *Abnormal Termination Block*.

   a) **Identificación de líderes.** Se consideran líderes aquellas instrucciones que cumplen alguna de las siguientes condiciones:

      1) Punto de entrada del método
      2) Destino de un salto
      3) Instrucción inmediatamente posterior a un salto o retorno
      4) Primera instrucción de un bloque *catch* o *finally*
      5) Instrucción inmediatamente posterior a un bloque *catch* o *finally*

   b) **Construcción de Basic Blocks.** A partir de los líderes identificados, se construyen los *Basic Blocks* agrupando las instrucciones entre líderes consecutivos.

   c) **Construcción de aristas del CFG.** Se agregan aristas entre *Basic Blocks* según el flujo de control del programa. Además:

      - Se agrega una arista desde *Entry Block* al primer *Basic Block* del método
      - Se agregan aristas desde bloques finales hacia *Exit Block*

2. **Cálculo de dominadores.** Se define que un nodo d domina a un nodo i si todo camino desde *Entry Block* hasta i pasa por d.

   Los dominadores se representan mediante una estructura del tipo:

   ```
   Map(BasicBlock, Set(BasicBlock))
   ```

[3] Steven Muchnick, *Advanced Compiler Design and Implementation*, Capítulo 7: Control-Flow Analysis

3. **Detección de back edges.** Una arista es una *back edge* si su nodo destino domina al nodo origen.

4. **Cálculo de natural loops.** Dada una *back edge*, el *natural loop* está compuesto por el nodo destino (loop header) y todos los nodos desde los cuales se puede alcanzar el origen sin pasar por el header.

5. **Fusión de natural loops.** Si múltiples *natural loops* comparten el mismo header, se combinan en una única estructura.

6. **Construcción de loops.** A partir de los *natural loops*, se crean instancias de la clase *Loop*, con identificadores únicos dentro de cada método.

7. **Construcción del Loop Nesting Tree.** Se construye una estructura jerárquica que representa la relación de anidamiento entre loops.

   Cada método se modela como un loop artificial (*MethodFakeLoop*) con identificador 0.

8. **Persistencia de la estructura.** La estructura se almacena mediante:

   `Map(Integer, Map(Integer, MethodFakeLoop))`

9. **Generación de reglas de trazado.** Se genera una estructura que permite determinar, para cada transición entre *Basic Blocks*, si corresponde a:

   - entrada a un loop
   - iteración de un loop
   - salida de un loop

10. **Persistencia de reglas.** Las reglas generadas se almacenan para su uso en tiempo de ejecución.

3.9. Entrada a Basic Blocks

TracerTool permite registrar la ejecución de *Basic Blocks* mediante el flag *TraceManager.tracingBasicBlocks*.

Antes de la ejecución de cada *Basic Block*, se inserta una llamada al método *TraceManager.traceLoop*, el cual:

- Registra el *Basic Block* a ejecutar (si el flag está habilitado)
- Determina si la transición corresponde a entrada, iteración o salida de un loop

3.10. Profiling

En este punto hemos descripto cómo obtener la información necesaria para generar el *trace* de la ejecución de un programa Java. Dicho *trace* incluye eventos relevantes como *method entry* y *method exit*, accesos a objetos, eventos de loops (entrada, salida e iteración) y también entradas a *Basic Blocks*.

Es importante notar que el volumen de información generado por este *trace* puede ser muy grande. Incluso para programas pequeños con pocas iteraciones, el tamaño puede alcanzar cientos de megabytes. Por esta razón, se puso un fuerte énfasis en optimizar tanto la complejidad de los algoritmos como el consumo de memoria. En particular, se realizaron optimizaciones iterativas sobre las estructuras de datos utilizadas, midiendo el impacto en performance en cada paso. El objetivo principal fue reducir la complejidad, mejorar los tiempos de acceso y minimizar el consumo de memoria.

A continuación se describen en detalle las decisiones de diseño adoptadas para el caso del *trace* de eventos de loops.

Para cada método se construye su correspondiente *Control Flow Graph* (CFG). Una posible estrategia para determinar, en tiempo de ejecución, si una transición corresponde a entrada, salida o iteración de un loop, sería recorrer el CFG en cada paso de la ejecución. Sin embargo, este enfoque introduce un overhead significativo: requiere mantener los CFGs en memoria, recorrerlos constantemente y gestionar estructuras auxiliares como stacks de ejecución por método.

Para evitar este costo en tiempo de ejecución, se adoptó un enfoque basado en precómputo. En particular, todas las posibles acciones asociadas a transiciones entre *Basic Blocks* se calculan en tiempo de carga (*loading time*). Para cada transición posible dentro del CFG de un método, se determina si corresponde a una entrada a un loop, una iteración o una salida. Esta información se almacena en una estructura denominada *loopTraceRulesMap*.

De este modo, durante la ejecución, la detección del tipo de evento asociado a una transición entre *Basic Blocks* se reduce a una simple consulta en esta estructura, logrando tiempo constante $O(1)$. Esto elimina completamente la necesidad de recorrer el CFG en tiempo de ejecución.

Una vez definida la estructura *loopTraceRulesMap*, se analizó su comportamiento en términos de performance. Esta estructura está implementada utilizando *HashMap* y *ArrayList*, priorizando accesos rápidos de lectura. En una primera versión, se utilizaron claves de tipo *String*. Sin embargo, al reemplazarlas por identificadores enteros (*Integer*), se obtuvieron mejoras significativas en los tiempos de acceso. Esto permitió eliminar completamente el uso de *String* como clave, evitando comparaciones costosas y reduciendo el overhead general del sistema.

Adicionalmente, se diseñó un mecanismo para asignar identificadores únicos a cada *Basic Block* del programa. Estos identificadores son consecutivos dentro de cada método, lo que implica que los *Basic Blocks* de un mismo método se encuentran dentro de un rango continuo de valores.

Esta propiedad permite determinar de manera eficiente si un *Basic Block* pertenece a un método simplemente verificando si su identificador se encuentra dentro de un determinado rango. Esta optimización resulta particularmente útil en presencia de excepciones no capturadas.

En efecto, si el *Basic Block* actual no pertenece al rango asociado al último método ejecutado, se puede inferir que ocurrió una excepción no manejada que transfirió el control a otro contexto. Esta verificación puede realizarse en tiempo constante $O(1)$, lo cual contribuye a mantener bajo el overhead del sistema de trazado.

En conjunto, estas decisiones de diseño permiten reducir significativamente el costo en tiempo de ejecución del proceso de generación de *traces*, trasladando la mayor parte del trabajo al momento de carga de las clases.

La información recolectada mediante este proceso constituye la base para el análisis dinámico de dependencias, que se presenta en el siguiente capítulo, donde se procesan los *traces* para detectar y clasificar dependencias entre accesos a memoria.

4. ANÁLISIS DE *TRACES* DE EJECUCIÓN

What we observe is not nature itself, but nature exposed to our method of questioning.
— Werner Heisenberg

Sobre esta información, el presente capítulo introduce el modelo de análisis dinámico de dependencias que constituye el núcleo conceptual del enfoque propuesto.

En el capítulo anterior se describió cómo obtener información relevante a partir de la ejecución de un programa mediante instrumentación de bytecode. El resultado de ese proceso es un *trace* de ejecución que registra eventos como accesos a memoria, entradas y salidas de métodos, y comportamiento de loops.

El objetivo de este capítulo es presentar el modelo utilizado para analizar dichos *traces* y detectar dependencias dinámicas entre accesos a memoria. A partir de este análisis, se busca determinar si ciertas estructuras, en particular loops, presentan oportunidades de paralelización.

Para ello, se introduce una noción formal de orden en la ejecución mediante los conceptos de execution points y execution tree. Luego, se describe cómo detectar dependencias dinámicas y cómo clasificarlas en RAW, WAR, WAW y RAR. Finalmente, se explica cómo esta información puede ser utilizada para asistir al programador en la identificación de regiones potencialmente paralelizables.

Este enfoque se enmarca dentro de la línea de trabajo de análisis dinámico de dependencias, como el propuesto por Ketterlin y Clauss en el contexto de profiling de dependencias a nivel de código binario [8]. A diferencia de dicho enfoque, en esta tesis el análisis se realiza sobre programas en lenguaje Java, operando directamente sobre el bytecode de la JVM. Esta decisión permite trabajar a un nivel de abstracción intermedio, conservando información estructural del programa que resulta útil para asociar dependencias dinámicas con construcciones de alto nivel como loops.

4.1. Descripción general del pipeline de análisis

En esta sección se describe el proceso específico de análisis aplicado sobre los *traces* de ejecución, en el marco del pipeline general presentado en el Capítulo 2.

El proceso comienza con la ejecución del programa previamente instrumentado, el cual genera un *trace* de ejecución que contiene eventos relevantes, tales como accesos a memoria, entradas y salidas de métodos, y eventos asociados a la ejecución de loops.

A partir de este *trace*, se reconstruye un árbol de ejecución que representa la estructura dinámica del programa durante su ejecución. En paralelo, se mantiene una tabla de objetos que permite registrar los accesos más recientes a cada posición de memoria, lo cual resulta fundamental para la detección de dependencias.

Finalmente, el analizador de dependencias procesa esta información con el objetivo de identificar dependencias dinámicas entre distintas partes del programa, particularmente entre iteraciones de loops, permitiendo así clasificar su potencial de paralelización.

En las siguientes secciones se describe en detalle cada uno de los componentes que conforman este proceso.

Para ilustrar el proceso general del enfoque, se presenta el pipeline de la herramienta.

Como se describió en el Capítulo 2 (Figura 2.1), el proceso completo se organiza en distintas etapas, desde la instrumentación del programa hasta el análisis de dependencias.

El pipeline resume las etapas principales del enfoque propuesto. En primer lugar, el programa es instrumentado y ejecutado para generar un *trace* de eventos. A partir de este *trace*, se reconstruye una representación estructurada de la ejecución, que permite analizar las relaciones entre accesos a memoria. Finalmente, el analizador de dependencias procesa esta información para identificar dependencias dinámicas y evaluar el potencial de paralelización de los loops.

4.2. Un marco formal para el análisis de dependencias dinámicas

El análisis de dependencias dinámicas requiere observar los accesos a memoria realizados durante la ejecución del programa. Se dice que existe una dependencia cuando dos accesos refieren a la misma ubicación de memoria y al menos uno de ellos es una escritura.

Para detectar estas dependencias es necesario poder responder a la siguiente pregunta:

> *¿Cuándo ocurrió el último acceso a una determinada ubicación de memoria y qué tipo de acceso fue (lectura o escritura)?*

Para formalizar la noción de *cuándo*, introducimos el concepto de **execution point**, que permite establecer un orden entre los eventos observados durante la ejecución.

