Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Banco de pruebas poliédrico en Python

Estudiante: Miguel Ángel Abella González
Dirección: Gabriel Rodríguez Álvarez

A Coruña, setembro de 2020.

Resumen

En este trabajo se estudia una aplicación de los métodos de optimización poliédrica, que se basa en aplicar métodos matemáticos sobre estructuras de código afines que se caracterizan por utilizar bucles regulares de gran tamaño en donde el control y los datos dependen únicamente de las variables de inducción del bucle y constantes mediante funciones afines. Estas regiones, que se suelen llamar Static Control Parts (SCoPs), se modelan y optimizan usando técnicas de compilación poliédrica.

El objetivo principal de este trabajo consiste en portar las aplicaciones de PolyBench/C, que conforman un conjunto de pruebas de rendimiento (benchmarks) empleadas para el desarrollo y validación de técnicas de optimización poliédrica en el lenguaje de programación C, al lenguaje de programación Python para de forma similar conformar un banco de pruebas estándar de cara al futuro desarrollo de optimizaciones poliédricas en este lenguaje.

Abstract

This paper explores an application of polihedral optimization, which consists on using mathematical methods on affine code structures which are characterized by using large regular loops where control and data depend solely on loop induction variables and constants using affine functions. These regions, often called Static Control Parts (SCoPs), are modeled and optimized using polyhedral compilation.

The main objective of this work is porting the applications present in PolyBench/C, which form a set of performance tests (benchmarks) used for the development and validation of polyhedric optimization techniques for the C programming language, to the Python programming language to form in a similar manner a standard test bench for future development of polihedral optimizations on this language.

Palabras clave:

- Código fuente
- Compiladores
- Lenguajes de programación
- Optimización

Keywords:

- Compilers
- Computer languages
- Optimization
- Source code

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Las técnicas de optimización poliédrica permiten mejorar el rendimiento de aplicaciones que presentan códigos afines, como aplicaciones de supercomputación, sistemas empotrados y aplicaciones multimedia. Los códigos afines se caracterizan por ejecutar bucles de gran tamaño en dónde el control y los datos dependen únicamente de las variables de inducción del bucle y de constantes mediante funciones afines.

El benchmarking es el proceso por el cual se evalúa el rendimiento de un sistema mediante la elaboración de benchmarks. En el campo de la optimización de código el benchmarking permite evaluar la eficacia de las distintas técnicas de optimización que se aplican sobre un programa de software. Para ello, las técnicas de benchmarking ofrecen una serie de métricas estandarizadas que permiten comparar las distintas variaciones de un programa.

PolyBench es un conjunto de pruebas de rendimiento para la evaluación de optimizaciones poliédricas que implementa un total de 30 núcleos computacionales diferentes que presentan códigos afines. PolyBench es actualmente el estándar de facto para la evaluación de optimizaciones poliédricas y se encuentra disponible en el lenguaje de programación C. El objetivo de este trabajo consiste en portar las aplicaciones de PolyBench de C a Python con el fin de conformar un banco de pruebas estándar de cara al futuro desarrollo de optimizaciones poliédricas en Python.

1.1 Estructura del trabajo

El transcurso de este trabajo se divide en 6 capítulos que introducen los conceptos necesarios de forma secuencial.

El capítulo 2, *Base teórica*, introduce la base teórica y los conceptos generales en los que se apoya este trabajo. En este capítulo se explica de forma breve el modelo poliédrico en el que se basan las técnicas de optimización poliédrica y se da una definición de benchmarking más extensa.

En el capítulo 3, *Estado del arte: PolyBench/C*, se realiza un estudio al detalle de PolyBench/C, el banco de pruebas de referencia para las técnicas de optimización poliédrica en el lenguaje de programación C, con el fin de poder realizar la traducción a Python.

El capítulo 4, *Diseño y desarrollo*, describe las etapas de diseño de PolyBench en Python aplicando el conocimiento visto en el capítulo 3. Una vez definidas las bases del diseño, se procede a explicar los detalles de implementación aplicando los conceptos vistos hasta el momento.

En el capítulo 5, *Resultados experimentales*, se muestran los resultados del experimento, ejecutando y comparando las distintas variaciones de PolyBench en C y Python. Finalmente, sobre los resultados obtenidos, se discuten las causas que llevan a las diferencias entre implementaciones con el fin de entender qué puntos pueden requerir una mejora.

Finalmente el capítulo 6 contiene las *Conclusiones* sobre este trabajo en dónde se comentan las conclusiones derivadas de los resultados del estudio y además se plantean futuras líneas de trabajo

Capítulo 2

Base teórica: optimización poliédrica y benchmarking

2.1 Modelo poliédrico

Desde la aparición de los primeros compiladores, la representación interna de los programas ha estado en directa correspondencia con su semántica operacional. En esta sintaxis abstracta, cada sentencia aparece sólo una vez incluso cuándo se ejecuta múltiples veces. Esta representación tiene limitaciones severas. Lo primero, limita la exactitud del análisis del programa. Por ejemplo, si una sentencia en un bucle tiene alguna relación de dependencia con otra sentencia, se pueden considerar a ambas como una única entidad mientras que la relación de dependencia puede que implique sólo a algunas de las iteraciones dinámicas de estas sentencias. Esto es particularmente común en programas basados en bucles que acceden a arrays. Lo siguiente, puede limitar la aplicabilidad de transformaciones al programa. Por ejemplo, las transformaciones de bucles operan sobre *iteraciones de sentencias individuales*. Por último, limita la expresividad de las transformaciones del programa: las transformaciones de anidamiento de bucle más significativas no se pueden expresar como actualizaciones estructurales e incrementales de la estructura del árbol del bucle.

El modelo poliédrico es una representación semántica y algebraica que combina capacidad de ánalisis, expresividad de transformación y flexibilidad para diseñar heurísticas de optimización sofisticadas. El modelo poliédrico es más próximo a la ejecución del programa que las representaciones operacionales/sintácticas ya que opera sobre iteraciones de sentencias individuales, o *instancias de sentencias*. Para cada instancia, el algoritmo optimizador computa un *mapeo* que determinará en qué instante (mapeo temporal, o *planificación*) y/o en qué procesador (mapeo espacial, o *colocación*) tiene que ejecutarse esta instancia.

El origen de este modelo data de finales de los años sesenta con el trabajo de Karp, Miller y Winograd para sistemas de programación de ecuaciones de recurrencia uniformes. Este

trabajo seminal se ha transpuesto por un lado al diseño de la matriz sistólica y por otro lado a la paralelización automática de programas, con el método del hiperplano de Lamport. Hasta principios de los años noventa, la mayoría de las técnicas posteriores de optimización y paralelización dependieron de transformaciones sintácticas de bucles en lugar de la planificación afín. Pero también se basaron en una precisión de abstracción de dependencia de datos cada vez mayor, siempre definida como alguna clase de conjuntos afines convexos. La convergencia de la representación del programa, abstracción de dependencias y mapeo, todas expresadas utilizando conjuntos afines, finalmente lleva al modelo poliédrico (también referido en la literatura como *modelo del politopo*). Este modelo ha sido la base de grandes avances en la optimización y paralelización automática de programas. Después de décadas de desarrollo, los compiladores de producción se aproximan cada vez más a hacer un uso efectivo del modelo poliédrico para compilar contra arquitecturas multinúcleo, incluyendo GCC [1], LLVM [2], IBM XL [3] y el compilador de alto nivel R-Stream [4] de Reservoir Labs Inc.

Aproximadamente, la reestructuración de un programa en el modelo poliédrico es un proceso en tres pasos. Primero, un programa (un código fuente o su árbol sintáctico abstracto) que encaje en el modelo se traduce a la representación poliédrica, después un algoritmo optimizador computa una secuencia de transformaciones de optimización *en el modelo*, finalmente el modelo se traduce de nuevo a un programa. Las expectativas son (1) facilitar la extracción de las propiedades del programa (por ejemplo, reutilización de datos, paralelismo) a través de la representación matemática, (2) computar una secuencia de transformaciones de optimización y paralelización en el modelo para explotar esas propiedades usando técnicas de programación lineal clásicas y (3) aplicar esas optimizaciones como un paso único y directo de transformación del modelo.

Las partes de un programa que este modelo puede codificar se llaman partes de control estático (del inglés *static control parts*, o *SCoP*).

2.2 Benchmarking

2.2.1 Introducción

Los benchmark son herramientas para la comparación de productos software y hardware. Habitualmente son empleados para la evaluación de aproximaciones metodológicas a problemas en múltiples campos de las ciencias de la computación.

Para ser aceptados para la estandarización, los benchmarks deben cumplir con una serie de criterios. Los candidatos a benchmarks deben pasar por una serie de etapas, incluyendo la definición de metodologías de medición, selección de la carga de trabajo, y varias pruebas de aceptación rigurosas.

2.2.2 Definición de benchmark

Se define benchmark como una "herramienta estándar para la evaluación y comparación competitiva de sistemas o componentes que compiten de acuerdo a características específicas, como rendimiento, fiabilidad o seguridad".

Esta definición de benchmark presenta un énfasis en la parte competitiva del benchmark, como es el caso para los benchmarks estandarizados desarrollados por SPEC y TPC.

Se definen las herramientas para la evaluación de sistemas no competitiva y de comparación como *rating tools* (herramientas de evaluación). El objetivo de las rating tools reside principalmente en métodos de evaluación con fines de investigación, programas regulatorios, o como parte de una aproximación de mejora y desarrollo de un sistema. Las rating tools pueden ser estandarizadas y en general deben seguir los mismos criterios de diseño y calidad que los benchmark.

2.2.3 Tipos de benchmark

Los benchmark, en computación, se pueden categorizar generalmente en tres tipos: basados en especificación, basados en kits, y en híbridos entre los otros dos. Los benchmark basados en especificación describen las funciones que se tienen que conseguir, parámetros de entrada requeridos y salidas esperadas. La implementación para alcanzar la especificación se deja en manos del individuo que ejecuta el benchmark. Los benchmark basados en kits proveen la implementación tal cual se requiere para la ejecución oficial del benchmark. Cualquier diferencia funcional entre los productos empleados por el benchmark debe ser resuelta *ahead of time* (antes de tiempo) y la persona responsable de ejecutar el benchmark no podrá, normalmente, alterar el flujo de ejecución del benchmark.

Los benchmark basados en especificación empiezan con la definición del problema de negocio que será simulado por el benchmark. El criterio clave para esta definición son los temas de relevancia que se describirán brevemente en el apartado 2.2.4. Los benchmarks basados en especificación tienen la ventaja de permitir al software innovador solucionar el problema de negocio demostrando los requisitos especificados de la nueva implementación. Por otra parte, este tipo de benchmark requiere un desarrollo sustancial previo a la ejecución del mismo, y puede presentar desafíos a la hora de demostrar que los requisitos del benchmark se cumplen.