4.3. Execution Points

Un execution point representa un instante en la ejecución del programa. Cada acceso a memoria está asociado a un execution point único, lo que permite comparar dos accesos y determinar cuál ocurrió antes.

El análisis de dependencias dinámicas requiere poder establecer un orden entre accesos a memoria. Una primera aproximación consiste en utilizar una numeración global de instrucciones, asignando a cada instrucción un identificador incremental que refleje el orden de ejecución. Este enfoque ha sido utilizado en trabajos previos para analizar paralelismo potencial en programas secuenciales [6].

Sin embargo, este modelo resulta insuficiente en presencia de estructuras como llamadas a métodos y loops. En particular, una numeración lineal no logra capturar correctamente el contexto de ejecución cuando existen múltiples invocaciones de un mismo método o múltiples iteraciones de un loop.

Una alternativa consiste en modelar el tiempo de ejecución utilizando estructuras más ricas, que reflejen el contexto dinámico de ejecución, como la pila de llamadas y la posición relativa dentro de cada método [20]. En este enfoque, cada execution point puede representarse mediante una estructura jerárquica en lugar de un valor escalar.

En este trabajo adoptamos un enfoque basado en la estructura del programa, donde la noción de tiempo se define en función de llamadas a métodos, loops y sus iteraciones. Esta representación permite distinguir correctamente accesos a memoria que ocurren en diferentes contextos de ejecución, incluso cuando pertenecen al mismo código fuente.

En particular, la presencia de loops introduce un desafío adicional: accesos a memoria en distintas iteraciones pueden referirse a la misma ubicación, y una representación lineal no permite distinguir correctamente su orden relativo.

Para resolver este problema, se adopta una representación jerárquica de la ejecución que permite capturar el contexto dinámico en el cual ocurren los accesos a memoria. Esta representación, denominada execution tree, será introducida en la siguiente sección y constituye la base para el análisis de dependencias dinámicas.

4.4. Árbol de ejecución (Execution Tree)

Para poder modelar correctamente el orden de ejecución en presencia de llamadas a métodos y loops, utilizamos una representación jerárquica denominada *execution tree*.

Las estructuras que nos interesan analizar son: llamadas a métodos, loops y sus iteraciones, y accesos a memoria. Estas estructuras, que en el código fuente están organizadas de manera estática, se transforman en tiempo de ejecución en una estructura dinámica que refleja el contexto real en el cual ocurren los eventos.

El execution tree es un árbol que representa esta ejecución dinámica. En él, cada nodo corresponde a un evento relevante durante la ejecución del programa, y la estructura del árbol captura el contexto en el cual dicho evento ocurre.

En particular, el execution tree contiene los siguientes tipos de nodos:

- **MethodCallNode**: representa la invocación de un método.
- **LoopNode**: representa la existencia de un loop dentro de un método.
- **LoopIterationNode**: representa una iteración particular de un loop. Cada iteración se modela como un nodo independiente.
- **ObjectAccessNode**: representa un acceso a memoria (lectura o escritura). Estos nodos son hojas del árbol.

Durante la ejecución del programa, el *trace* generado por la etapa de instrumentación permite reconstruir incrementalmente este árbol. A medida que se procesan los eventos del trace:

- Un evento de entrada a método genera un nuevo *MethodCallNode*.
- Un evento de entrada a loop genera un *LoopNode*.
- Cada iteración del loop genera un *LoopIterationNode*.
- Cada acceso a memoria genera un *ObjectAccessNode*, que se ubica como hoja dentro del contexto correspondiente.

De este modo, cada acceso a memoria queda ubicado en una posición única dentro del árbol, determinada por el camino desde la raíz hasta el nodo hoja correspondiente. Este camino define lo que denominamos *execution path*.

El execution path permite identificar unívocamente cada acceso a memoria, incluyendo el contexto completo en el cual ocurre: método, loops anidados e iteraciones específicas.

Esta representación es fundamental para el análisis de dependencias dinámicas. Dado que múltiples accesos pueden referirse a la misma ubicación de memoria, es necesario poder determinar no solo su identidad, sino también su orden relativo dentro de la ejecución.

Para ello, al comparar dos accesos a memoria sobre la misma ubicación, analizamos sus respectivos execution paths y determinamos su ancestro común más cercano en el execution tree. Este nodo ancestro es el contexto en el cual se manifiesta la dependencia.

El tipo de dependencia detectada (por ejemplo, RAW, WAR o WAW) y el contexto en el cual ocurre son claves para determinar si un loop es potencialmente paralelizable.

Para poder realizar este análisis de manera eficiente durante el procesamiento del trace, se introduce una estructura adicional denominada *Object Table*, que será descripta en la siguiente sección.

En resumen, el execution tree permite representar de manera precisa el contexto dinámico de ejecución, y constituye una base sólida para comparar accesos a memoria y ubicar correctamente las dependencias dentro de la estructura del programa.

4.4.1. Construcción del árbol de ejecución

El árbol de ejecución se construye dinámicamente a partir de la traza generada durante la ejecución del programa. Esta estructura permite representar el contexto jerárquico de ejecución, incluyendo la anidación de llamadas a métodos y la estructura de loops e iteraciones.

A continuación se presenta el algoritmo utilizado para construir esta representación a partir de los eventos registrados en la traza.

Algorithm 1 Construcción del árbol de ejecución

```
1: current ← root
2: for cada evento e en la traza do
3:     if e es METHOD_ENTRY then
4:         crear nodo método n
5:         agregar n como hijo de current
6:         current ← n
7:     end if
8:     if e es METHOD_EXIT then
9:         current ← parent(current)
10:    end if
11:    if e es LOOP_ENTRY then
12:        crear nodo loop l
13:        agregar l como hijo de current
14:        crear nodo iteración i
15:        agregar i como hijo de l
16:        current ← i
17:    end if
18:    if e es LOOP_ITER then
19:        crear nodo iteración i
20:        agregar i como hijo del loop actual
21:        current ← i
22:    end if
23:    if e es LOOP_EXIT then
24:        current ← parent(parent(current))
25:    end if
26:    if e es acceso a memoria then
27:        registrar acceso en el nodo current
28:    end if
29: end for
```

Este procedimiento reconstruye el contexto dinámico de ejecución del programa, representando explícitamente la anidación de llamadas a métodos y la estructura de loops e iteraciones.

Cada nodo del árbol corresponde a un punto de ejecución, permitiendo ubicar los accesos a memoria dentro de una jerarquía que refleja el flujo real del programa. En particular, esta representación permite distinguir entre accesos que ocurren en distintas iteraciones de un loop, lo cual es fundamental para el análisis de dependencias dinámicas.

La construcción del árbol se realiza en una única pasada sobre la traza, con costo lineal respecto al número de eventos registrados.

4.5. Object Table

Todo acceso a memoria está identificado unívocamente por su *execution path*, definido como el camino desde la raíz del execution tree hasta el nodo hoja correspondiente (*ObjectAccessNode*). Este camino refleja el contexto completo de ejecución en el cual ocurre el acceso.

Dados dos accesos a memoria sobre la misma ubicación, podemos comparar sus respectivos *execution paths* para determinar su relación en la ejecución.

En particular, es posible calcular su ancestro común más cercano dentro del execution tree. Sea este nodo J. Este nodo representa el contexto de ejecución más profundo compartido por ambos accesos.

A partir de J, los execution paths divergen hacia dos subramas distintas. Llamamos C1 y C2 a los hijos inmediatos de J que conducen a cada uno de los accesos.

Decimos que el nodo J **lleva** la dependencia, ya que es el punto del programa donde se manifiesta la relación entre ambos accesos. En este contexto, C1 representa la fuente (source) de la dependencia y C2 el destino (sink).

Por ejemplo, si el nodo J corresponde a un loop, entonces la dependencia ocurre entre distintas iteraciones de ese loop. Esta información es fundamental para determinar si dicho loop es paralelizable.

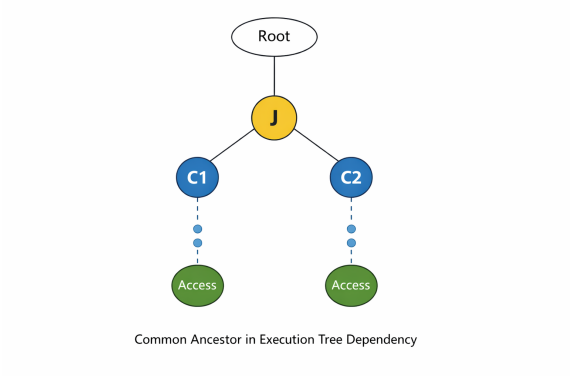

Fig. 4.1: Dependencia entre accesos a memoria en el execution tree. El nodo J representa el ancestro común, mientras que C1 y C2 corresponden a los caminos divergentes hacia cada acceso.

4.5.1. Estructura de la Object Table

Para cada ubicación de memoria, se mantiene información sobre los accesos más recientes realizados durante la ejecución.

En particular, para cada ubicación se almacena:

- El último acceso de lectura
- El último acceso de escritura
- El execution path asociado a dichos accesos

Este enfoque permite detectar dependencias en tiempo constante por acceso, evitando la necesidad de recorrer estructuras completas de ejecución.

Esta información es suficiente para detectar dependencias dinámicas considerando únicamente los accesos más recientes, lo que permite mantener el análisis eficiente.

4.5.2. Algoritmo de detección de dependencias

El proceso de detección de dependencias dinámicas se basa en el análisis incremental de los accesos a memoria registrados durante la ejecución. Para cada acceso, se consulta

la tabla de accesos a objetos y se compara con el último acceso previo sobre la misma ubicación de memoria.

Este procedimiento se apoya en la estructura del árbol de ejecución para determinar el contexto en el cual ocurre la dependencia, identificando el ancestro común más cercano entre los accesos involucrados.

Algorithm 2 Detección de dependencias dinámicas basada en execution tree

```
1: for cada acceso a memoria a do
2:     entry ← ObjectAccessTable.lookup(a.objectId, a.fieldId)
3:     if entry ≠ null then
4:         last ← último acceso registrado
5:         p1 ← last.node
6:         p2 ← a.node
7:         while depth(p1) > depth(p2) do
8:             p1 ← parent(p1)
9:         end while
10:        while depth(p2) > depth(p1) do
11:            p2 ← parent(p2)
12:        end while
13:        while p1 ≠ p2 do
14:            p1 ← parent(p1)
15:            p2 ← parent(p2)
16:        end while
17:        if p1 corresponde a un loop then then
18:            if last es write y a es read then
19:                dependencia ← RAW
20:            end if
21:            if last es read y a es write then
22:                dependencia ← WAR
23:            end if
24:            if last es write y a es write then
25:                dependencia ← WAW
26:            end if
27:            if last es read y a es read then
28:                dependencia ← RAR
29:            end if
30:        end if
31:    end if
32:    actualizar ObjectAccessTable con el acceso a
33: end for
```

Este procedimiento corresponde a la búsqueda del ancestro común más cercano (Lowest Common Ancestor, LCA) en el árbol de ejecución, lo que permite determinar el contexto compartido entre ambos accesos.

Para determinar si una dependencia ocurre entre distintas iteraciones de un mismo loop, el analizador inspecciona los nodos hijos inmediatos del ancestro común más cercano en los caminos correspondientes a ambos accesos, verificando si pertenecen a diferentes

instancias de `LoopIterationNode`.

El algoritmo opera de manera incremental, procesando cada acceso a memoria en el orden en que ocurre durante la ejecución. Para cada acceso, se consulta la tabla de objetos para recuperar el último acceso registrado sobre la misma ubicación de memoria, evitando recorridos completos sobre la traza de ejecución.

La determinación del contexto de la dependencia se realiza mediante la búsqueda del ancestro común más cercano (LCA) en el árbol de ejecución, lo que permite asociar la dependencia a una estructura de control específica, como un loop.

El acceso a la tabla de objetos se realiza en tiempo constante, mientras que la determinación del contexto tiene un costo proporcional a la profundidad del árbol de ejecución. En la práctica, esta profundidad suele ser acotada, permitiendo mantener un costo reducido por acceso incluso en ejecuciones de gran tamaño.

Overview del algoritmo de detección de dependencias

El análisis de dependencias dinámicas se realiza procesando secuencialmente el *trace* de ejecución. En resumen, para cada acceso a memoria:

1. Se identifica la ubicación de memoria accedida.
2. Se consulta la Object Table para obtener los últimos accesos registrados.
3. Se determina el tipo de dependencia dinámica detectada (RAW, WAR, WAW o RAR).
4. En caso afirmativo, se calcula el ancestro común en el execution tree.
5. Se registra la dependencia asociada al contexto correspondiente.
6. Se actualiza la Object Table con el acceso actual.

4.5.3. Detección de dependencias

Al procesar un nuevo acceso a memoria, se consulta la Object Table para recuperar los accesos previos sobre la misma ubicación.

A partir de esta información, se detectan los siguientes tipos de dependencias:

- RAW (Read After Write)
- WAR (Write After Read)
- WAW (Write After Write), algunas de las cuales pueden ser consideradas dependencias falsas (ver Sección 4.6)

Una vez detectada la dependencia, se determina el ancestro común en el execution tree, que define el contexto en el cual la dependencia ocurre.

4.5.4. Representación interna

En la implementación, las dependencias detectadas se almacenan asociadas a cada loop.

Se utiliza una estructura:

```
Map<Integer, LoopDependenceInfo> falseDepTables
```

donde cada loop tiene asociadas las dependencias detectadas durante la ejecución.

Cada instancia de `LoopDependenceInfo` contiene conjuntos de dependencias clasificadas por tipo:

```
Set<Dependence> raw;
Set<Dependence> war;
Set<Dependence> waw;
```

Cada dependencia almacena información sobre los accesos involucrados, incluyendo los basic blocks y la posición relativa dentro de ellos.

De esta manera, la Object Table actúa como una estructura central en el análisis, permitiendo detectar dependencias de manera eficiente durante el procesamiento del trace, y vinculándolas con su contexto de ejecución en el execution tree.

4.6. Dependencias falsas

Una dependencia es considerada **falsa** cuando no refleja una verdadera dependencia de datos, sino una reutilización de memoria.

4.6.1. Ejemplo

```
Color temp = ...;
for (int i= ...)
    for (int j=...){
        temp.set(a[i][j]);      // WRITE
        ...
        b[i][j].set(temp);      // READ
    }
```

En este caso se detecta una dependencia WAR llevada por el loop interno.

Sin embargo, esta dependencia se debe únicamente a la reutilización del objeto `temp`, no a su valor. Por lo tanto, se trata de una *false dependence*.

Desde la perspectiva del análisis dinámico, la dependencia aparece porque todas las iteraciones acceden a la misma ubicación de memoria asociada al objeto `temp`. La herramienta observa una secuencia repetida de escrituras y lecturas sobre esa misma ubicación de memoria y, en consecuencia, registra una relación de dependencia entre accesos pertenecientes a distintas iteraciones.

No obstante, esta dependencia no refleja una transferencia real de información entre iteraciones del algoritmo, sino únicamente la reutilización de una estructura de almacenamiento temporal. En otras palabras, las iteraciones comparten la misma ubicación de memoria por una decisión de implementación, aun cuando conceptualmente podrían ejecutarse utilizando copias independientes del objeto temporal.

4.6.2. Privatization

Esta situación puede resolverse mediante una transformación conocida como ***privatization***, donde cada iteración utiliza su propia instancia:

```
for (int i= ...)
    for (int j=...){
        temp[i][j].set(a[i][j]);
        ...
        b[i][j].set(temp[i][j]);
    }
```

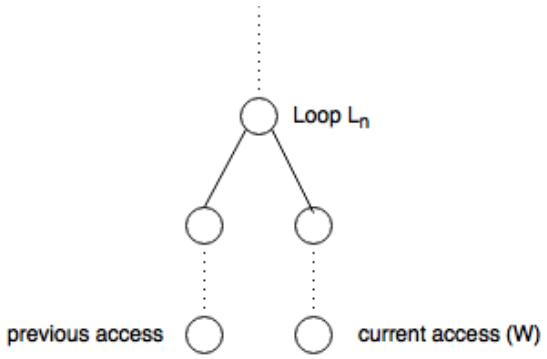

Fig. 4.2: Ejemplo de dependencia detectada durante la ejecución.

Cabe destacar que la herramienta detecta la dependencia observada durante la ejecución, pero actualmente no distingue automáticamente si ésta corresponde a una dependencia verdadera de datos o a un patrón de reutilización de almacenamiento susceptible de ser eliminado mediante privatización. No obstante, la presencia sistemática de accesos sobre una misma ubicación de memoria podría utilizarse como indicio para sugerir este tipo de optimizaciones al programador.

En consecuencia, la identificación de false dependencies resulta clave para no descartar oportunidades de paralelización, permitiendo distinguir entre restricciones reales del programa y limitaciones introducidas por decisiones de implementación.

4.7. Ejemplos de dependencias

4.7.1. Dependencia RAW

```
if(i==1)
    t[i].a = t[i-1].a;
```

4.7.2. Dependencia WAR

```
t[i].a = t[i+1].a;
```

4.7.3. Dependencia WAW

```
t[i].a = i;
t[i+1].a = i;
```

4.7.4. Dependencia RAR

```
temp = t[i].a;
temp = t[i+1].a;
```

4.7.5. Loop con iteraciones conflictivas

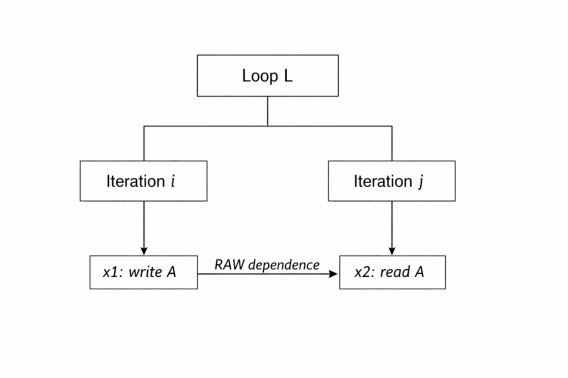

Fig. 4.3: Dependencia entre iteraciones de un loop.

Los ejemplos presentados fueron utilizados para validar el comportamiento de la herramienta, verificando que las dependencias detectadas coincidan con el comportamiento esperado.

Estos patrones básicos permiten ilustrar los distintos tipos de dependencias que pueden surgir durante la ejecución. En las siguientes secciones, se analizará cómo estas dependencias son detectadas automáticamente a partir del *trace* de ejecución y cómo impactan en la posibilidad de paralelizar loops.

4.8. Resultados del análisis y asistencia al programador

El objetivo final de este análisis es asistir en la toma de decisiones sobre paralelización, proporcionando información concreta sobre las restricciones impuestas por dependencias dinámicas.

Esta sección integra los distintos componentes presentados previamente, mostrando cómo la información obtenida a partir del *trace* de ejecución es procesada para detectar dependencias dinámicas y asistir al programador en la identificación de oportunidades de paralelización.

A partir del análisis dinámico de dependencias, es posible clasificar los loops de acuerdo al tipo de dependencias detectadas durante la ejecución:

- **Dependencias verdaderas (RAW):** indican que existe una dependencia real entre iteraciones del loop. En estos casos, el loop no es paralelizable sin modificar la lógica del programa.
- **Dependencias falsas (WAR/WAW):** corresponden a conflictos de escritura o lectura-escritura que no implican una dependencia real de datos. Estos casos pueden resolverse mediante técnicas como privatización de variables o renombrado, permitiendo potencialmente la paralelización.

- **Dependencias no conflictivas (RAR):** cuando se detectan únicamente lecturas sobre los mismos datos compartidos, se registra una dependencia de tipo RAR. Al no existir escrituras asociadas, este tipo de dependencia no impone restricciones estrictas de orden y el loop permanece potencialmente paralelizable.

De esta manera, la herramienta permite asociar a cada loop una clasificación basada en su comportamiento dinámico, brindando al programador una guía concreta sobre qué partes del código pueden ser paralelizadas y cuáles requieren transformaciones adicionales.

4.8.1. Presentación del feedback

El análisis realizado por la herramienta genera un conjunto de archivos de salida que contienen información detallada sobre la ejecución del programa y las dependencias detectadas.

Entre los principales artefactos generados se encuentran:

- Información sobre la estructura de loops (Loop Nesting Tree)
- *Traces* de ejecución con accesos a memoria y eventos relevantes
- Mapeos entre identificadores internos (clases, métodos, loops, basic blocks) y el código fuente original

Estos datos permiten reconstruir el comportamiento dinámico del programa y analizar, para cada loop, la presencia y tipo de dependencias entre iteraciones.

4.8.2. Clasificación de loops

A partir de la información recolectada, cada loop puede ser interpretado de la siguiente manera:

- **Loop no paralelizable:** se detectan dependencias verdaderas (RAW) entre iteraciones, lo que impone un orden de ejecución.
- **Loop potencialmente paralelizable:** se detectan únicamente dependencias WAR o WAW, las cuales pueden eliminarse mediante transformaciones como privatización de variables.
- **Loop paralelizable:** no se detectan dependencias entre iteraciones (o únicamente RAR), lo que indica independencia entre iteraciones.

Sin embargo, en la implementación actual, la herramienta clasifica los loops según las dependencias dinámicas detectadas durante la ejecución analizada.