Los benchmark basados en kit puede parecer que restrinjan algunas aproximaciones innovadoras al problema de negocio, pero tienen las ventajas significativas de proveer implementaciones "load and go" que reducen notablemente el coste y tiempo requeridos para la ejecución de los benchmark. Para los benchmark basados en kit, la especificación se usa como guía de diseño para la creación del kit. Para los benchmark basados en especificación, la especificación se presenta como un conjunto de reglas a seguir por un tercero que implemen-

tará y ejecutará el benchmark. Esto permite una flexibilidad sustancial en cómo se resuelve el problema de negocio del benchmark (una ventaja principal de los benchmark basados en especificación).

Un híbrido entre estos puede ser necesario si la mayor parte del benchmark se puede proveer a través de un kit, pero existe el deseo de permitir que algunas funciones se implementen bajo el criterio del responsable de ejecutar el benchmark.

Mientras los benchmark basados en especificación y en kit han sido exitosos en el pasado, las tendencias actuales han favorecido el desarrollo del modelo de desarrollo basado en kit.

2.2.4 Criterios de calidad

Los diseñadores de las cargas de trabajo deben balancear una serie de criterios, normalmente en conflicto entre ellos, para tener éxito. Para ello, se deben tener en cuenta varios factores y se han de realizar sacrificios en las opciones de diseño para influenciar las fortalezas y debilidades de las cargas de trabajo. Dado que ninguna carga de trabajo puede ser fuerte en todas las áreas, existirá siempre una necesidad de crear múltiples cargas de trabajo y benchmarks.

Es importante entender las características de una carga de trabajo y determinar si es aplicable a una situación particular. Durante el desarrollo de una nueva carga de trabajo, las metas deben ser definidas de tal forma en que las opciones entre los criterios de diseño competentes se puedan hacer de acuerdo con esas metas para alcanzar el equilibrio necesario. Varios investigadores y miembros de la industria han extraído varias características deseables de los benchmarks. Estas características se pueden organizar en los siguientes grupos:

- **Relevancia**: determina cuán cerca se encuentra el comportamiento del benchmark con los comportamientos de interés para los interesados en los resultados.
- **Reproducibilidad**: determina la habilidad de reproducir, de forma consistente, resultados similares durante distintas ejecuciones del benchmark con los mismos parámetros de configuración.
- **Imparcialidad**: determina la capacidad de las diferentes configuraciones de prueba para competir por méritos propios sin limitaciones artificiales.
- **Verificabilidad**: provee confianza de que los resultados de un benchmark son precisos.
- **Usabilidad**: pretende evitar barreras que impidan a los usuarios ejecutar benchmarks en sus entornos de prueba.

Todos los benchmarks están sujetos a este tipo de criterios, pero cada categoría incluye problemas adicionales que son específicos a cada benchmark, dependiendo de los objetivos del benchmark.

Capítulo 3

Estado del arte: PolyBench/C

PolyBench es un proyecto software que recoge una serie de benchmarks específicos utilizados para el desarrollo y verificación de técnicas de optimización poliédrica.

Este capítulo describe la estructura e implementación de PolyBench/C, que es el conjunto de pruebas desarrollado en el lenguaje de programación C, con el fin de entender las decisiones de diseño tomadas por sus autores y analizar el funcionamiento del software. Este estudio supone un paso esencial en el proceso de adaptación al lenguaje Python.

Durante el desarrollo de este capítulo se realizan múltiples referencias a los ficheros de PolyBench/C y por ello se recomienda disponer de dichos ficheros para su consulta. No obstante los detalles más relevantes se recogen en el contenido de este trabajo, por lo que no es obligatorio disponer de dichos ficheros. En el anexo A.1 se indican las instrucciones a seguir para obtener el código fuente empleado durante el desarrollo de este trabajo.

La versión utilizada de PolyBench/C para el desarrollo de este trabajo es la *4.2.1-beta*, que es la versión pública preempaquetada más actual a la fecha de realización de este trabajo. Es posible obtener una versión más reciente a través de los repositorios de software, no obstante, los cambios existentes no suponen ninguna diferencia para el desarrollo de este trabajo.

3.1 Análisis de ficheros y directorios

El primer paso para el estudio de PolyBench/C consiste en realizar un análisis de la estructura del proyecto para poder localizar los puntos de interés en dónde se encuentra el código más relevante para poder proceder después a su estudio más específico. Para ello es necesario comenzar observando la estructura del proyecto desde el punto de vista del sistema de ficheros para posteriormente poder realizar un análisis ordenado sobre el contenido de los distintos ficheros.

La tabla 3.1 muestra el listado de ficheros y directorios presentes en la raíz del proyecto PolyBench/C. En un primer vistazo se puede apreciar una estructura de proyecto típica en

Tabla 3.1: Contenido del directorio **polybench-c-4.2.1-beta**

Fichero	Descripción
AUTHORS	Listado de autores del proyecto
CHANGELOG	Resumen de cambios entre versiones
datamining	Directorio de categoría
LICENSE.txt	Licencia de Ohio State University
linear-algebra	Directorio de categoría
medley	Directorio de categoría
polybench.pdf (*)	Documentación sobre los algoritmos
README	información que se debe leer
stencils	Directorio de categoría
THANKS	Agradecimientos
utilities	Directorio de herramientas

(*) Fichero no disponible si se descarga desde Subversion

dónde existen una serie de ficheros en la raíz que son de carácter informativo y una serie de directorios adicionales en dónde se deberían encontrar los distintos componentes software.

3.1.1 Ficheros y directorios en el directorio *raíz*

El primer paso en este estudio consiste en analizar el contenido de los ficheros informativos de la raíz para comprender la estructura de directorios y el contenido de los mismos.

A simple vista parece una buena idea comenzar el análisis por los ficheros *README* y *polybench.pdf* ya que parecen los ficheros candidatos a contener la información más relevante sobre el proyecto.

El fichero README

Este fichero contiene un resumen sobre el proyecto, sus funciones y características, herramientas auxiliares e instrucciones de uso, así como las distintas opciones de configuración disponibles para su implementación. También incluye información de contacto y una lista de cambios a modo informativo sobre el progreso del proyecto.

De las instrucciones de uso y herramientas auxiliares se puede deducir que el directorio *utilities* contiene una parte relevante del proyecto y el resto de directorios representan las distintas categorías en las que se pueden clasificar los distintos benchmarks y, en consecuencia, estos directorios contienen las implementaciones de los mismos.

El fichero polybench.pdf

Este fichero contiene una descripción detallada de los algoritmos implementados en PolyBench. Esta descripción no está vinculada a ninguna implementación particular por lo que planea ser un punto de referencia consistente independiente del lenguaje en el que se implemente.

Este fichero incluye también algunos detalles específicos de la implementación en C y un listado de problemas conocidos.

De entre los detalles de implementación cabe mencionar que los bucles de algunos benchmarks están intencionalmente invertidos para hacer que los kernels sean más complejos de analizar. Por otra parte algunos kernels del grupo BLAS utilizan escalares en los bucles, dificultando la paralelización de los mismos. Finalmente también se menciona que la reserva de memoria para los arrays se realiza de la forma más natural posible, haciéndolos compactos y manteniéndolos rectangulares. Estos detalles se tendrán en cuenta durante la adaptación a Python como se detalla en el capítulo 4.

Del listado de problemas se menciona que los resultados del kernel *correlation* son todo 1 debido a una propiedad del algoritmo combinada con el método de inicialización y, del kernel *gramschmidt* menciona que gran parte de la matriz de salida R es 0 e incluso puede contener valores NaN. Esto también se debe a cómo se generan los valores de entrada. Los valores NaN pueden llegar a suponer un problema durante la traducción como también se detalla en el capítulo 4.

El directorio utilities

En este directorio se encuentran las principales herramientas sobre las que se apoya el proyecto, así como ficheros auxiliares que permiten controlar y generar la parte personalizable de los benchmarks.

Entre los ficheros destacados se encuentran *polybench.c* y *polybench.h*. Juntos forman una base común de funciones y definiciones para todos los benchmarks. Estos ficheros son el objeto de estudio principal en los apartados 3.2.2 y 3.2.3.

Acompañando a estos ficheros se encuentra *template-for-new-benchmark.c*, que define una plantilla genérica con código de ejemplo para crear un nuevo benchmark.

De especial importancia son también los ficheros *polybench.spec* y *papi_counters.list*. En el primero se define la lista de los distintos benchmarks junto con sus especificaciones técnicas (tipos de datos, variables de control y tamaño de los conjuntos de datos) y es utilizado por los scripts Perl para generar benchmarks a partir de estos parámetros. El segundo fichero incluye una lista de identificadores de eventos PAPI que se monitorizarán en caso de habilitar el soporte de PAPI durante la compilación. La tecnología PAPI se explica brevemente en el apartado 3.3.2.

Los distintos scripts Perl se mencionan al final del fichero README y sirven para realizar diversas tareas como compilar y ejecutar benchmarks, limpiar ficheros intermedios y generar benchmarks a partir del fichero *polybench.spec*.

Existe también un script auxiliar, escrito en Bash, que sirve para ejecutar los benchmarks y monitorizar su tiempo de ejecución. Este script tiene como objetivo ejecutar el benchmark seleccionado varias veces y computar el tiempo promedio de las ejecuciones. Los valores temporales los obtiene de la salida del programa, siendo un requisito que el benchmark se haya compilado con la opción *POLYBENCH_TIME*.

Finalmente el fichero *polybench.R* contiene la implementación de varios benchmarks en el lenguaje de programación R. La documentación no hace mención de este programa, pero parece que su propósito es implementar los algoritmos para comparar sus resultados con los de PolyBench/C.

3.1.2 Otros ficheros y directorios

La tabla 3.1 contiene una descripción suficiente para el resto de ficheros que aparecen en la raíz y que no almacenan contenido relevante para el desarrollo de este trabajo.

Por otra parte, los directorios *datamining*, *linear-algebra*, *medley* y *stencils* contienen las implementaciones concretas de cada uno de los benchmarks. Éstos se estudian más en detalle en el capítulo 4.

3.2 Análisis de contenido

En este apartado se realiza un análisis más exhaustivo del contenido de los ficheros más relevantes vistos hasta ahora con el fin de crear una visión general del proyecto que permita la extracción de características que simplifiquen el proceso de traducción al lenguaje Python.

3.2.1 Análisis del fichero README

Para comenzar con el análisis se procede leyendo el contenido del fichero *README*. Para no alargar este texto con contenido poco relevante, se mencionarán los elementos más interesantes de este fichero de cara al análisis.

Características de PolyBench

Lo primero que se puede observar son la lista de características de PolyBench:

- Existe un único fichero, configurable en tiempo de compilación, empleado para la instrumentación del kernel. Aunque no se menciona explícitamente, este fichero es *utilities/polybench.h*

- Implementa operaciones adicionales como liberar la memoria caché antes de la ejecución del kernel, o establecer la prioridad del proceso a tiempo real para prevenir la interferencia del sistema operativo y de otros procesos.
- Inicialización de datos no nula y volcado de datos de salida.
- Construcciones sintácticas para prevenir la eliminación de código muerto en el kernel por parte del compilador.
- Límites paramétricos en los kernels para ofrecer una implementación de propósito general.
- Marcado del kernel en el código mediante directivas *#pragma*.

Algunos puntos de esta lista deben ser considerados con especial atención durante la traducción a Python. Por ejemplo, cómo cambiar la prioridad de un proceso es una operación dependiente del sistema operativo. Si el entorno no ofrece de medios para acceder a estas funciones será necesario implementarlas de algún modo, quizás a través de librerías de código C auxiliares. Otro problema visible a simple vista resulta cómo marcar el código del kernel, ya que Python no provee por sí mismo de mecanismos que actúen como pragma. Estos problemas y su resolución se tratan con más detalle en el capítulo 4.

Compilación y opciones de configuración

El siguiente punto que explorar en el fichero README es la sección de ejemplos de compilación. Esta sección resulta de especial importancia ya que en ella se muestra cómo compilar y ejecutar las pruebas, y también se confirma que los ficheros *polybench.c* y *polybench.h* son una dependencia de compilación para todas las pruebas.

En esta sección del README también se detalla cómo aplicar las diferentes opciones de compilación que permiten alterar el comportamiento de PolyBench. A continuación se enumeran las opciones de compilación típicas.

- **POLYBENCH_TIME**: muestra el tiempo de ejecución mediante una llamada al sistema *gettimeofday*. Por defecto se encuentra desactivado.
- **MINI_DATASET**, **SMALL_DATASET**, **MEDIUM_DATASET**, **LARGE_DATASET**, **EXTRALARGE_DATASET**: permiten controlar el tamaño del conjunto de datos que se emplea durante la ejecución del benchmark. Por defecto *MEDIUM_DATASET*.
- **POLYBENCH_DUMP_ARRAYS**: imprime en la salida estándar de error los datos de salida de los arrays. Por defecto se encuentra desactivado.
- **POLYBENCH_STACK_ARRAYS**: emplea reserva de memoria en la pila en lugar de usar *malloc()* para los arrays. Por defecto se encuentra desactivado.

Cabe mencionar que, de la anterior lista, no se puede implementar en Python la opción *POLYBENCH_STACK_ARRAYS* mediante el uso de listas debido a que estas están gestionadas internamente por el intérprete y no hay forma de alterar este comportamiento sin modificar

el intérprete. Además dados dos intérpretes cualesquiera, no hay garantías de que la implementación interna de las listas entre ellos sea la misma, incluso entre distintas versiones de un mismo intérprete.

Opciones de rendimiento

- **POLYBENCH_USE_RESTRICT**: añade la palabra clave *restrict* en las declaraciones de arrays para indicar al compilador que no existe aliasing sobre los mismos. Por defecto se encuentra desactivado y en general no se usa debido a que sólo está definido en el estándar C99 (en C++ no está definido) y no todos los compiladores lo manejan correctamente.
- **POLYBENCH_USE_SCALAR_LB**: indica al compilador que utilice valores constantes en las variables de control de bucle en lugar de variables paramétricas en los SCoPs. Por defecto se encuentra desactivado.
- **POLYBENCH_PADDING_FACTOR**: añade relleno en todas las dimensiones de todos los arrays introduciendo tantos elementos adicionales como indique este parámetro. Por defecto, el valor es 0.
- **POLYBENCH_INTER_ARRAY_PADDING_FACTOR**: traslada la dirección de comienzo de los arrays PolyBench almacenados en el montículo (por defecto) por un múltiplo de el valor que se especifique en este parámetro. Por defecto 0.
- **POLYBENCH_USE_C99_PROTO**: usa prototipos de funciones de C99 estándar. Por defecto desactivado.

De esta lista, las opciones *POLYBENCH_USE_RESTRICT* y *POLYBENCH_USE_C99_PROTO* no se aplican en Python. La primera de ellas no se puede replicar sin realizar modificaciones internas al intérprete mientras que la segunda es específica del estándar C99 y está relacionada con *POLYBENCH_USE_SCALAR_LB*. Lo que hace básicamente es declarar arrays con valores escalares o paramétricos en función del valor de esta última opción.

Del mismo modo que las anteriores, *POLYBENCH_USE_SCALAR_LB* no se aplica a Python pues en este lenguaje no existen los valores constantes. En Python todas las declaraciones son referencias a objetos. En el capítulo 4.1 se detallan este y otros aspectos que diferencian a C y Python. En ese mismo capítulo se detalla también por qué las opciones *POLYBENCH_PADDING_FACTOR* y *POLYBENCH_INTER_ARRAY_PADDING_FACTOR* pueden suponer un problema.

Opciones de temporización y perfilado

- **POLYBENCH_PAPI**: habilita el soporte para eventos PAPI. Por defecto desactivado.
- **POLYBENCH_CACHE_SIZE_KB**: permite seleccionar la cantidad, en Kilo Bytes, de memoria caché para liberar. Por defecto 33MiB.

- **POLYBENCH_NO_FLUSH_CACHE**: indica a PolyBench que no realice la liberación de memoria caché antes de llamar al temporizador. Por defecto desactivado (se libera la caché).
- **POLYBENCH_CYCLE_ACCURATE_TIMER**: habilita el uso del contador TSC para monitorizar el tiempo de ejecución del kernel. Por defecto desactivado.
- **POLYBENCH_LINUX_FIFO_SCHEDULER**: habilita el uso del planificador FIFO en tiempo real para la ejecución del kernel. El programa debe ejecutarse con privilegios elevados y funciona sólo en Linux. Adicionalmente el programa debe ser compilado con "-lc". Por defecto está desactivado.

Las opciones aquí definidas no parecen suponer, en general, un problema para la traducción a Python. No obstante, en el capítulo 4 se comentan los detalles de implementación de estas opciones y se puede ver que algunas de ellas, como el acceso a los eventos PAPI o la reconfiguración del planificador no resultan ser tan triviales como aparentan.

Con respecto a las opciones de temporización, se debe tener en cuenta que algunas pruebas deben ejecutarse múltiples veces, como viene destacado en la documentación, debido al poco tiempo de ejecución de las mismas. PolyBench/C se vale de un script externo que se encarga de realizar múltiples ejecuciones y calcular el tiempo promedio de ejecución. Todo esto se implementa en Python a través de una herramienta que se describe en el anexo A.4.2.

3.2.2 Análisis del fichero polybench.h

El fichero *polybench.h* es la cabecera en la que se definen símbolos para ser exportados (como constantes y prototipos de funciones) y utilizados en los ficheros de implementación ".c".

Procediendo con el análisis, un vistazo rápido muestra a grandes rasgos que se usan de forma intensiva las directivas de preprocesador de C (para más información sobre directivas de preprocesador consultar la sección 6.10.3 del borrador de referencia de C99 [5]). Concretamente se usa de forma masiva la directiva *#define identificador lista_de_reemplazo* por una parte para implementar los parámetros de configuración vistos en el fichero README y por otra parte para implementar azúcar sintáctico de operaciones que serían tediosas de realizar manualmente.

Un ejemplo sencillo para mostrar cómo funcionan las transformaciones se puede ver con el parámetro de configuración *POLYBENCH_USE_C99_PROTO*:

```
1 /* C99 arrays in function prototype. By default, do not use. */
2 # ifdef POLYBENCH_USE_C99_PROTO
3 #  define POLYBENCH_C99_SELECT(x,y) y
4 # else
5 /* default: */
6 #  define POLYBENCH_C99_SELECT(x,y) x
```

```
# endif
```

En este fragmento de directivas se puede observar el siguiente comportamiento: en caso de estar definido *POLYBENCH_USE_C99_PROTO*, el reemplazo "*POLYBENCH_C99_SELECT(x,y) y*" es seleccionado. En este caso, cuando aparezca en algún lugar del código o en alguna lista de reemplazo el texto *POLYBENCH_C99_SELECT(x,y)* se reemplazará el contenido de dicho texto por *y*. Del mismo modo si no está definido *POLYBENCH_USE_C99_PROTO*, se define el reemplazo "*POLYBENCH_C99_SELECT(x,y) x*", en este caso reemplazando el contenido por lo que represente *x*.

Un ejemplo de uso se puede encontrar más abajo en el mismo fichero, en la línea 97 en dónde se definen las macros para el uso de arrays en los prototipos de las funciones:

```
# define POLYBENCH_1D(var, dim1,ddim1) var[POLYBENCH_RESTRICT
    POLYBENCH_C99_SELECT(dim1,ddim1) + POLYBENCH_PADDING_FACTOR]
```

Si se aplicase sólamente la sustitución "*POLYBENCH_C99_SELECT(x,y) y*", la definición quedaría de la siguiente forma:

```
# define POLYBENCH_1D(var, dim1,ddim1) var[POLYBENCH_RESTRICT ddim1
    + POLYBENCH_PADDING_FACTOR]
```

Aplicando el resto de sustituciones, en dónde *POLYBENCH_RESTRICT* se sustituye por "nada" (usando la definición *#define POLYBENCH_RESTRICT* sin lista de reemplazos) y *POLYBENCH_PADDING_FACTOR* se sustituye por *0* la definición de *POLYBENCH_1D* quedaría:

```
# define POLYBENCH_1D(var, dim1,ddim1) var[ ddim1 + 0]
```

Un ejemplo de cómo se aplican las sustituciónes en el código fuente se puede encontrar en el fichero *datamining/correlation/correlation.c*. Por ejemplo, en *kernel_correlation()*:

```
/* Main computational kernel. The whole function will be timed,
    including the call and return. */
static
void kernel_correlation(int m, int n,
    DATA_TYPE float_n,
    DATA_TYPE POLYBENCH_2D(data,N,M,n,m),
    DATA_TYPE POLYBENCH_2D(corr,M,M,m,m),
    DATA_TYPE POLYBENCH_1D(mean,M,m),
    DATA_TYPE POLYBENCH_1D(stddev,M,m))
{
    ...
}
```

Asumiendo que *DATA_TYPE* se reemplaza por *double* (un tipo de dato en C), y aplicando las mismas transformaciones para *POLYBENCH_2D* que en *POLYBENCH_1D*, el ejemplo ilustrado se transforma en lo siguiente:

```
/* Main computational kernel. The whole function will be timed,
   including the call and return. */
static
void kernel_correlation(int m, int n,
    double float_n,
    double data[N][M],  // Array de dos dimensiones transformado
    double corr[M][M],  // Array de dos dimensiones transformado
    double mean[M],     // Array de una dimension transformado
    double stddev[M])   // Array de una dimension transformado
{
    ...
}
```

Las reglas de sustitución se aplican del mismo modo para todas las macros del tipo *#define*, por lo que no se describirán todas y cada una de las transformaciones. No obstante sí se describirán, a modo general, los distintos tipos de transformaciones que se realizan en PolyBench.

Como se mencionaba anteriormente en este capítulo se emplean dos tipos de macros: las que definen parámetros de configuración y las que implementan azúcar sintáctico.

El ejemplo visto anteriormente con las definiciones de arrays *POLYBENCH_1D* se trata de azúcar sintáctico. Es fácil entender por qué se define así y no de forma manual en cada fichero de implementación. Tener que escribir en la implementación todas las posibles variantes de configuración (cambiar el tipo de dato de forma dinámica, modificar palabras clave, seleccionar el tipo de parámetros, etc.) resulta bastante tedioso y fuera de contexto para el objetivo de PolyBench.