En particular, los loops que presentan únicamente dependencias de tipo RAR son reportados explícitamente como potencialmente paralelizables, mientras que aquellos con dependencias RAW o WAR/WAW son identificados como loops que requieren preservación del orden de ejecución o transformaciones adicionales.En la configuración analizada, los loops que presentan únicamente dependencias de tipo RAR son reportados preliminarmente por la herramienta como potencialmente paralelizables. No obstante, esta clasificación está sujeta a las limitaciones de cobertura del escenario dinámico observado; la ausencia de escrituras registradas en la traza para una ubicación de memoria específica puede provocar que dependencias de reutilización subyacentes se reporten erróneamente como RAR, requiriendo validación complementaria por parte del programador.

4.8.3. Localización en el código fuente

Para permitir al programador ubicar los loops en el código fuente, la herramienta provee información que vincula los elementos del análisis dinámico con estructuras del programa original.

En particular, se dispone de:

- Identificadores de métodos y loops
- Conjuntos de basic blocks asociados a cada loop
- Mapeo de basic blocks a líneas de código fuente

Esta información permite reconstruir la ubicación aproximada de cada loop en el código fuente, facilitando la inspección manual por parte del programador.

4.8.4. Ejemplo ilustrativo

Considérese el siguiente loop:

```
for (int i = 0; i < N; i++) {
    A[i] = B[i] + 1;
}
```

En este caso, cada iteración accede a posiciones independientes del arreglo, por lo que no existen dependencias entre iteraciones. Si este comportamiento se mantiene durante la ejecución analizada, la herramienta no registrará conflictos asociados a este loop, lo que permite inferir que es paralelizable.

Por otro lado, considérese el siguiente caso:

```
for (int i = 1; i < N; i++) {
    A[i] = A[i-1] + 1;
}
```

Aquí existe una dependencia verdadera (RAW) entre iteraciones consecutivas, ya que cada iteración depende del resultado de la anterior. Este tipo de dependencia será detectado por la herramienta, indicando que el loop no es paralelizable sin modificaciones adicionales.

4.8.5. Limitaciones del enfoque

Es importante destacar que, al tratarse de un análisis dinámico, los resultados dependen del input utilizado durante la ejecución. Por lo tanto, la ausencia de dependencias en un caso particular no garantiza que el loop sea paralelizable para todas las posibles ejecuciones.

Asimismo, la identificación de loops potencialmente paralelizables depende estrictamente de las dependencias observadas durante la ejecución analizada. En particular, la presencia exclusiva de dependencias de tipo RAR puede sugerir oportunidades de paralelización, aunque esto no constituye una garantía general sobre todos los posibles escenarios de ejecución.

El enfoque propuesto permite analizar el comportamiento concreto observado durante la ejecución del programa, proporcionando información útil como soporte para la toma de decisiones del programador y complementando enfoques estáticos más conservadores.

4.8.6. Ejemplo de salida real de la herramienta

En esta sección se presenta un ejemplo real del output generado por la herramienta durante la ejecución de un programa con loops.

Fragmento de trace de ejecución
A continuación se muestra un fragmento del archivo `tracer.out`:

```
LEN L6
...
AAS 1163904012-1
AAL 1163904012-1
AAL 1163904012-0
...
LIT L6
...
AAS 1163904012-2
...
LEX L6
```

En este fragmento:

- `LEN L6` indica la entrada al loop L6
- `LIT L6` indica una iteración del loop
- `LEX L6` indica la salida del loop
- `AAS` representa una escritura en memoria (store)
- `AAL` representa una lectura en memoria (load)

Se observa que durante distintas iteraciones del loop se accede a posiciones de memoria relacionadas (por ejemplo, índices consecutivos del mismo arreglo), lo cual puede dar lugar a dependencias entre iteraciones.

Estructura de loops
El archivo `loop_nesting_tree.out` permite identificar la estructura de loops:

```
METHOD 0-2
    LOOP L0
        LOOP L1
        LOOP L2
```

Esto permite ubicar los loops analizados dentro de cada método y entender su anidamiento.

Análisis de dependencias
El resultado del analyzer se encuentra en el archivo `outDepAnalyzerNew.txt`. A continuación se muestra un resumen de los resultados obtenidos:

```
*** loopsWithRAWDependences - RAW - (NON PARALLEL): [25]

*** loopsWithFalseDependences - WAR & WAW - (MAYBE PARALLEL): [5, 27]

*** loopsWithRarDependencesDetectedList - RAR - (PARALLEL): [2, 3, 4, 29, 30, 31]
```

Este resultado clasifica los loops detectados en tres categorías:

- Loops con dependencias RAW: no paralelizables
- Loops con dependencias WAR/WAW: potencialmente paralelizables mediante transformaciones
- Loops con dependencias RAR: paralelizables

Además, el analyzer reporta dependencias específicas, por ejemplo:

```
New RAW Dependence found!!
originBasicBlockId=400 -> destBasicBlockId=398
```

Esto indica una dependencia verdadera entre accesos a memoria en distintas iteraciones del loop.

Interpretación
A partir de estos resultados, es posible concluir que:

- Los loops listados en `loopsWithRAWDependences` no pueden paralelizarse sin modificar la lógica del programa.
- Los loops en `loopsWithFalseDependences` pueden ser paralelizados aplicando técnicas como privatización de variables.
- Los loops en `loopsWithRarDependencesDetectedList` son paralelizables, ya que no presentan dependencias que impongan orden entre iteraciones.

Este ejemplo muestra el flujo completo de la herramienta, desde la captura de eventos en tiempo de ejecución hasta la clasificación final de loops según su potencial de paralelización.

En conjunto, la herramienta no solo permite detectar dependencias dinámicas, sino también orientar al programador en la comprensión de su impacto, facilitando la identificación de oportunidades concretas de paralelización.

4.8.7. Archivos de salida

Para poder interpretar los resultados experimentales, es necesario comprender los archivos de salida generados por la herramienta.

Durante la ejecución, la herramienta genera distintos archivos que permiten analizar el comportamiento dinámico del programa.

En los ejemplos presentados se utilizan principalmente:

- `tracer.out`: contiene el *trace* de ejecución, incluyendo accesos a memoria y eventos de loops.
- `loop nesting tree.out`: describe la estructura jerárquica de loops detectados.
- `basic blocks line numbers.out`: permite mapear bloques básicos a líneas de código fuente.

En las siguientes secciones se muestran fragmentos relevantes de estos archivos para ilustrar el proceso de detección de dependencias.

Mapeo a código fuente

Para poder interpretar correctamente los resultados del análisis, la herramienta genera archivos auxiliares que permiten mapear la información dinámica al código fuente original.

Archivo `basic_blocks_line_numbers.out` Este archivo permite relacionar cada basic block con las líneas de código fuente correspondientes.

Ejemplo:

```
0-2    B4    6-9
0-2    B7    9-10
0-2    B10   15-16
```

En este formato:

- `0-2` identifica el método (classId-methodId)
- `B4`, `B7`, etc., identifican basic blocks
- El último valor indica el rango de líneas de código fuente asociadas

Esto permite ubicar en el código fuente los basic blocks involucrados en una dependencia reportada por el analyzer.

Archivo `class_method_field_names.out` Este archivo permite mapear los identificadores numéricos utilizados internamente a nombres reales de clases, métodos y campos.

Ejemplo:

```
C0    inria/camus/tracer/test/DemoSimpleLoop
C0-M2 main.([Ljava/lang/String;)V
C1    java/lang/System
C1-F1 out
```

En este formato:

- `C0` representa una clase
- `C0-M2` representa un método de esa clase
- `C1-F1` representa un campo

Este archivo es fundamental para interpretar los resultados del analyzer en términos del código original.

Uso conjunto Combinando estos archivos con la salida del analyzer, es posible:

- Identificar qué loop presenta una dependencia
- Determinar en qué método ocurre
- Ubicar las líneas exactas del código fuente involucradas

De esta manera, la herramienta no solo detecta dependencias dinámicas, sino que también permite rastrear su origen en el código, facilitando el análisis y la posible optimización del programa.

En conjunto, las estructuras y mecanismos presentados —execution tree, object table y detección de dependencias— conforman un modelo coherente que permite identificar dependencias dinámicas y asociarlas a estructuras del programa. Este modelo constituye la base del análisis de paralelización utilizado en el capítulo siguiente para la validación experimental del enfoque.

5. VALIDACIÓN DEL ENFOQUE MEDIANTE EJEMPLOS Y BENCHMARKS

In God we trust. All others must bring data.
— W. Edwards Deming

Este capítulo presenta la evaluación del enfoque propuesto desde dos perspectivas complementarias: una validación cualitativa basada en ejemplos representativos y una evaluación cuantitativa utilizando benchmarks estándar.

Por un lado, se realiza una validación cualitativa orientada a verificar la correctitud del análisis y su interpretabilidad. Por otro lado, se presenta una evaluación cuantitativa utilizando benchmarks de la suite JavaGrande, con el objetivo de analizar el costo de ejecución y la escalabilidad del enfoque.

5.1. Metodología de evaluación

La evaluación se divide en dos partes:

- **Evaluación cualitativa:** basada en programas de prueba diseñados para cubrir distintos patrones de dependencias.
- **Evaluación cuantitativa:** basada en benchmarks estándar para medir overhead y escalabilidad.

El objetivo es evaluar tanto la correctitud del enfoque como su costo de ejecución en escenarios controlados y en programas reales.

5.2. Evaluación cualitativa mediante ejemplos

En esta sección se presentan ejemplos ilustrativos que permiten comprender el funcionamiento de la herramienta y validar cualitativamente su comportamiento. En las secciones siguientes se complementa esta evaluación con benchmarks y mediciones cuantitativas de overhead.

Se utilizaron programas diseñados para representar distintos patrones de acceso a memoria:

- Loops sin dependencias entre iteraciones
- Loops con dependencias verdaderas (RAW)
- Loops con dependencias falsas (WAR/WAW)
- Uso de variables temporales compartidas

Cada programa fue instrumentado y ejecutado, generando trazas que fueron posteriormente analizadas.

5.2.1. Criterios de evaluación

La evaluación cualitativa se centró en:

- **Correctitud:** correspondencia entre dependencias detectadas y comportamiento esperado
- **Clasificación:** correcta identificación de RAW, WAR, WAW y RAR
- **Asociación a loops:** vinculación correcta con estructuras del programa
- **Interpretabilidad:** utilidad de la información para el programador

5.2.2. Enfoque experimental

En esta sección se presentan distintos ejemplos con el objetivo de ilustrar el funcionamiento de la herramienta y su utilidad en la detección de dependencias dinámicas.