Puesto que las sustituciones de configuración ya se han mencionado en el fichero README, se procede explicando las sustituciones de azúcar sintáctico. Estas se pueden categorizar, según van apareciendo en el fichero, de la siguiente forma:

- Declaración de arrays en los prototipos de las funciones
- Declaración de arrays en el cuerpo de las funciones
- Prevención de código muerto
- Impresión por pantalla
- Instrumentación

Los siguientes apartados ilustran estos tipos de macros. En los fragmentos de código se incluyen tanto las definiciones de las macros como un ejemplo de aplicación de las mismas, así como el resultado tras el preprocesado.

Macros para la declaración de arrays en los prototipos de las funciones.

Este tipo de macros es similar al explicado previamente en los ejemplos, ejerciendo de azúcar sintáctico para la declaración de arrays en los prototipos de las funciones y permitiendo seleccionar si el valor de referencia del tamaño del array lo constituye un valor escalar o paramétrico.

```
# define POLYBENCH_1D(var, dim1,ddim1) (...)

// Prototipo de foo() antes del preprocesado
void foo(double POLYBENCH_1D(data, M, m)) { ... }
// Prototipo de foo() después del preprocesado
void foo(double data[M]) { ... }
```

Macros para la declaración de arrays en el cuerpo de una función.

Este tipo de macros dependen, en el momento de analizar las listas de sustitución, de distintos parámetros de configuración como por ejemplo si la memoria para los arrays se debe reservar en la pila o en el montículo.

```
# ifndef POLYBENCH_STACK_ARRAYS
/* arrays en el montículo */
#  define POLYBENCH_1D_ARRAY_DECL(var, type, dim1, ddim1) ( ... )
# else
/* arrays en la pila */
#  define POLYBENCH_1D_ARRAY_DECL(var, type, dim1, ddim1) ( ... )
# endif

void foo() {
    // Declaración del array antes del preprocesado
    POLYBENCH_1D_ARRAY_DECL(data, double, M, m);
}

void foo() {
    // Declaración del array tras el preprocesado (en pila).
    double data[M];
}

void foo() {
    // Declaración del array tras el preprocesado (en montículo).
    double (*data)[M];
    // La siguiente línea se incluye en la sustitución.
    data = (double (*)[M]) polybench_alloc_data(M * sizeof(double));
}
```

Macros que fuerzan a que el compilador no realice eliminación de código muerto.

Para prevenir que el compilador realice DCE los autores de PolyBench/C simplemente añaden una estructura de bifurcación en la que las variables de control son los parámetros de entrada del programa principal. De este modo, la función *print_array()* no puede ser marcada como eliminable por el compilador ya que este no puede determinar si esa rama se va a ejecutar o no.

```
/* Dead-code elimination macros. Use argc/argv for the run-time
    check. */
# ifndef POLYBENCH_DUMP_ARRAYS
#  define POLYBENCH_DCE_ONLY_CODE    if (argc > 42 && !
    strcmp(argv[0], ""))
# else
#  define POLYBENCH_DCE_ONLY_CODE
# endif
...
# define polybench_prevent_dce(func)    \
  POLYBENCH_DCE_ONLY_CODE     \
  func

int main(int argc, char** argv) {
    ...
    // Uso de polybench_prevent_dce(func)
    polybench_prevent_dce(print_array(m, data));
    ...
}

int main(int argc, char** argv) {
    ...
    // Macro expandida sin eliminación de código
    print_array(m, data);
    ...
}

int main(int argc, char** argv) {
    ...
    // Macro expandida con eliminación de código
    if (argc > 42 && ! strcmp(argv[0], ""))
        print_array(m, data);
    ...
}
```

Macros para imprimir arrays por pantalla.

Este tipo de macros simplemente definen una serie de plantillas textuales en las que figura la función de impresión por pantalla de C con una serie de parámetros preestablecidos, por lo que no resulta interesante mostrarlas.

Macros que controlan la instrumentación.

Son una serie de macros que, dependiendo de las macros de configuración, pueden vincular los símbolos que definen con prototipos de funciones o pueden implementar dichas funciones a través de las listas de sustitución (en el caso de PAPI). Debido a que son bastantes macros y sencillas de entender, no se muestran en este trabajo. Se encuentran a partir de la línea 175 del fichero *polybench.h*.

Finalmente del fichero *polybench.h* quedan sólo los prototipos de funciones, que no serán mencionados en este apartado, sino durante el análisis de *polybench.c*, en donde se encuentran sus respectivas implementaciones.

3.2.3 Análisis del fichero polybench.c

En el fichero *polybench.c* se encuentran las implementaciones de las funciones elementales necesarias para PolyBench, así como las implementaciones de las funciones prototipo definidas en *polybench.h* para poder desarrollar posteriormente las distintas pruebas. Este fichero se puede ver como una librería en la que se implementan los distintos bloques funcionales de un programa de pruebas: manejo de memoria, estructuras de datos, instrumentación y acceso a APIs del sistema.

En un vistazo rápido del fichero, en seguida se llega a la conclusión de que resulta más productivo analizar a gran escala qué es lo que realiza cada una de las funciones que se implementan frente a intentar explicar cada línea de código. El análisis detallado de cada función no tiene sentido ya que la mayor parte de las implementaciones son llamadas a APIs del sistema operativo y de otras librerías. Es por esto que resulta más interesante entender los problemas que intentan resolver. Para ello se enumeran las funciones existentes y se describe brevemente cuál es el objetivo de las mismas. Por simplicidad se divide este apartado en categorías lógicas y es posible que el orden de aparición de las funciones mencionadas no se corresponda con el orden de aparición en el código fuente.

Funciones de temporización.

```
1 static double rtclock() { ... }
2 static unsigned long long int rdtsc() { ... }
```

- *rtclock()* obtiene el número de segundos y microsegundos que han pasado desde un instante prefijado en el sistema. Internamente utiliza *gettimeofday()* [6].
- *rdtsc()* lee el valor del registro TSC disponible en la arquitectura x86. Este registro almacena el número de ciclos de la CPU desde el último reset. El valor devuelto por este registro puede no ser estable en sistemas que soporten escalado de frecuencias. Además, en sistemas multiprocesador, si el proceso migra de un procesador a otro, el valor obtenido puede ser diferente ya que cada núcleo se puede inicializar en momentos distintos y puede escalar frecuencias de forma independiente. En principio estos problemas están resueltos en arquitecturas con soporte para TSC invariante (en general, cualquier procesador Intel posterior a 2008 (aparición de los *Core i*) y en caso de AMD, a partir de la familia K10).
  Para obtener información más detallada se puede consultar el capítulo 17.17 del manual del desarrollador de Intel [7].

Funciones de gestión de memoria para arrays creados en el montículo.

```
1 static void* xmalloc(size_t alloc_sz) { ... }
2 void* polybench_alloc_data(unsigned long long int n, int elt_size)
    { ... }
3 void polybench_free_data(void* ptr) { ... }
```

- *xmalloc()* es una función interna que se encarga de reservar la memoria suficiente para nuevos arrays. Esta función se encarga también del padding y el alineado de los datos en memoria.
  Internamente utiliza la función *posix_memalign()* [8] para realizar una reserva de memoria alineada a 4096 bytes. La cantidad de memoria reservada incluye padding y, en caso de que se haya configurado padding al principio del array, se crea una tabla en dónde se almacena el puntero original a esta memoria y se devuelve un puntero desplazado a la dirección correspondiente al punto en dónde deben empezar los datos del array.
- *polybench_alloc_data()* es la función que usan los benchmarks de forma indirecta (a través de macros) para crear arrays en el montículo. Esta función llama a *xmalloc()* para la reserva de memoria.
- *polybench_free_data()* es la función que usan los benchmarks de forma indirecta (a través de macros) para liberar la memoria de arrays creados en el montículo.

Funciones de gestión de memoria para arrays con inter-padding.

```
1 static void grow_alloc_table() { ... }
```

```
2 static void* register_padded_pointer(void* ptr, size_t orig_sz,
      size_t padded_sz) { ... }
3 static void free_data_from_alloc_table (void* ptr) { ... }
4 static void check_alloc_table_state() { ... }
```

Este conjunto de funciones se emplea cuando los arrays tienen relleno. Lo que realizan internamente es gestionar una tabla en la que se almacena un puntero al array original (con padding) y otro puntero que apunta al primer elemento del array de datos (vista de usuario). Este último puntero es con el que trabaja el programa de benchmark.

Funciones de instrumentación.

```
1 void polybench_prepare_instruments() { ... }
2 void polybench_timer_start() { ... }
3 void polybench_timer_stop() { ... }
4 void polybench_timer_print() { ... }
```

- *polybench_prepare_instruments()* realiza un borrado de la caché de datos y reconfigura el planificador del sistema operativo. Estas operaciones sólo las realiza si las respectivas opciones de configuración están activas.
- *polybench_timer_start()* llama a *polybench_prepare_instruments()* y almacena el instante de tiempo actual a partir de una de las funciones de temporización citadas anteriormente.
- *polybench_timer_stop()* almacena el instante de tiempo actual a partir de las funciones de temporización y restaura el planificador del sistema operativo en caso de ser necesario.
- *polybench_timer_print()* imprime por pantalla los resultados de las mediciones de tiempo.

Funciones de instrumentación mediante PAPI.

```
1 static void test_fail(char *file, int line, char *call, int retval)
      { ... }
2 void polybench_papi_init() { ... }
3 void polybench_papi_close() { ... }
4 int polybench_papi_start_counter(int evid) { ... }
5 void polybench_papi_stop_counter(int evid) { ... }
6 void polybench_papi_print() { ... }
```

Este conjunto de funciones se encargan de gestionar el uso de los eventos PAPI que permiten conocer valores específicos de eventos hardware. La lista de eventos que se van a monitorizar se llama *_polybench_papi_eventlist* y se encuentra al principio del fichero, en la línea 51. En

esta lista se incluye, literalmente mediante la directiva *#include* de C, el contenido del fichero *papi_counters.list*.

Funciones del sistema.

```
void polybench_flush_cache() { ... }
void polybench_linux_fifo_scheduler() { ... }
void polybench_linux_standard_scheduler() { ... }
```

- *polybench_flush_cache()* intenta descartar la caché del procesador realizando una operación lineal sobre un array grande (del orden de millones de bytes).
- *polybench_linux_fifo_scheduler()* cambia el planificador de tiempo de CPU para el proceso actual por una implementación basada en FIFO en sistemas Linux con la prioridad máxima disponible. Requiere elevación de privilegios.
- *polybench_linux_standard_scheduler()* restaura el planificador de tiempo de CPU para el proceso actual. Requiere elevación de privilegios.

3.2.4 Análisis del fichero template-for-new-benchmark.c

Hasta ahora se han visto los distintos componentes que conforman PolyBench. En esta sección se analiza el cómo aplicar dichos componentes en una prueba.

El directorio *utilities* contiene un fichero de implementación de pruebas a modo de plantilla que será objeto de estudio. En este fichero se puede comprobar cómo el uso de macros ha facilitado mucho el desarrollo de una prueba que soporte todos los tipos de configuración disponibles sin hacer que el código fuente se vuelva complicado de entender.

A continuación se detallan las funciones contenidas en el fichero plantilla *template-for-new-benchmark.c*.

La primera función que conviene analizar es la función *main()*, ya que representa el cuerpo del programa principal en C.

```
int main(int argc, char** argv)
{
    /* Retrieve problem size. */
    int n = N;

    /* Variable declaration/allocation. */
    POLYBENCH_2D_ARRAY_DECL(C,DATA_TYPE,N,N,n,n);

    /* Initialize array(s). */
    init_array (n, POLYBENCH_ARRAY(C));

    /* Start timer. */
```

```
13      polybench_start_instruments;
14
15      /* Run kernel. */
16      kernel_template (n, POLYBENCH_ARRAY(C));
17
18      /* Stop and print timer. */
19      polybench_stop_instruments;
20      polybench_print_instruments;
21
22      /* Prevent dead-code elimination. All live-out data must be
        printed
23          by the function call in argument. */
24      polybench_prevent_dce(print_array(n,  POLYBENCH_ARRAY(C)));
25
26      /* Be clean. */
27      POLYBENCH_FREE_ARRAY(C);
28
29      return 0;
30  }
```

En esta función se puede ver perfectamente cómo se emplean los conceptos vistos a lo largo de este capítulo, las macros de inicialización de arrays, tipo de datos y paso por parámetros de arrays, así como el uso de la instrumentación de monitorización antes y después de la llamada a la función *kernel_template()*, la macro de prevención de eliminación de código muerto y liberación de memoria de arrays.

En esta función se introduce el uso de las funciones *init_array()*, *kernel_template()* y *print_array()*. La primera función que aparece es *init_array()*:

```
1  static
2  void init_array(int n, DATA_TYPE POLYBENCH_2D(C,N,N,n,n))
3  {
4      int i, j;
5
6      for (i = 0; i < n; i++)
7          for (j = 0; j < n; j++)
8              C[i][j] = 42;
9  }
```

Esta función tiene como objetivo inicializar el array (o los arrays) con los criterios necesarios para el algoritmo que se desee implementar. En esta función se puede observar cómo el array se pasa empleando macros para definir el tipo de dato contenido en el array y el tipo de array, así como sus dimensiones como un escalar (letras “N” mayúsculas) o como un parámetro (letras “n” minúsculas).

La siguiente función que aparece en el cuerpo de *main()* es la función *kernel_template()*,

que implementa el algoritmo en cuestión.

```
static
void kernel_template(int n, DATA_TYPE POLYBENCH_2D(C,N,N,n,n))
{
    int i, j;

#pragma scop
    for (i = 0; i < _PB_N; i++)
        for (j = 0; j < _PB_N; j++)
            C[i][j] += 42;
#pragma endscop

}
```

A simple vista en esta función se puede apreciar que llama la atención el uso de directivas *#pragma*. Estas directivas se usan en PolyBench para delimitar la parte de control estático del algoritmo. Esta información es empleada por herramientas como PoCC [9] para poder determinar dónde se encuentra la parte de control estático y así poder ser compilada mediante el uso de optimizadores poliédricos.

Finalmente queda por explicar la función *print_array()*.

```
static
void print_array(int n, DATA_TYPE POLYBENCH_2D(C,N,N,n,n))
{
    int i, j;

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++) {
            fprintf (stderr, DATA_PRINTF_MODIFIER, C[i][j]);
            if (i % 20 == 0) fprintf (stderr, "\n");
        }
    fprintf (stderr, "\n");
}
```

Esta función imprime por pantalla los datos del array (o arrays) que se pasa como parámetro. Un objetivo no tan evidente de esta función consiste en prevenir al compilador la eliminación de código muerto sobre el array de datos al forzar el recorrido sobre el mismo. En esta función también se puede apreciar el uso de macros para pasar el array como parámetro de la función y como modificador para imprimirlo por pantalla.

Teniendo en cuenta el conjunto general de este fichero plantilla, realizar cambios para crear nuevas pruebas sería tan sencillo como modificar en *main()* la declaración, iniciación y liberación de arrays (añadiendo nuevos elementos, cambiando parámetros, etc.), y reescribir

las funciones *init_array()*, *print_array()* y *kernel_*()* para que admitan el número y tipo de parámetros apropiados para el algoritmo que se desea implementar, así como adaptar el cuerpo de las funciones a las necesidades del propio algoritmo.

3.3 Conceptos relacionados

3.3.1 Python

El objetivo de este trabajo consiste en implementar los conceptos vistos a lo largo de este capítulo en Python.

Python es un lenguaje de programación interpretado de alto nivel, con tipado dinámico y multiparadigma, dando soporte para la programación orientada a objetos, programación imperativa y, hasta cierto punto, programación funcional.

Los intérpretes de Python se encuentran disponibles en múltiples plataformas. Este trabajo se centra en los intérpretes de Python CPython [10] y PyPy [11]. La versión del lenguaje empleada para el desarrollo de este trabajo es Python 3.6, ya que es la versión máxima soportada por PyPy.

3.3.2 PAPI

PAPI [12], del inglés Performance Application Programming Interface, es una libreria que facilita el uso de los contadores de rendimiento integrados en la unidad de procesamiento central. Esta interfaz se encuentra disponible en las CPUs de los principales fabricantes del momento y permite a los ingenieros de software visualizar, en casi tiempo real, la relación entre el rendimiento del software y los eventos del procesador. Algunos de estos eventos incluyen el número de ciclos de ejecución de un programa, los fallos de caché, el número de operaciones de punto flotante, etc.

Actualmente la librería PAPI se encuentra disponible únicamente en sistemas Unix que permitan el acceso a los monitores del sistema. En el caso de Linux, que es el sistema utilizado para el desarrollo de este trabajo, PAPI utiliza el subsubsistema del núcleo *perf*.

3.4 Conclusiones

Del análisis visto hasta ahora se pueden extraer algunas ideas generales que permitirán realizar la implementación de gran parte del código de PolyBench en Python.

Se ha visto que PolyBench/C utiliza herramientas externas para realizar una estimación de los tiempos de ejecución de los benchmarks. También se ha mencionado en los scripts Perl que estos permiten realizar la ejecución de todos los benchmarks con una simple llamada. Te-

niendo esto en cuenta, parece importante disponer de herramientas que realicen una función equivalente en PolyBench/Python, por lo que es interesante diseñar una herramienta externa que permita realizar este tipo de operaciones.

También se ha visto que los benchmarks pueden implementarse con distintas opciones de configuración. La forma más sencilla de implementar esto en Python es a través de parámetros de línea de comandos. En este caso, en lugar de especificar las opciones al compilador se pueden pasar a la herramienta externa. Por otra parte, PolyBench/C tiene en cuenta el contenido del fichero *papi_counters.list* y los scripts Perl, en tiempo de compilación, se encargan del fichero *polybench.spec*. Se debe tener en cuenta que las opciones de configuración se pueden implementar a través de referencias en tiempo de ejecución sin ningún tipo de problema siempre y cuando tengan sentido, sin embargo, directivas como *POLYBENCH_USE_C99_PROTO* no tendrían ningún sentido en Python al carecer de tipos de valores constantes.

De cara al desarrollo de PolyBench en Python, la idea principal es crear una clase abstracta en dónde se implemente la funcionalidad vista en los ficheros *polybench.c* y *polybench.h*. De cara al desarrollo de PolyBench en Python, la idea principal consiste en aglutinar la funcionalidad vista en los ficheros *polybench.c*, *polybench.h* y *template-for-new-benchmark.c* en una clase abstracta, de tal modo en que la clase abstracta implemente la funcionalidad común (gestión de memoria, impresión de datos por pantalla, manejo de contadores de rendimiento, etc.), delegando en las subclases la implementación de los propios algoritmos. No obstante, existen una serie de problemas a tener en cuenta en este proceso. Entre los problemas a resolver se encuentran los provenientes de las diferencias entre C y Python, que pueden variar entre el tratamiento de los tipos de datos y el manejo de memoria, e incluso la implementación de funciones matemáticas como el manejo de la librería matemática. También hay que tener en cuenta que es muy probable encontrarse con diferencias entre las distintas implementaciones de Python a través de los intérpretes CPython y PyPy. De entrada, el manejo de librerías C a través de FFI es diferente y puede acarrear problemas a la hora de acceder a los eventos PAPI y las funciones del sistema. También se deben tener en cuenta las posibles diferencias que puedan aparecer durante la etapa de validación de los benchmarks. Como mínimo se espera que existan problemas relacionados con el redondeo de datos de punto flotante que deben resolverse de algún modo, bien intentando "forzar" a que este redondeo se realice de otra forma o bien aplicando técnicas de validación específicas en dónde puedan surgir estos problemas.

Capítulo 4

Diseño y desarrollo

Hasta este momento se ha analizado el código de PolyBench/C para entender su estructura, implementación y funcionamiento. También se han identificado una serie de posibles problemas que pueden surgir durante la traducción a Python y que se verán en detalle durante el transcurso de este capítulo.

El objetivo de este capítulo es aplicar el conocimiento obtenido sobre PolyBench/C para crear en Python un conjunto de benchmarks equivalente intentando mantener, en la medida de lo posible, los mismos mecanismos de medición (contadores PAPI, medición de tiempos) y las distintas opciones de configuración aprovechando las virtudes ofrecidas por un lenguaje de más alto nivel que C como lo es Python.

Para alcanzar este objetivo es imprescindible conocer las diferencias más importantes entre los lenguajes C y Python prestando especial atención a las características empleadas en PolyBench/C. Este breve estudio forma parte de la sección 4.1. Una vez conocidas estas diferencias y agregando el conocimiento del capítulo 3 se puede comenzar a diseñar PolyBench/Python para a continuación poder implementarlo, como se detalla en las secciones 4.2 y 4.3 respectivamente.

4.1 Los lenguajes C y Python

El proceso de traducción de software requiere un conocimiento específico del funcionamiento de los lenguajes de origen y objetivo. En este caso, el trabajo consiste en traducir código en lenguaje C al lenguaje Python. Estos lenguajes, pese a ser considerados de alto nivel y propósito general, apenas comparten características que permitan realizar una conversión del código de forma directa.

En la tabla 4.1 se enuncian algunas de las diferencias más notables entre los lenguajes C y Python.

Debido a que son lenguajes, en un principio, muy diferentes entre sí, la mejor forma de

Tabla 4.1: Comparativa: C vs Python

C	Python
Propósito general (orientado a sistemas)	Propósito general
Imperativo	Orientado a objetos
Aritmética de punteros	Sin aritmética de punteros
Compilado	Interpretado
Funciones incluidas limitadas	Gran librería de funciones
Tipado estático	Tipado dinámico
Complicado de aprender	Fácil de aprender

ilustrar diferencias en este caso consiste en seleccionar fragmentos de código de PolyBench/C para analizar qué características del lenguaje se utilizan concretamente e ilustrar cuál o cuáles serían las aproximaciones posibles para traducirlo a Python, aprovechando este proceso para comentar también cómo funciona Python en un contexto similar.