Para cada caso se sigue el siguiente esquema:

- Se presenta un fragmento de código representativo
- Se describe el comportamiento esperado en términos de dependencias
- Se analiza el resultado producido por la herramienta
- Se interpreta el resultado desde el punto de vista de la paralelización

5.2.3. Problemas en la detección de dependencias WAR vs RAR

En esta sección se presentan una serie de experimentos diseñados para analizar el comportamiento de la herramienta en la detección de distintos tipos de dependencias de datos, con foco particular en los casos WAR (Write-After-Read) y RAR (Read-After-Read).

Descripción del caso de prueba

Se implementaron pequeños programas en Java con bucles anidados y variables temporales, en los cuales se esperaban dependencias de tipo WAR debido a la reutilización de almacenamiento intermedio.

En particular, se analizaron casos donde una variable temporal o una ubicación de memoria es escrita luego de haber sido leída en una iteración previa.

Comportamiento observado

La herramienta detectó dependencias de tipo RAR en algunos casos donde, desde el punto de vista semántico del programa, podrían esperarse dependencias WAR asociadas a la reutilización de almacenamiento temporal.

La situación anterior apareció en escenarios donde los primeros accesos registrados sobre una determinada ubicación de memoria correspondían a operaciones de lectura. En estos casos, la clasificación obtenida depende del orden concreto de eventos presentes en el trace y de la información disponible en la estructura incremental de seguimiento de accesos utilizada por el prototipo.

Análisis

Este resultado pone en evidencia una limitación importante del análisis dinámico de dependencias: la clasificación depende estrictamente del *trace* observado durante la ejecución.

En consecuencia, la clasificación dinámica de dependencias puede verse influenciada por el orden concreto de accesos observado durante la ejecución analizada, así como por decisiones de implementación relacionadas con el seguimiento incremental de estados en memoria.

Este efecto es especialmente relevante en presencia de estructuras de memoria no inicializadas explícitamente o inicializadas por defecto, como los arreglos en Java.

Implicancias

Estos resultados sugieren que el análisis dinámico puede subestimar ciertos tipos de dependencias dependiendo del escenario de ejecución.

Por lo tanto, la calidad del análisis está fuertemente condicionada por los datos de entrada y los caminos de ejecución explorados.

Esto refuerza la necesidad de interpretar los resultados del análisis dinámico en conjunto con el conocimiento del programa y del contexto de ejecución observado.

Ejemplo de código

A continuación se muestra un ejemplo representativo utilizado en los experimentos:

```
public class ExampleWAR2 {
    public static void main(String[] args) {
        int[][] array_a = new int[10][10];
        int[][] array_b = new int[10][10];
        int[][] array_c_temp = new int[10][10];

        for (int j = 0; j < array_a.length; j++) {
            for (int h = 0; h < array_a.length; h++) {
                array_c_temp[0][0] = array_a[j][h];
                array_b[j][h] = array_c_temp[0][0] + 1;
            }
        }
    }
}
```

En este ejemplo, la utilización de una ubicación de memoria compartida (`array_c_temp[0][0]`) permite observar una dependencia de tipo WAR entre iteraciones, ya que el valor es leído y posteriormente sobrescrito.

Este tipo de patrón resulta más claramente detectable por la herramienta en comparación con el uso de variables temporales locales.

Desde el punto de vista de la paralelización, este ejemplo ilustra cómo la presencia de una ubicación de memoria compartida entre iteraciones puede introducir dependencias que impiden la ejecución en paralelo.

Asimismo, pone en evidencia una limitación del análisis dinámico: la clasificación de dependencias depende del orden observado en la ejecución. En particular, si no se observa

una escritura previa, ciertas dependencias pueden ser clasificadas incorrectamente como RAR.

Este tipo de situaciones refuerza la necesidad de interpretar los resultados del análisis en conjunto con el conocimiento del programa, utilizando la herramienta como apoyo para la toma de decisiones.

5.2.4. Loop sin dependencias

Se considera el siguiente ejemplo:

```
for (int i = 0; i < N; i++) {
    A[i] = B[i] + 1;
}
```

En este caso, cada iteración accede a posiciones independientes de memoria. No existe solapamiento entre accesos, por lo que no se generan dependencias entre iteraciones.

Durante la ejecución, la herramienta no detecta conflictos asociados a este loop.

Esto permite inferir que el loop es paralelizable, ya que sus iteraciones son independientes y pueden ejecutarse en paralelo sin necesidad de transformaciones adicionales.

5.2.5. Loop con dependencia verdadera (RAW)

Considérese el siguiente loop:

```
for (int i = 1; i < N; i++) {
    A[i] = A[i-1] + 1;
}
```

En este caso, cada iteración depende del resultado de la iteración anterior, generando una dependencia verdadera (RAW).

La herramienta detecta accesos conflictivos sobre la misma ubicación de memoria entre iteraciones consecutivas.

Como resultado, este loop no es paralelizable sin modificar su lógica, ya que existe una dependencia que impone un orden de ejecución.

5.2.6. Ejemplos de algoritmos

Estos casos permiten validar el comportamiento de la herramienta en patrones típicos de programación.

La herramienta fue evaluada sobre distintos patrones comunes, tales como recorridos de arreglos y loops con acumulación.

En el caso de recorridos simples de arreglos, donde cada iteración accede a posiciones independientes, no se detectaron dependencias entre iteraciones, lo que indica paralelismo potencial.

En contraste, en algoritmos con acumulación (por ejemplo, sumatorias), se detectaron dependencias verdaderas (RAW), evidenciando la necesidad de transformaciones adicionales como reducción para su paralelización.

5.2.7. Aspectos de implementación y limitaciones

Nivel de bytecode.

El análisis se realiza a nivel de bytecode mediante instrumentación dinámica, lo que permite capturar eventos de acceso a memoria, entradas y salidas de métodos, y ejecución de loops.

Esto permite detectar dependencias independientemente de las construcciones del lenguaje fuente, incluyendo llamadas a métodos y estructuras complejas.

Sin embargo, trabajar a este nivel introduce desafíos adicionales, como la identificación precisa de estructuras de alto nivel (loops, variables) a partir de instrucciones de bajo nivel.

Decisiones de diseño

Una decisión clave del diseño fue desacoplar la generación del *trace* de su análisis posterior.

Esto permite:

- Reutilizar el mismo *trace* para distintos análisis
- Simplificar la instrumentación
- Facilitar la extensión de la herramienta

Asimismo, se optó por utilizar identificadores internos para clases, métodos y estructuras, junto con archivos de mapeo que permiten reconstruir la correspondencia con el código fuente.

Dificultades encontradas

Durante el desarrollo se identificaron diversas dificultades:

- Dependencia del input en el análisis dinámico, lo que puede ocultar dependencias reales
- Problemas de inicialización de memoria que afectan la clasificación de dependencias
- Dificultades en la detección de accesos a arrays dependiendo del mecanismo de instrumentación utilizado
- Complejidad en la reconstrucción de estructuras de alto nivel a partir del bytecode

Estas limitaciones refuerzan la necesidad de interpretar los resultados del análisis como una ayuda al programador, y no como una caracterización completa del comportamiento del programa.

5.2.8. Resultados cualitativos

Los resultados obtenidos muestran:

- **Dependencias RAW:** detectadas correctamente, indicando restricciones reales de paralelización
- **Dependencias WAR/WAW:** identificadas como dependencias falsas, resolubles mediante transformaciones

- **Dependencias RAR:** clasificadas como no conflictivas

Estos resultados confirman que el enfoque detecta y clasifica dependencias dinámicas de manera consistente y acorde al comportamiento esperado del programa.

5.2.9. Discusión (evaluación cualitativa)

El análisis mediante ejemplos muestra que el enfoque permite detectar e interpretar dependencias dinámicas en términos de paralelización, distinguiendo entre restricciones reales y oportunidades potenciales de optimización dentro de los escenarios evaluados.

Como limitación, los resultados dependen de los datos de entrada, característica inherente al análisis dinámico.

5.3. Evaluación cuantitativa con benchmarks

Se realizó una evaluación utilizando benchmarks de la suite JavaGrande con el objetivo de medir el overhead y analizar la escalabilidad del enfoque.

La suite JavaGrande fue seleccionada por tratarse de un conjunto de benchmarks clásico en el estudio de paralelización, con aplicaciones numéricas representativas y ampliamente utilizado en la literatura.

5.3.1. Metodología experimental

Se utilizaron los siguientes benchmarks:

- SOR
- LUFact
- SparseMatmult
- Series
- Crypt

Para cada uno se ejecutó:

- Versión sin instrumentación
- Versión instrumentada

La ejecución de los benchmarks y la recolección de métricas se automatizó mediante scripts que permiten ejecutar versiones baseline e instrumentadas, así como procesar los resultados de manera sistemática. Este enfoque permitió garantizar la reproducibilidad de los experimentos y facilitar la comparación entre configuraciones.

Se midieron:

- Tiempo de ejecución
- Consumo de memoria
- Tamaño de trazas
- Dependencias detectadas

5.3.2. Resultados (tamaño A)

Benchmark	Time Base (s)	Time Instr (s)	Overhead
SOR	0.81	27.94	34.49x
LUFact	0.02	8.20	410.00x
Sparse	0.12	14.41	120.08x
Series	0.93	5.28	5.68x
Crypt	0.11	3.88	35.27x

Tab. 5.1: Overhead de ejecución (tamaño A)

Benchmark	Mem Base (MB)	Mem Instr (MB)	Trace (MB)
SOR	47.3	135.3	9234
LUFact	35.3	144.5	2526
Sparse	44.8	147.9	4649
Series	40.4	132.1	1069
Crypt	50.3	133.8	1184

Tab. 5.2: Memoria y tamaño de trazas (tamaño A)

Los resultados muestran un overhead significativo y un crecimiento importante del tamaño de las trazas, evidenciando el costo asociado al análisis dinámico a nivel de accesos a memoria.

En términos relativos, este incremento representa factores que van desde 2.6x (en el caso de Crypt) hasta superar el 4x (en el caso de LUFact) respecto de la ejecución base, lo que evidencia el costo asociado a la instrumentación dinámica, el mantenimiento de metadatos auxiliares durante la ejecución y la generación de trazas de acceso a memoria.

Este comportamiento no es uniforme entre los distintos benchmarks y puede explicarse a partir de sus patrones de acceso a memoria. En particular, LUFact presenta el mayor overhead debido a la alta densidad de accesos sobre estructuras de datos intensivas, lo que genera una gran cantidad de eventos instrumentados. Por el contrario, Series muestra un overhead significativamente menor, lo que sugiere un patrón de acceso más regular o con menor cantidad de accesos instrumentados por unidad de cómputo. En el caso de SOR, el elevado tamaño de las trazas y su overhead asociado pueden atribuirse a su naturaleza iterativa sobre matrices, donde múltiples accesos a memoria compartida se repiten en cada iteración.