Para llevar a cabo esta comparativa, se emplearán fragmentos de código de los ficheros vistos a lo largo del capítulo 3. Ya se ha visto en ese mismo capítulo el uso de macros con distintos fines, uso de valores constantes y azúcar sintáctico. En Python no existen las macros ni nada que se le asemeje. Lo único que se puede hacer al respecto es crear referencias a valores para las constantes e implementar los elementos definidos mediante azúcar sintáctico.

4.1.1 Diferencias de implementación

En este apartado se comentan diferencias generales entre C y Python que afectan directamente a la traducción. Este apartado no pretende ser un manual exhaustivo en el que se detallan las implementaciones de cada lenguaje ni sus diferencias, sino que se pretende explicar de una forma sencilla las diferencias más relevantes a tener en cuenta para comenzar con el proceso de traducción.

Tipos de datos

PolyBench/C utiliza varios tipos de datos de forma extensiva. Por una parte, emplea tipos numéricos como *float* y *double* para valores en punto flotante de 32 y 64 bits y los distintos tipos enteros *short*, *int*, *long int* y *long long int* para valores de al menos 16, 16, 32 y 64 bits respectivamente. Sus versiones *unsigned* también se emplean para realizar operaciones sin signo. Como se puede entender, el programador es responsable de escoger el tipo de dato apropiado al problema. Además, el hecho de que el estándar de C defina que los tipos de enteros sean de "al menos X bits" puede reducir la portabilidad. Por ejemplo, el tipo *int* está definido como de al menos 16 bits. Hoy en día lo habitual es que este tipo sea de 32 bits en máquinas de 32 y 64 bits, pero esta decisión depende de los desarrolladores de los compiladores

y puede hacer que el código no se comporte como el programador desea si se cambia de compilador.

En Python no existe este tipo de problema. Los tipos numéricos están generalmente bien definidos y se encuentran categorizados en tres clases: *int* para enteros, *float* para punto flotante y *complex* para números complejos [13]. Los números enteros tienen precisión ilimitada mientras que los números en punto flotante dependen directamente de la implementación del intérprete. Normalmente CPython los implementa internamente con el tipo C *double* pero no da garantías de que esto sea siempre cierto. Este detalle debe tenerse en cuenta ya que PolyBench/C emplea números de punto flotante con distinta precisión según el algoritmo lo requiera y puede causar problemas con los resultados. Adicionalmente, Python cuenta con el módulo *decimal* [14] que puede ser interesante cuando se desea obtener mayor precisión en las operaciones de punto flotante.

Es importante mencionar también las diferencias entre intérpretes para los números flotantes. Por ejemplo, PyPy trata de forma diferente a CPython el operador de identidad "is" para el tipo float, dando como resultado que la operación "float('nan') is float('nan')" devuelva True en PyPy y False en CPython. Para más información, ver la sección "Object identity of primitive values" en la lista de diferencias de PyPy frente a Python [15].

Aprovechando el tema de los números flotantes, también existe un posible problema en la interpretación de los valores especiales Inf y NaN. Tanto las versiones actuales de GCC como de CPython y PyPy utilizan la misma nomenclatura para estos valores. No obstante esto no está garantizado que vaya a ser así siempre y es posible que una comparación entre los resultados de PolyBench/C y PolyBench/Python falle en algún momento por culpa de los mismos.

Regresando con PolyBench/C, el otro tipo de dato más empleado por los algoritmos es el *array* de tipos primitivos. Los arrays en C son simplemente direcciones de memoria consecutivas en dónde se almacenan los datos. Por ejemplo, un array de 4 enteros de 32 bits se puede declarar como "int array[4]". En este caso, se reserva una región contigua de memoria de 16 bytes (1 entero de 32 bits son (32 / 8) 4 bytes). Sin embargo, Python no define el tipo array. En su lugar, define un tipo de dato más específico que es el tipo *list* o lista en castellano. Esto tiene como ventaja que, al ser un tipo de dato abstracto, implementa una serie de funciones convenientes como añadir y eliminar elementos, ordenarlos en función a un criterio, etc. Otra ventaja que tiene es que, a diferencia de los arrays que requieren que todos los elementos sean del mismo tipo, las listas permiten elementos de distintos tipos. Estas características no resultan útiles en PolyBench, por lo que no se entrará en más detalles sobre las mismas. No obstante, sí es importante destacar cómo se implementan las listas internamente en Python, puesto que a simple vista, parecen el único sustituto disponible para los arrays.

Las listas se implementan, tanto en CPython como en PyPy, mediante estructuras com-

plejas que tienen como medio de soporte de datos un array de C. En CPython, las listas se implementan como un array contiguo de referencias a objetos [16] mientras que PyPy emplea una estrategia diferente basada en el patrón de diseño de estrategia [17]. Con respecto a la eficiencia de la implementación, PyPy tiene ventaja ya que, al usar el patrón estrategia, puede realizar una optimización basada en el tipo de los datos y, en el caso en que la lista contenga todos los elementos de un mismo tipo primitivo numérico (actualmente suportado para *int* y *float*) el rendimiento de las mismas se aproxima mucho al rendimiento de los arrays en C. En cualquier caso, el acceso a elementos de la lista mediante índices tiene un coste casi nulo en ambas implementaciones.

Un detalle importante sobre la implementación de las listas en ambos intérpretes es que realizan una optimización muy común de forma transparente. Esta optimización consiste en reservar un poco más memoria de la necesaria con el fin de mejorar el rendimiento en las operaciones tipo *append()*. De este modo, una lista de 10 elementos puede tener, internamente, memoria reservada para 13 elementos. En caso de llenarse hasta 13 elementos y realizar un nuevo append, la lista se redimensionaría de nuevo con una capacidad, por ejemplo, de 18, pudiendo en este caso insertar el elemento 14 y "desperdiciando" 4 huecos.

Estructuras de control

Las estructuras de control empleadas en PolyBench/C son la bifurcación y el bucle (*if* y *for*). Las diferencias de la estructura de bifurcación entre C y Python, en lo que se refiere a este trabajo, son meramente sintácticas. Por el contrario, el bucle *for* es algo diferente entre ellos. Un bucle *for* en C tiene una estructura dividida en tres secciones (todas ellas opcionales):

- en la primera sección se puede definir una secuencia de instrucciones, separadas por comas, que se ejecutarán una única vez antes de ejecutar el cuerpo del bucle.
- en la segunda sección se define la condición de ejecución que permite controlar la ejecución del bucle mediante expresiones booleanas.
- en la última sección se pueden introducir instrucciones que se ejecutan al finalizar cada iteración del bucle.

A continuación se muestran algunos ejemplos de bucles *for* en C. En ellos se puede ver como las secciones son opcionales en el caso del bucle infinito. Se puede ver también un bucle que realiza 10 iteraciones a partir de una variable de control *i* y, como curiosidad, su versión equivalente como bucle *while*.

```
1 /* Un bucle infinito */
2 for(;;) {}
3
4 /* Un bucle que realiza 10 iteraciones sobre i */
5 int i;
6 for(i = 0; i < 10; ++i) {}
```

```
7
8  /* Un bucle while equivalente al for anterior */
9  int i = 0;
10 while(i < 10) {
11     ++i;
12 }
```

Por otra parte, los bucles *for* en Python no se parecen en nada a los vistos en C. Los bucles *for* en Python están pensados para ser usados únicamente con iteradores. Esto restringe su uso a tipos de objetos iterables como listas, diccionarios y cualquier otra estructura que implemente un iterable. Por una parte esto supone una ventaja a la hora de escribirlos ya que recorrer elementos iterables se realiza de una forma muy natural. Sin embargo, realizar un número determinado de iteraciones se vuelve más complejo al tener que emplear estructuras auxiliares. Esto puede resultar en una ineficiencia tanto de uso de memoria como de tiempo de CPU si el intérprete no es capaz de optimizar este tipo de casos en concreto.

A continuación se muestra un fragmento de código Python en dónde se ilustra la estructura de un bucle *for* y cómo iterar sobre un conjunto iterable.

```
1  # Recorrer todos los elementos de una lista
2  lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3  for item in lst:
4      do_something(item)
5
6  # Recorrer sólo los 3 primeros elementos de la lista
7  for index in range(0, 3):  # el límite alto de range no se incluye
8      item = lst[index]
9      do_something(item)
10
11 # Otra forma de recorrer los 3 primeros elementos de la lista
12 for item in lst[:3]:
13     do_something(item)
```

En la primera forma del bucle, todos los elementos de la lista son explorados. Esta es la forma más eficiente (en tiempo de CPU y uso de memoria) de recorrer un iterable en Python al utilizar directamente el iterador de las listas.

En la segunda forma, se exploran los tres primeros elementos a partir de un rango [18]. Los rangos representan una secuencia inmutable de números comúnmente empleada en bucles, por lo que son muy eficientes en uso de memoria. No obstante esta forma es la más lenta en tiempo de CPU de las tres vistas.

En la tercera forma, al igual que en la segunda, se exploran los tres primeros elementos. No obstante, en esta forma se emplea la notación de *slice* (corte en castellano) que internamente realiza una copia de la lista con los elementos representados en el corte (en este caso el corte

es *lst[0:3]*, intervalo cerrado por la izquierda y abierto por la derecha, obteniendo en la lista resultante los elementos 0, 1 y 2). Una vez obtenida la copia se realiza la iteración sobre los elementos de la misma al igual que se ha visto en la primera forma. Esta forma es la menos eficiente de las tres en uso de memoria ya que requiere crear una copia de la lista original con los elementos especificados en el corte. Sin embargo, una vez obtenida la copia, esta forma es igual de eficiente en tiempo de CPU que la primera.

En el anexo A.2 se muestran los resultados de un experimento que demuestra las diferencias de rendimiento sobre los distintos métodos para recorrer una lista.

Llamadas a funciones

En C existe una única forma de pasar parámetros a las funciones: el paso por valor. En el paso por valor, lo que recibe el cuerpo de la función es una copia del elemento que se pasa. En el caso de que el elemento pasado sea un puntero, se copia dicho puntero. Para cualquier otro tipo de dato, se copian los datos que ocupe en memoria (por ejemplo, en el paso de un *struct*, se copian todos los campos de la estructura; en el paso de un entero, se copia el valor del entero). Un detalle especial en C son los arrays. Cuando se pasa un array a una función, lo que se pasa realmente es un puntero al primer elemento del array, por lo que nuevamente se aplicaría la regla del paso de puntero. Este detalle es la razón por la cual en C siempre que se pasa un array a una función se tiene que pasar también de alguna forma el tamaño de dicho array.

En Python la situación es diferente. En lugar del paso por valor, Python utiliza el paso por asignación. Dado que en Python una asignación crea una nueva referencia (asignación de una etiqueta con un objeto), cuando se llama a una función lo que ocurre es que se genera una nueva referencia al objeto pasado. El efecto que se obtiene es similar al paso por valor siempre y cuando el objeto pasado sea inmutable. Por el contrario, si el objeto pasado es mutable, sería posible modificar directamente el contenido del mismo, como es el caso, por ejemplo, de las listas o los diccionarios. Este comportamiento es deseable ya que, en lo que se refiere a la reimplementación de PolyBench, el comportamiento efectivo es muy similar: los valores numéricos en C son copias al paso por una función y son referencias inmutables en Python al paso por una función. Por otra parte, los arrays en C se pasan como punteros y esto permite modificar su contenido y en Python las listas son objetos mutables al paso por función, por lo que su contenido también se puede modificar.

4.1.2 Intérpretes de Python

En varias ocasiones se han mencionado los intérpretes de Python CPython y PyPy. Estos dos intérpretes son los que se emplean durante el desarrollo de este trabajo. El primero de ellos es el intérprete de referencia desarrollado por la Python Software Foundation. El objetivo de

este intérprete es ofrecer un entorno de ejecución multiplataforma que implementa el lenguaje de programación Python.

PyPy es otro intérprete que realiza otra implementación del lenguaje Python orientada a ofrecer un rendimiento muy superior al de CPython. No obstante, esta ventaja de rendimiento se ve afectada por la falta de soporte para algunas librerías, especialmente las que interactúan a través de FFI y además las sutiles diferencias en la implementación de componentes internos (librerías estándar del lenguaje) pueden llevar a que un programa bien formado en Python no se ejecute correctamente.

4.1.3 Convenciones

Durante el desarrollo de PolyBench/Python se seguirán una serie de convenciones de código para garantizar una estructura coherente. La base de estas convenciones reside en la guía de código propuesta en el PEP 8 [19]. Cabe mencionar que este estilo no se sigue al pie de la letra y ocasionalmente se realizarán excepciones por conveniencia. Por ejemplo, en el PEP se propone un tamaño máximo de línea de 79 caracteres. Esta sugerencia se relaja en PolyBench/Python y pasa a ser de 119 caracteres tanto para código como para comentarios. Por otra parte, durante la implementación de los benchmarks las convenciones para el nombrado de variables no se respetan con el fin de mantener los nombres de los datos lo más idénticos a sus homónimos en PolyBench/C. El resto de reglas seguidas se basan en las sugerencias propuestas por las herramientas de análisis automático que siguen distintas convenciones PEP, referenciadas de forma directa o indirecta en PEP 8.

4.2 Diseño de PolyBench/Python

Hasta ahora se ha estudiado en el capítulo 3 la estructura y funcionamiento de PolyBench/C y se han visto algunos de los problemas que pueden surgir para convertir PolyBench a Python y que afectan directamente al diseño del proyecto.

El objetivo ahora es aplicar el conocimiento visto hasta el momento para tomar las decisiones de diseño que permitan crear PolyBench/Python de forma en que se aprovechen las virtudes del lenguaje Python, pero siempre intentando mantener la mayor funcionalidad y similitud de comportamiento posible con respecto a PolyBench/C.

Un aspecto importante de PolyBench/C consiste en la reutilización de código. Por una parte, el contenido de los ficheros *polybench.c* y *polybench.h* se incluye en todos los benchmarks sin excepción (obviando las opciones de compilación que pueden alterar la inclusión de ciertas partes de código). Otro detalle a tener en cuenta es que la estructura de todos los benchmarks se basa en la estructura definida en el fichero *template-for-new-benchmark.c* apli-

cando ligeras variaciones en los tipos de datos y parámetros que reciben las funciones según las necesidades de cada benchmark. Todo esto en un lenguaje de más alto nivel que C como lo es Python y que además soporta el manejo de objetos, invita a agrupar toda esta funcionalidad común en una clase abstracta. Dicha clase deberá implementar toda la funcionalidad presente en los ficheros *polybench.c* y *polybench.h* (gestión de memoria, medición de tiempos, contadores de eventos PAPI, funciones de control del sistema operativo, etc.) mientras que las funciones presentes en la plantilla *template-for-new-benchmark.c* deberán ser marcadas como abstractas, delegando su implementación específica a cada uno de los distintos benchmarks. Dicha clase deberá también ofrecer mecanismos que permitan controlar el estado de las distintas opciones que controlan el comportamiento de los benchmarks de modo similar al que lo hace PolyBench/C con las opciones en tiempo de compilación.

Dicho esto, PolyBench/Python se puede reducir a dos "grandes problemas" que resolver: la clase abstracta que implementa la funcionalidad necesaria para poder ejecutar un benchmark y por otra parte la implementación de los propios benchmarks, que heredan de dicha clase abstracta e implementan únicamente las partes específicas particulares de cada algoritmo.

Para el diseño de los benchmarks resulta interesante permitir el desarrollo de diferentes implementaciones con el fin de poder evaluar las diferencias de rendimiento entre las opciones disponibles en Python para trabajar con datos de tipo array (el tipo *lista* propio del lenguaje, los arrays de la librería *NumPy*, etc.).

Por último es importante mencionar cómo se pueden organizar los ficheros de código que componen a PolyBench/Python. Una buena idea es aprovechar los mecanismos estándar del lenguaje para el manejo de dependencias y a través de estos definir un *paquete Python*. Dado que el diseño global consiste en utilizar una clase abstracta como plantilla que debe implementar cada benchmark, una opción lógica consiste en posicionar la clase abstracta al nivel de la raíz del paquete mientras que las implementaciones específicas se pueden categorizar en subdirectorios del paquete respetando la categorización definida en PolyBench/C (datamining, linear-algebra, etc.).

4.2.1 Diseño de una clase abstracta

Para el diseño de la clase abstracta es necesario conocer qué elementos se deben incluir. La aproximación más lógica es realizar una extracción de características de PolyBench/C para intentar encontrar patrones que permitan aglutinar la funcionalidad necesaria. Una vez obtenido este conocimiento se pueden aplicar patrones de diseño que permitan desarrollar una implementación abstracta ordenada y coherente con la que poder desarrollar implementaciones concretas de los benchmarks reutilizando la mayor parte posible de código a través de los mecanismos de herencia.

Extracción de características

El primer paso para el diseño de la clase abstracta consiste en definir cuáles son los métodos que debe definir. En este caso se puede comenzar definiendo grupos funcionales a partir de los vistos en el apartado 3.2.3 sobre el fichero *polybench.c* en dónde se describen los siguientes grupos:

1. Gestión de memoria de arrays (montículo/pila; padding)
2. Temporización (medición de tiempos "wall clock" y TSC)
3. Métricas hardware a través de PAPI
4. Funciones del sistema (control del planificador de Linux, vaciado de memoria caché)
5. Control de la instrumentación

Este conjunto de grupos de funciones define las herramientas básicas con las que trabajan los benchmarks y permiten realizar distintas tareas de gestión y medición de rendimiento. Ya que estos grupos de funciones son empleados por todos los benchmarks sus implementaciones deben estar presentes en la clase abstracta para que su código pueda ser reutilizado a través de los mecanismos de herencia. En el apartado 4.3 se detallan las implementaciones de los métodos de cada uno de estos grupos, así como se ilustran los detalles de resolución empleando el patrón de diseño *decorador* para los métodos que requieren algún tipo de inicialización especial (especialmente para métodos que requieren acceso a funciones del sistema o del procesador de muy bajo nivel para los cuales no existe una librería conveniente). Por ahora basta con saber que los métodos que se definen son los mismos que los vistos en PolyBench/C.

El siguiente paso en el diseño de la clase abstracta consiste en definir la interfaz que deben implementar los benchmarks a través de la definición de métodos abstractos. Para definir estos métodos se utiliza la estructura de código definida en el fichero *template-for-new-benchmark.c* vista en el apartado 3.2.4 ya que es la estructura común que utilizan cada uno de los benchmarks de PolyBench/C en su implementación.

En este fichero se definen las firmas de las funciones de inicialización de datos (*init_array()*), impresión de resultados de salida (*print_array()*), ejecución del algoritmo (*kernel()*) y preparación del código/programa principal (*main()*). Estas funciones pueden convertirse en los siguientes métodos abstractos:

```
def initialize_array(self, *args, **kwargs): ...
def print_array_custom(self, array: list, dump_message: str = ''):
    ...
def kernel(self, *args, **kwargs): ...
def run_benchmark(self) -> list[tuple]: ...
```

Estos métodos, en sus respectivas implementaciones dentro de las subclases, deben implementarse de forma similar a los métodos vistos en PolyBench/C de tal modo en que el código

que implementen sea único de cada algoritmo.

Los métodos *initialize_array()* y *kernel()* utilizan una notación especial para los parámetros que permite el paso de un número de parámetros arbitrarios con nombres arbitrarios. De este modo, una subclase puede implementar estos métodos con sus propias firmas sin que haya ningún tipo de problema semántico. Los parámetros especiales "**args*" y "***kwargs*" definen una lista y un diccionario respectivamente que utiliza Python internamente en la llamada a una función. El parámetro "**args*" contiene una lista que se utiliza para llamar a funciones con parámetros anónimos, de tal modo que cada elemento de la lista se corresponde con cada parámetro de la función en el orden en que aparecen. En el caso de "***kwargs*" se tiene un diccionario en el cual las claves representan a los nombres de los parámetros de una función y los valores representan a los valores que se pasan por parámetro. Esta opción permite el paso de parámetros con orden arbitrario.

En las definiciones de métodos abstractos se pueden observar discrepancias en los nombres de dos de ellos con respecto a sus versiones correspondientes en PolyBench/C. En el caso del método *run_benchmark()* que ofrece una funcionalidad similar a main() el razonamiento es meramente lógico. Simplemente se asigna un nombre más significativo dentro del contexto de una clase abstracta ya que un método abstracto llamado *main()* puede dar lugar a confusión. También se puede observar en la firma del método que este devuelve una lista de tuplas. La idea es devolver una lista de resultados, cada uno representado por una tupla. El primer elemento de la tupla se corresponde con un nombre significativo para el resultado mientras que el segundo elemento contiene el resultado, típicamente un array. Esto permite trabajar con los distintos resultados de salida del benchmark desde la propia clase abstracta a la vez que permite a benchmarks como *stencils/fdtd-2d* devolver más de un resultado de salida.

Por otra parte, el método *print_array_custom()* recibe este nombre debido a una abstracción que se realiza en la clase abstracta para facilitar la impresión de datos por pantalla. En esta abstracción el objetivo es aprovechar la versatilidad paramétrica de la función *print()* de Python para moldear su comportamiento adaptándolo a las necesidades concretas de PolyBench, permitiendo realizar la impresión de cadenas de texto de un modo similar a la función *fprintf()* de C para así poder generar resultados imprimibles similares a partir de cadenas de texto idénticas. De este modo, la impresión de datos por pantalla se abstrae a través del método *print_message()* que controla el comportamiento de la función *print()* de Python y, por conveniencia, se definen los métodos *print_value()*, que simplifica la impresión de los valores de un array aplicando un formato apropiado, y *print_array()*, que se encarga de imprimir un mensaje común para todos los arrays y delega la parte específica de la impresión en el método *print_array_custom()* implementado por el benchmark.