5.3.3. Resultados (tamaño B)

Benchmark	Time Base (s)	Time Instr (s)	Overhead
SOR	1.70	65.43	38.49x
LUFact	0.17	64.12	377.18x
Sparse	0.20	28.68	143.40x
Series	9.39	51.18	5.45x
Crypt	0.56	24.76	44.21x

Tab. 5.3: Overhead de ejecución (tamaño B)

Se observa un incremento significativo tanto en el tiempo de ejecución como en el tamaño de las trazas, evidenciando el costo asociado al análisis dinámico.

Benchmark	Mem Base (MB)	Mem Instr (MB)	Trace (MB)
SOR	57.1	146.8	20798
LUFact	49.5	166.1	19641
Sparse	48.8	154.5	9298
Series	41.5	163.5	10694
Crypt	98.3	279.3	7896

Tab. 5.4: Memoria y tamaño de trazas (tamaño B)

El comportamiento diferencial entre benchmarks también se mantiene en este tamaño. En particular, el crecimiento del tamaño de las trazas varía considerablemente según el tipo de acceso a memoria: mientras que en Series el crecimiento es cercano a un orden de magnitud, en otros benchmarks como SOR o SparseMatmult el incremento es más moderado. Esto sugiere que la cantidad de eventos generados no depende únicamente del tamaño del input, sino también de la estructura del algoritmo y de cómo se accede a los datos en memoria.

El incremento en el consumo de memoria observado en las ejecuciones instrumentadas puede explicarse por la necesidad de mantener estructuras auxiliares asociadas a la instrumentación, buffers de escritura de trazas y metadatos utilizados por el agente durante la ejecución.

Este comportamiento es consistente con los valores observados en las tablas, donde el consumo de memoria instrumentado se incrementa entre aproximadamente 2x y 3x respecto de la ejecución base.

El overhead extremo observado en benchmarks como LUFact (superior a 350x) pone en evidencia una limitación importante del prototipo actual. En algoritmos con alta densidad de accesos a arreglos dentro de loops intensivos, el costo asociado a la instrumentación de operaciones de memoria domina significativamente el tiempo total de ejecución. Estos resultados refuerzan la necesidad de incorporar mecanismos de reducción de overhead, como filtrado dinámico de eventos o técnicas de muestreo, en futuras versiones de la herramienta.

Una posible explicación de este comportamiento es que la herramienta registra de manera explícita cada acceso relevante a memoria, generando eventos adicionales, estructuras auxiliares y operaciones de escritura sobre las trazas. Como consecuencia, el costo total del análisis crece con la cantidad de accesos observados durante la ejecución, afectando especialmente a algoritmos numéricos intensivos en operaciones sobre arreglos.

Entre las posibles estrategias para reducir este costo se encuentran mecanismos de muestreo de eventos, técnicas de caching para evitar consultas repetitivas sobre estructuras auxiliares, agregación parcial de información durante la captura y enfoques híbridos que combinen análisis estático y dinámico para identificar regiones del código donde la instrumentación pueda reducirse sin afectar significativamente la calidad del análisis.

5.3.4. Comparación y análisis

Los resultados obtenidos en ambos tamaños permiten analizar no sólo el costo del enfoque, sino también la estabilidad de las dependencias detectadas.

- **Crecimiento no lineal de las trazas:** el tamaño de las trazas crece significativamente, alcanzando factores de hasta 10x, dependiendo de la intensidad de accesos a memoria.

- **Overhead relativamente estable:** el overhead se mantiene en el mismo orden de magnitud, indicando que el costo está dominado por la instrumentación por acceso.
- **Dependencias estructuralmente estables:** la inspección manual de muestras de reportes sugiere que los cuellos de botella asociados a dependencias en los loops principales tienden a preservarse entre distintos tamaños de entrada.
- **Sensibilidad al patrón de acceso:** benchmarks con mayor intensidad de accesos presentan mayor overhead y volumen de trazas.

El tamaño de las trazas está directamente relacionado con la cantidad de accesos a memoria instrumentados, ya que cada acceso genera un evento registrado. En consecuencia, programas con mayor intensidad de accesos producen volúmenes de datos significativamente mayores. Este crecimiento no sólo impacta en los requerimientos de almacenamiento, sino también en el tiempo necesario para el procesamiento offline de las trazas, condicionando la escalabilidad del enfoque.

Benchmark	Trace A (MB)	Trace B (MB)	Growth	Overhead A/B
SOR	9234	20798	2.25x	34.49 / 38.49
LUFact	2526	19641	7.77x	410.00 / 377.18
Sparse	4649	9298	2.00x	120.08 / 143.40
Series	1069	10694	10.00x	5.68 / 5.45
Crypt	1184	7896	6.67x	35.27 / 44.21

Tab. 5.5: Comparación entre tamaños A y B

5.3.5. Análisis de dependencias en benchmarks

El análisis de dependencias detectadas en los benchmarks muestra comportamientos consistentes entre los distintos tamaños de entrada.

- **Estabilidad estructural:** los reportes obtenidos sugieren que los patrones principales de dependencias detectados se mantienen relativamente consistentes entre distintos tamaños de entrada, aunque el trabajo no realiza una cuantificación exhaustiva de todas las dependencias observadas.
- **Predominio de RAW:** en benchmarks numéricos, reflejando dependencias entre iteraciones. Este comportamiento tiene implicancias directas en términos de paralelización. En particular, en benchmarks como SOR o LUFact, la presencia predominante de dependencias RAW entre iteraciones indica que los loops no pueden paralelizarse directamente sin modificaciones adicionales. En contraste, en casos como Series, donde la cantidad de dependencias es menor, se observa un mayor potencial de paralelización. De esta manera, el análisis no sólo detecta dependencias, sino que permite inferir el grado de paralelismo explotable en cada caso.
- **Dependencias falsas:** presencia de WAR/WAW que indican oportunidades de optimización.
- **Errores de memoria (OutOfMemoryError):** observados en ejecuciones de tamaño B, asociados al crecimiento de estructuras auxiliares utilizadas por el Java

Agent durante la instrumentación, al mantenimiento de metadatos de ejecución y al tamaño de las trazas generadas. Este comportamiento evidencia que la densidad de recolección del prototipo actual genera una saturación de recursos en ejecuciones prolongadas o con iteraciones masivas, marcando un límite de escalabilidad operativa para la infraestructura de trazado utilizada.

5.3.6. Discusión (evaluación cuantitativa)

Los resultados cuantitativos evidencian que el análisis dinámico introduce un costo significativo en tiempo y espacio, consistente con su nivel de detalle.

Adicionalmente, se observaron limitaciones prácticas relevantes:

- **Errores de memoria (OutOfMemoryError)**
- **Trazas extremadamente grandes (hasta decenas de GB)**
- **Alto costo de procesamiento offline**
- **Dependencia de recursos de hardware**

Estos resultados reflejan un trade-off inherente al análisis dinámico: a mayor precisión en la captura del comportamiento real del programa, mayor es el costo computacional asociado. Este comportamiento se debe al seguimiento detallado de accesos a memoria requerido por el enfoque. En este sentido, la herramienta se posiciona como un mecanismo de análisis detallado, más que como una técnica de uso directo en entornos productivos sin optimizaciones adicionales.

5.4. Conclusión del capítulo

Los resultados obtenidos permiten validar el enfoque propuesto desde múltiples dimensiones. Los ejemplos confirman la correctitud e interpretabilidad del análisis, mientras que los benchmarks demuestran su aplicabilidad en programas reales.

Asimismo, los resultados sugieren que ciertos patrones de dependencias observados tienden a mantenerse entre distintos tamaños de entrada evaluados, aunque el análisis continúa dependiendo de las ejecuciones concretas consideradas.

Sin embargo, el costo en tiempo, memoria y volumen de datos plantea desafíos prácticos que deberán ser abordados en futuras extensiones del trabajo.

La evaluación realizada se centra principalmente en benchmarks numéricos basados en arreglos y loops intensivos, por lo que no permite extrapolar directamente el comportamiento del enfoque a aplicaciones Java modernas con uso intensivo de polimorfismo dinámico, reflection o estructuras complejas de objetos.

En conjunto, el enfoque resulta efectivo como herramienta de apoyo al programador para la identificación de oportunidades de paralelización, aun considerando sus limitaciones.

En este sentido, los resultados obtenidos contribuyen a responder la pregunta de investigación en un contexto práctico, mostrando que el análisis dinámico a nivel de bytecode es capaz de identificar oportunidades de paralelización de manera precisa e interpretable, aunque con limitaciones en términos de costo computacional.

5.5. Exploraciones adicionales

Con el objetivo de evaluar el comportamiento del enfoque en contextos más complejos, se realizaron experimentos sobre otras benchmarks de suites estándar como Renaissance y DaCapo.

5.5.1. Aplicación a entornos modernos de la JVM

Se exploró la aplicación del enfoque sobre benchmarks de la suite *Renaissance*, orientados a cargas de trabajo modernas sobre la JVM. En particular, se intentó ejecutar el benchmark *dotty*, correspondiente al compilador Scala.

La instrumentación produjo errores de verificación de bytecode (*VerifyError*), impidiendo completar la ejecución. Si bien el experimento evidencia una limitación de la implementación actual al instrumentar este tipo de aplicaciones, no se realizó un análisis exhaustivo que permita determinar si el origen del problema se encuentra en restricciones del mecanismo de instrumentación o en incompatibilidades con características presentes en versiones más modernas del ecosistema JVM.

Este resultado evidencia que la herramienta presenta limitaciones al aplicarse sobre entornos modernos de la JVM, especialmente cuando el bytecode es generado por lenguajes distintos de Java o incorpora optimizaciones avanzadas.

En cualquier caso, el resultado pone de manifiesto la necesidad de extender y modernizar la infraestructura de instrumentación utilizada por el prototipo para soportar de manera más robusta bytecode generado por compiladores y herramientas actuales de la plataforma JVM.

5.5.2. Exploración con benchmarks de DaCapo

Como complemento a la evaluación, se exploró la aplicación del enfoque sobre benchmarks de la suite *DaCapo*, ampliamente utilizada en la evaluación de aplicaciones Java reales.

Se analizaron distintos benchmarks, observándose comportamientos diversos. En particular, en el benchmark *fop* la herramienta logró instrumentar la ejecución y detectar dependencias dinámicas en loops. En este caso, se identificaron loops con dependencias de tipo RAR, lo que indica ausencia de conflictos entre iteraciones y potencial paralelismo.

Sin embargo, también se observaron limitaciones. En algunos casos, la instrumentación modifica el comportamiento observable del programa, afectando los mecanismos de validación de la suite. Asimismo, ciertos benchmarks presentan estructuras de bytecode que dificultan la instrumentación o el análisis.

Estos resultados muestran que el enfoque puede aplicarse parcialmente a aplicaciones reales, aunque su efectividad depende de las características del bytecode y de la complejidad del programa analizado.

Adicionalmente, se observaron limitaciones asociadas a la presencia de bytecode legacy, en particular instrucciones como *JSR* y *RET*, utilizadas en versiones antiguas de Java para la implementación de subrutinas. Estas estructuras no son compatibles con el modelo de análisis adoptado, por lo que quedan fuera del alcance del enfoque.

Estos experimentos permiten observar el comportamiento del enfoque en aplicaciones reales, aunque no forman parte de la evaluación principal presentada en este trabajo.

6. EVOLUCIÓN DEL ANÁLISIS DINÁMICO DE DEPENDENCIAS

Este capítulo complementa el estado del arte presentado en la introducción, ofreciendo una perspectiva histórica sobre la evolución del análisis dinámico de dependencias desde los primeros enfoques hasta las herramientas actuales, incorporando además avances recientes en técnicas de profiling dinámico y análisis en la JVM.

6.1. Introducción

El análisis de dependencias es un componente central en la optimización de programas y en la identificación de oportunidades de paralelización. Tradicionalmente, este análisis se ha abordado mediante técnicas estáticas en compiladores, como describe Muchnick (1997) [11], donde las dependencias se infieren a partir del código fuente o intermedio.

Sin embargo, estas técnicas presentan limitaciones importantes frente a aliasing, estructuras dinámicas y dependencias dependientes de los datos de ejecución. En estos casos, el análisis estático tiende a ser conservador, reportando dependencias potenciales que no necesariamente ocurren en una ejecución real.

Como alternativa, surge el análisis dinámico de dependencias, donde las dependencias se obtienen a partir de la ejecución concreta del programa mediante instrumentación y registro de accesos a memoria. Este enfoque permite capturar dependencias efectivas, observadas en tiempo de ejecución.

6.2. Análisis dinámico de dependencias

El análisis dinámico de dependencias se basa en observar accesos a memoria durante la ejecución del programa y detectar relaciones entre ellos. Una dependencia ocurre cuando dos accesos refieren a la misma dirección de memoria y al menos uno de ellos es una escritura.

Ketterlin (2010) [7] propone estructurar estos accesos mediante puntos de ejecución (*execution points*), que permiten agrupar eventos dinámicos en unidades asociadas a estructuras del programa como métodos o iteraciones de loops. Esta contribución resulta clave, ya que introduce una forma de elevar el nivel de abstracción del análisis dinámico, facilitando su interpretación.

Por otro lado, Ball y Larus (1996) [1] introducen técnicas de profiling dinámico basadas en caminos de ejecución (*path profiling*), permitiendo identificar patrones frecuentes en la ejecución real de programas. Este trabajo sienta las bases para el análisis empírico del comportamiento de programas, que posteriormente sería aprovechado por técnicas de análisis dinámico de dependencias.

6.3. Evolución del análisis dinámico de dependencias

6.3.1. Primeras aplicaciones en paralelización

El análisis dinámico de dependencias ha sido inicialmente explorado en el contexto de la paralelización automática y especulativa.

Steffan y Mowry (2000) [18] introducen el modelo de *Thread-Level Speculation* (TLS), donde regiones del programa se ejecutan en paralelo de forma especulativa, validando dinámicamente las dependencias. Este enfoque permite explotar paralelismo aun cuando las dependencias no son completamente conocidas en tiempo de compilación.

En esta línea, Ketterlin y Clauss (2012) [8] proponen el profiling dinámico de dependencias como herramienta para asistir al programador en la identificación de paralelismo potencial. Este enfoque demuestra que el análisis dinámico puede capturar dependencias reales con mayor precisión que los enfoques estáticos, especialmente en presencia de estructuras dinámicas y aliasing.

Sin embargo, estos enfoques presentan limitaciones prácticas, principalmente debido al alto costo de instrumentación y a la complejidad de gestionar conflictos en tiempo de ejecución.

6.3.2. Consolidación en debugging y profiling

Durante la década de 2010 y comienzos de 2020, el análisis dinámico de dependencias se consolidó principalmente en herramientas orientadas al debugging y profiling, con un fuerte énfasis en la reducción del overhead y la escalabilidad.

Flanagan y Freund (2009) [5] proponen FastTrack, una técnica eficiente para la detección dinámica de *data races*, basada en relojes vectoriales optimizados. Este trabajo reduce significativamente el overhead respecto de enfoques anteriores.

Serebryany et al. (2012) [17] desarrollan ThreadSanitizer, una herramienta ampliamente adoptada en entornos industriales para detectar condiciones de carrera en programas concurrentes.

Asimismo, Bond et al. (2013) [2] introducen Octet, un sistema que permite capturar dependencias entre threads con bajo overhead, consolidando el uso del análisis dinámico en el contexto de concurrencia.

En paralelo, el ecosistema Java evolucionó con herramientas como ASM [15], Byte Buddy [19] y DiSL [10], que permiten instrumentación eficiente a nivel de bytecode. En particular, DiSL propone un framework de instrumentación declarativa y modular para la JVM, facilitando la inserción de lógica de análisis en puntos específicos del programa sin modificar directamente el código fuente.

Asimismo, soluciones integradas en la JVM como Java Flight Recorder (JFR) [14] ofrecen capacidades de profiling de bajo overhead orientadas a observabilidad en producción.

Estos avances reflejan una transición desde enfoques orientados a paralelización hacia herramientas centradas en debugging y análisis de performance.

6.3.3. Líneas recientes y vigencia del problema (2024–2026)

En particular, se observa un retorno del análisis dinámico hacia su motivación original: la identificación de oportunidades de paralelización, ahora en el contexto de sistemas complejos como la JVM moderna.

Nahian y Demsky (2024) [12] proponen FlowProf, un enfoque basado en flujo de información (*information-flow tracking*) para el profiling de programas multihilo. A diferencia de enfoques tradicionales que registran cada acceso a memoria, FlowProf modela cómo la información fluye entre variables, reduciendo significativamente el overhead del análisis. Esta contribución resulta particularmente relevante, ya que aborda uno de los principales desafíos históricos del análisis dinámico: su costo. Sin embargo, este enfoque prioriza la

eficiencia del profiling sobre la reconstrucción explícita de dependencias a nivel estructural, lo cual plantea desafíos para su aplicación directa en la reconstrucción explícita de dependencias estructurales a nivel de loops.

Por otro lado, Carvalho et al. (2024) [3] presentan técnicas de trazado automático (*automatic tracing*) en sistemas basados en tareas. En este enfoque, se identifican patrones repetitivos en la ejecución y se reutiliza información previamente analizada, evitando recomputaciones innecesarias. Este cambio introduce una mejora en la escalabilidad del análisis dinámico. No obstante, estos enfoques no establecen explícitamente una relación entre las dependencias dinámicas y estructuras de alto nivel como loops o métodos.

La vectorización consiste en la ejecución simultánea de múltiples operaciones sobre conjuntos de datos utilizando instrucciones SIMD (Single Instruction, Multiple Data). En el contexto de loops, esto implica procesar varias iteraciones en paralelo mediante operaciones vectoriales.

Sin embargo, este tipo de optimización solo es posible cuando no existen dependencias entre las iteraciones del loop, lo que convierte al análisis de dependencias en un componente fundamental para su aplicación.

Trabajos recientes en esta línea han explorado la inferencia dinámica de oportunidades de vectorización en aplicaciones Java a partir de trazas de ejecución, como en la familia de enfoques representados por herramientas tipo JVProf (ca. 2025–2026) [16].

Estos sistemas utilizan instrumentación a nivel de bytecode para reconstruir dependencias dinámicas y detectar regiones donde es posible relajar el orden de ejecución, permitiendo la aplicación de optimizaciones SIMD.

A diferencia de enfoques tradicionales de vectorización estática, estos métodos se basan en dependencias efectivas observadas en ejecución, lo que les permite identificar paralelismo oculto en presencia de aliasing o estructuras dinámicas.

En particular, la identificación de oportunidades de vectorización depende directamente de la ausencia de dependencias entre iteraciones, lo que refuerza la relación entre estos enfoques y el análisis dinámico de dependencias desarrollado en esta tesis.

Estos enfoques evidencian la vigencia del análisis dinámico como herramienta para descubrir paralelismo oculto. Sin embargo, se enfocan principalmente en la generación automática de optimizaciones SIMD en el backend de la JVM, sin exponer explícitamente reportes estructurados de dependencias orientados al programador.

No obstante, mientras estos enfoques se enfocan en la detección de paralelismo a nivel de instrucciones, el enfoque propuesto en esta tesis se orienta a la identificación y estructuración de dependencias a nivel de loops, con el objetivo de asistir al programador en la paralelización manual del código.

En conjunto, estos avances reflejan una tendencia clara: el análisis dinámico continúa siendo una herramienta poderosa, pero su adopción depende críticamente de la reducción del overhead y de la capacidad de interpretar los resultados.

6.3.4. Contexto de la JVM moderna

La evolución de la JVM introduce nuevos desafíos para el análisis dinámico.

En primer lugar, la compilación Just-In-Time (JIT) transforma dinámicamente el código ejecutado, lo que introduce una limitación metodológica inherente a este tipo de enfoques. La instrumentación a nivel de bytecode puede inhibir optimizaciones agresivas del compilador JIT, como inlining o vectorización automática, generando diferencias entre el

comportamiento del programa instrumentado y una ejecución altamente optimizada en un entorno de producción.

En segundo lugar, técnicas como escape analysis, introducidas originalmente por Choi et al. (1999) [4] y ampliamente utilizadas en implementaciones modernas de la JVM (por ejemplo, en HotSpot), permiten eliminar asignaciones en el heap, reduciendo accesos a memoria y afectando la visibilidad de dependencias.

Finalmente, Project Loom (OpenJDK, 2024) [13] introduce hilos virtuales, modificando el modelo de concurrencia y planteando nuevos desafíos para herramientas de análisis dinámico en términos de escalabilidad.

6.3.5. Situación actual y brecha existente

Actualmente, el análisis dinámico permite capturar con gran precisión el comportamiento real de los programas.

Sin embargo, diversos trabajos han señalado que la utilidad práctica de esta información depende críticamente de su interpretabilidad a nivel de estructuras del programa. En particular, enfoques como los de Ketterlin y Clauss [8] destacan la necesidad de asociar dependencias dinámicas con construcciones como loops para asistir efectivamente al programador en la paralelización.

Asimismo, herramientas de análisis y paralelización asistida, como SUIF Explorer [9], muestran que la visualización y estructuración de dependencias es clave para su utilización práctica.

En este contexto, persiste una brecha entre la información recolectada a bajo nivel (accesos a memoria, eventos de ejecución) y su interpretación en términos de estructuras de alto nivel como loops. Esta limitación reduce la utilidad práctica del análisis dinámico en el contexto de paralelización manual, donde el programador necesita comprender cómo las dependencias afectan estructuras específicas del código.

6.4. Gap en la literatura

A partir del análisis presentado, se identifican las siguientes limitaciones en el estado del arte:

- Las herramientas modernas priorizan la eficiencia del profiling (reducción de overhead, escalabilidad), pero sacrifican la interpretabilidad de los resultados.
- Los enfoques recientes basados en information-flow y tracing automático priorizan la reducción del overhead y la reutilización eficiente de información dinámica, pero no se enfocan específicamente en la interpretación estructural de dependencias sobre construcciones algorítmicas locales, como loops o métodos.
- Los trabajos orientados a vectorización en la JVM utilizan dependencias dinámicas para detectar paralelismo, pero se enfocan en paralelismo a nivel de instrucciones, sin ofrecer una visión estructural útil para el programador.
- En consecuencia, existe una brecha entre la información recolectada a bajo nivel (accesos a memoria, trazas de ejecución) y su utilización efectiva para asistir al programador en la identificación de oportunidades de paralelización a nivel de código fuente.

6.5. Síntesis

El análisis dinámico de dependencias ha evolucionado significativamente en términos de precisión, eficiencia y aplicabilidad.

Sin embargo, su utilización para asistir directamente al programador en la identificación de oportunidades de paralelización sigue siendo limitada.

En este contexto, el enfoque propuesto en esta tesis busca reducir esta brecha, combinando un análisis dinámico a nivel de bytecode con una interpretación estructural basada en loops, permitiendo transformar información de bajo nivel en conocimiento útil para la toma de decisiones de paralelización.

7. CONCLUSIONES, APORTES Y TRABAJO FUTURO

Do. Or do not. There is no try.
— Yoda

7.1. Contribuciones

Este trabajo propone un enfoque integrado de análisis dinámico de dependencias que combina instrumentación a nivel de bytecode con una interpretación estructural del programa, permitiendo vincular información de bajo nivel con estructuras de alto nivel como loops.

Las principales contribuciones de este trabajo son:

- El diseño e implementación de un enfoque de análisis dinámico de dependencias a nivel de bytecode para programas Java.
- La definición de un modelo de ejecución que representa el contexto dinámico mediante estructuras como métodos y loops.
- Un mecanismo para detectar y clasificar dependencias dinámicas (RAW, WAR, WAW, RAR) a partir de accesos reales a memoria.
- La asociación explícita de dependencias a estructuras de loops, permitiendo evaluar su impacto en la paralelización.
- Una herramienta funcional que integra instrumentación, generación de *traces* y análisis posterior.
- Un enfoque que permite interpretar dependencias dinámicas en el contexto de estructuras del programa como loops, reduciendo la brecha entre análisis de bajo nivel y decisiones a nivel de código fuente.

7.2. Respuesta a la pregunta de investigación

En la introducción de este trabajo se planteó la siguiente pregunta de investigación:

> ¿Es posible utilizar análisis dinámico de dependencias a nivel de bytecode para identificar oportunidades de paralelización en loops de programas Java de una manera útil e interpretable para el programador?

A partir de los resultados obtenidos, es posible responder afirmativamente a esta pregunta en el contexto de los escenarios analizados.

El enfoque propuesto permite detectar dependencias dinámicas reales observadas durante la ejecución del programa, clasificarlas según su tipo (RAW, WAR, WAW, RAR) y asociarlas a estructuras de alto nivel como loops, con el objetivo de analizar su impacto en la paralelización.

Esta información resulta útil para el programador, ya que permite identificar regiones potencialmente paralelizables, así como aquellas que requieren transformaciones adicionales para eliminar dependencias.

En particular, la ausencia de dependencias verdaderas (RAW) y la presencia exclusiva de accesos de lectura (RAR) constituyen indicios favorables para la paralelización, aunque su validez debe interpretarse en el contexto de la ejecución observada.

Dado que se trata de un análisis dinámico, los resultados dependen de los datos de entrada utilizados durante la ejecución. En consecuencia, deben interpretarse como evidencia empírica del comportamiento del programa, y no como una caracterización exhaustiva de todos los posibles escenarios, lo cual es consistente con la naturaleza del enfoque adoptado.

Esta limitación implica que el enfoque no es sound en el sentido clásico de los análisis de programas: la ausencia de dependencias observadas en una ejecución particular no garantiza su ausencia para todas las entradas o caminos de ejecución posibles. Por este motivo, los resultados deben interpretarse como una herramienta de apoyo para sugerir oportunidades de optimización y paralelización, y no como una prueba formal de que dichas transformaciones sean correctas en todos los escenarios.

En este marco, la propuesta se posiciona como una herramienta de apoyo al programador, que complementa otras técnicas de análisis y optimización.

En conjunto, los resultados muestran que el análisis dinámico de dependencias a nivel de bytecode no sólo es viable, sino que puede estructurarse de manera tal que resulte interpretable y útil para la identificación de oportunidades de paralelización, aportando una interpretación estructural que no suele encontrarse explícitamente en herramientas tradicionales de profiling dinámico.

Finalmente, el enfoque presentado aporta una forma concreta y estructurada de interpretar el comportamiento dinámico de programas en términos de paralelización, logrando vincular dependencias observadas en tiempo de ejecución con decisiones prácticas a nivel de código fuente.

7.3. Revalidación experimental de la herramienta

Como parte del presente trabajo, se llevó a cabo una revalidación experimental de la herramienta desarrollada originalmente, con el objetivo de recuperar su capacidad operativa y evaluar su funcionamiento en entornos actuales.

Dado que la implementación original se basaba en versiones antiguas de la máquina virtual de Java y librerías de instrumentación de bytecode, fue necesario realizar tareas de adaptación y configuración para lograr nuevamente la ejecución del pipeline completo de análisis dinámico, incluyendo la instrumentación del bytecode, la generación de *traces* y el procesamiento de accesos a memoria y estructuras de control.

Esta revalidación permitió volver a ejecutar los experimentos y analizar el comportamiento del enfoque propuesto en la tesis, poniendo además en evidencia algunas limitaciones prácticas de la implementación original, como el volumen de datos generado y la necesidad de mejorar las herramientas de visualización y análisis de resultados.

En conjunto, esta experiencia mostró que el enfoque continúa siendo aplicable en entornos actuales y permitió identificar posibles líneas de trabajo futuro relacionadas con la modernización de la infraestructura y la mejora de las herramientas de soporte al análisis.

7.4. Trabajo futuro

A partir de los resultados obtenidos y de la experiencia en la revalidación de la herramienta, se identifican diversas líneas de trabajo futuro orientadas a mejorar tanto la escalabilidad del enfoque como su utilidad práctica.

En primer lugar, resulta relevante abordar el problema del volumen de datos generado por las *traces* de ejecución. En línea con trabajos recientes como FlowProf [12], una dirección prometedora consiste en explorar técnicas de reducción de información, como el filtrado selectivo de eventos o la representación de dependencias mediante modelos más compactos, con el objetivo de disminuir el overhead sin perder precisión relevante para el análisis.

En segundo lugar, una extensión natural del trabajo consiste en profundizar la interpretación estructural de las dependencias dinámicas. En particular, resulta de interés avanzar hacia la clasificación automática de loops en términos de su potencial de paralelización, incorporando criterios derivados del análisis dinámico. Esta línea apunta a cerrar el ciclo entre la detección de dependencias y la toma de decisiones concretas por parte del programador.

Asimismo, el enfoque podría complementarse mediante la integración con herramientas modernas de análisis estático y profiling dinámico en la JVM, permitiendo combinar información proveniente de múltiples fuentes. En esta línea, trabajos recientes han mostrado el valor de reutilizar información de ejecución y reducir la redundancia en el análisis, como en el caso de Apophenia [3]. Integrar este tipo de ideas permitiría mejorar la eficiencia del enfoque propuesto sin sacrificar su capacidad de interpretación estructural.

Otra línea de trabajo futuro consiste en mejorar la robustez del proceso de instrumentación para aumentar su compatibilidad con bytecode moderno generado por compiladores y herramientas actuales de la plataforma JVM. Esto permitiría extender el enfoque a aplicaciones desarrolladas en lenguajes como Scala o Kotlin, así como a entornos que incorporan transformaciones y optimizaciones más complejas que las consideradas por la implementación actual.

Por último, una línea de investigación relevante consiste en explorar la aplicación del enfoque en contextos más complejos, como programas concurrentes o modelos de ejecución basados en tareas. En estos escenarios, la identificación de dependencias dinámicas adquiere un rol aún más crítico, y su interpretación a nivel estructural podría contribuir a mejorar la comprensión y optimización del comportamiento del programa.

Bibliografía

[1] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1996.

[2] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. Octet: Capturing and controlling cross-thread dependences efficiently. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[3] Pedro Carvalho et al. Automatic tracing in task-based runtime systems. *arXiv preprint arXiv:2406.18111*, 2024.

[4] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Samuel Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[5] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[6] John Giacomoni, Tipp Moseley, and Neil Vachharajani. Visualizing potential parallelism in sequential programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[7] Alain Ketterlin. Dynamic dependence analysis, 2010. INRIA Technical Report / internal communication.

[8] Alain Ketterlin and Philippe Clauss. Profiling data dependence to assist parallelization: Framework, scope and optimization. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012.

[9] Shih-Wei Liao, Amer Diwan, Robert P Bosch, Anwar Murphy, and Monica S Lam. Suif explorer: An interactive and interprocedural parallelizer. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1999.

[10] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: A domain-specific language for bytecode instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development (AOSD)*, 2012.

[11] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[12] Md Nahian and Brian Demsky. Flowprof: Profiling multi-threaded programs using information flow. In *Proceedings of the 33rd International Conference on Compiler Construction (CC)*, 2024.

[13] OpenJDK Community. Project loom. `https://openjdk.org/projects/loom/`, 2024.

[14] Oracle Corporation. Java flight recorder. `https://docs.oracle.com/en/java/javase/17/jfapi/introduction.html`, 2023. Accessed: 2026-03.

[15] OW2 Consortium. Asm: A java bytecode engineering library. `https://asm.ow2.io/`, 2023. Accessed: 2026-03.

[16] Andrea Rosà, Joy Albertini, Júnior Löff, and Walter Binder. Jvprof: Locating vectorization candidates in jvm applications. In *Thirteenth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 380–382. IEEE, 2025.

[17] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2012.

[18] J. Gregory Steffan and Todd C. Mowry. Thread-level speculation for general purpose programs. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000.

[19] Rafael Winterhalter. Byte buddy. `https://bytebuddy.net/`, 2024. Runtime code generation for the Java virtual machine. Accessed: 2026-03.

[20] Xiangyu Zhang, Arash Navabi, and Suresh Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2009.