Quedan así definidos los métodos de impresión de la clase abstracta y los métodos abstractos que deben implementar los benchmarks.

Tratamiento de opciones

Se ha visto en el capítulo 3 cómo PolyBench/C maneja una serie de opciones de compilación que permiten configurar varios parámetros como por ejemplo seleccionar qué tipo de métrica de rendimiento se va a utilizar durante la ejecución o definir el tamaño del conjunto de datos con los que va a trabajar el algoritmo. Del mismo modo, PolyBench/Python debe ofrecer mecanismos similares para el tratamiento de opciones.

El inconveniente procede de la naturaleza de Python, que es un lenguaje normalmente interpretado a través de un entorno de ejecución y que no permite fijar las opciones en tiempo de compilación puesto que no existe dicha etapa. Para resolver esto, la opción más sencilla consiste en obtener los valores de configuración en tiempo de ejecución. Para ello se crea una clase contenedor que se encarga de almacenar los valores de todas las opciones disponibles. Esta clase se define en el apartado 4.4 y debe ser inicializada por el usuario.

De cara al diseño de la clase abstracta lo que importa es cómo obtener la clase contenedora de opciones para poder manejarlas y seleccionar el flujo de código apropiado. Una opción sería pasar esta clase contenedor como parámetro del método abstracto *run_benchmark()* pero esto tiene el inconveniente de que el código de inicialización de las opciones pasaría a estar manejado por la clase que implemente un benchmark, generando código repetido evitable. Otra opción más natural consiste en utilizar el inicializador de clase, tanto en cada uno de los benchmarks como en la clase abstracta, de modo en que las implementaciones simplemente llamarían al código de inicialización de la superclase abstracta. Esta parece una buena opción ya que se estaría usando un mecanismo estándar de inicialización de clases de Python al reimplementar el método *__init__()*. Esta última es la opción que se implementa y cuyos detalles se describen en el apartado 4.5.

A parte de las opciones de usuario existen otro tipo de parámetros configurables en PolyBench que permiten configurar las especificaciones de los benchmarks a través del fichero *polybench.spec*. Estos parámetros contienen la información sobre tipo de dato y tamaño del conjunto de datos para cada benchmark. Para el manejo de estos parámetros se utiliza otra clase contenedora que debe ser pasada al método *__init__()* junto con las opciones de PolyBench.

Por último sobre la clase abstracta queda detallar la interfaz pública de ejecución de un benchmark. Anteriormente se ha definido el método abstracto *run_benchmark()* que debe implementar cada benchmark. No obstante se ha decidido abstraer la llamada a este método añadiendo un método en la clase abstracta llamado *run()* cuyo objetivo es llamar al método abstracto *run_benchmark()*, usar los datos de salida del mismo para imprimirlos por pantalla (o dónde corresponda según las opciones de ejecución) y devolver los resultados de la ejecución del benchmark (tiempos de ejecución, contadores de eventos PAPI, etc.). De este modo se

permite al usuario trabajar con los resultados de la ejecución según convenga.

4.2.2 Aplicación de la clase abstracta

El objetivo de esta sección es ilustrar la eficacia del diseño la clase abstracta a través de un ejemplo de implementación. Para ello, se ha seleccionado el algoritmo *Jacobi-1D* ya que su implementación es una de las más breves y sencillas. El código de este algoritmo se ilustra al completo en el anexo A.3.

En este ejemplo se puede ver cómo la clase *Jacobi_1D* extiende mediante herencia a la clase abstracta *PolyBench* e implementa todos sus métodos abstractos *initialize_array()*, *print_array_custom()*, *kernel()* y *run_benchmark()*.

Un detalle importante que se puede observar en el método *kernel()* consiste en cómo marcar la región de control estático para que otras herramientas puedan identificar la parte de código que se debe optimizar. En PolyBench/C este código se delimita entre las directivas *#pragma scop* y *#pragma endscop*. Ya que Python no provee de un mecanismo equivalente, la opción que se ha tomado es delimitar estas secciones de código mediante un comentario de una línea que reemplace directamente a los pragmas. De este modo, el código delimitado entre *# scop begin* y *# scop end* conforma la parte de control estático. Esta práctica es común a todos los benchmarks y define una propuesta de marcado de la parte de control estático para el futuro desarrollo de herramientas que se aprovechen de los métodos de optimización poliédrica.

Se puede observar que las firmas de los métodos sobreescritos no coinciden con las firmas de los métodos abstractos definidas en la clase PolyBench. Esto se debe a que los parámetros son dependientes del algoritmo.

En otros lenguajes como Delphi o Java el hecho de definir un nuevo método utilizando diferentes parámetros de entrada se conoce como *sobrecarga* de un método y no es equivalente a una sobreescritura del método (que es lo que se pretende conseguir al implementar un método abstracto) ya que en la sobrecarga el objetivo es poder agrupar con un mismo nombre de método distintas implementaciones, mientras que en la sobreescritura el objetivo es modificar la implementación. No obstante, ya que en Python todos los nombres son referencias a "cosas" (objetos, funciones, etc.), lo que sucede en este caso es efectivamente una sobreescritura del método. Esto se debe a que la referencia más actual para el nombre del método es la más recientemente parseada y en este caso la referencia más reciente coincide siempre con la reescritura realizada por la clase hija.

Gracias a que los métodos abstractos de la clase abstracta se han definido con los parámetros especiales "**args*" y "***kwargs*" los métodos sobreescritos en los benchmarks son sintácticamente correctos siempre y cuándo el número de elementos en la lista *args* (o el diccionario *kwargs*) coincida con el número de parámetros del método.

Antes de terminar con esta sección es importante recordar que uno de los objetivos de PolyBench/Python es permitir evaluar los distintos algoritmos a través de diferentes implementaciones. El ejemplo ilustrado *Jacobi-1D* es lo suficientemente sencillo para mostrar la aplicación de la clase abstracta. No obstante, este ejemplo no ofrece margen para ilustrar posibles implementaciones alternativas para los arrays de datos. En la sección 4.5 se muestra y estudia un ejemplo más detallado en el que los datos de los arrays se organizan en múltiples dimensiones. En este caso, como en la mayoría de los algoritmos que implementa PolyBench, es posible la implementación con estructuras de datos alternativas.

La idea general para facilitar múltiples implementaciones consiste en aplicar una serie de patrones de diseño, concretamente los patrones factoría y estrategia, de tal modo en que la factoría es responsable de seleccionar una estrategia de implementación a partir de las opciones de configuración especificadas por el usuario. Esta combinación de patrones se implementa en el constructor de la clase específica del benchmark con el fin de devolver una estrategia con la implementación correspondiente. Es importante tener en cuenta que en este caso las estrategias son a la vez subclases de la clase que implementa el algoritmo, de tal modo que las estrategias únicamente deben implementar las partes especificas de cada implementación (típicamente los métodos *init_array()* y *kernel()*, y en ocasiones *print_array_custom()*) mientras que la clase principal de implementación del algoritmo debe implementar como mínimo *run_benchmark()*.

4.3 Desarrollo de la clase abstracta PolyBench

En esta sección se pretende explicar los detalles de implementación específicos de la clase abstracta PolyBench vista en el apartado 4.2.1. La estructura de este apartado continúa con la agrupación de funcionalidad vista en el apartado citado respetando el orden de aparición y especificando para cada grupo de métodos los detalles de implementación más relevantes.

4.3.1 Gestión de memoria

Es necesario recordar por un momento que PolyBench/C utiliza dos esquemas diferentes para reservar la memoria de los arrays: a través de la pila y a través del montículo. Ya que en Python no se puede decidir dónde se crea un objeto, la reserva de memoria para las listas se realiza de forma predeterminada en el montículo. Esto reduce el problema a simplemente crear una lista de un tamaño prefijado.

Realizar esta operación para listas de una única dimension es trivial. A través de las comprensiones de las listas se puede, mediante una sintaxis sencilla, crear una lista de un tamaño determinado con un valor prefijado. La sintaxis de las comprensiones de listas se puede consultar en la sección 6.2.4 de la documentación sobre expresiones [20].

El siguiente fragmento de código muestra una comprensión en la que se crea una lista de *5* elementos con un valor predeterminado establecido en *7*.

```
>>> [7 for x in range(5)]
[7, 7, 7, 7, 7]
```

El valor de iteración *x* se ignora ya que no tiene ningún valor práctico en este ejemplo. Es fácil entender que, en cada iteración del bucle, el valor de *x* se irá incrementando desde *0* hasta *4*.

Hasta aquí, lo visto con las comprensiones permitiría crear listas que representen a arrays unidimensionales. Para crear una lista multidimensional, basta con hacer que los elementos de la lista sean a su vez listas. Por ejemplo, la matriz identidad 2x2 se puede representar de la siguiente forma en Python:

```
>>> id_m = [[1, 0], [0, 1]]
```

Nótese que *id_m* es una lista de dos elementos y cada uno de estos elementos es a su vez una lista con dos elementos. Con esta representación se puede acceder a través de índices a los distintos valores de la lista del mismo modo que se realiza para el acceso a elementos del array en C. De este modo, se puede comprobar desde un intérprete que esto es cierto:

```
>>> id_m[0][0]
1
>>> id_m[0][1]
0
>>> id_m[1][0]
0
>>> id_m[1][1]
1
```

Esta matriz se puede crear también a través de comprensiones:

```
>>> id2_m = [[(x + y) % 2 for x in [1, 0]] for y in [0, 1]]
>>> id_m == id2_m
True
```

En este caso es una comprensión anidada en la que el bucle *for* relativo a *x* es descendente y el bucle *for* relativo a *y* es ascendente.

Se puede observar que la sintaxis para crear listas a través de comprensiones es una forma sencilla de crear listas unidimensionales en función de una serie de criterios. No obstante, esta sintaxis no simplifica el proceso de crear listas multidimensionales con N dimensiones arbitrarias.

Conceptualmente, en Python, añadir más dimensiones a una lista es tan sencillo como anidar más listas dentro de las mismas, tantas como dimensiones se deseen. Una forma más natural de crear listas multidimensionales es a través de una función recursiva. Esta función

deberá aceptar como parámetros el número de dimensiones deseadas y, opcionalmente, un valor de inicialización predeterminado. De este modo, se puede crear la siguiente función recursiva:

```
def create_array_rec(dimensions: int, size: int,
    initialization_value: int = 0) -> list:
    if dimensions == 1:
        # Just create a list with as many zeros as specified in size
        return [initialization_value for x in range(size)]
    else:
        # Generate lists of the same size per dimension
        return [create_array_rec(dimensions - 1, size,
    initialization_value) for x in range(size)]
```

Esta función permite crear listas de M dimensiones con N elementos en cada dimensión, inicializando por defecto los elementos de la lista al valor *0*.

Tal y como está definida la función ahora mismo, una lista de N dimensiones está restringida a tener el mismo número de elementos en todas las dimensiones. Para flexibilizar esta parte, se puede realizar un pequeño cambio en la definición de la función, en donde el parámetro *size: int* pasa a ser una lista de tamaños, representando cada elemento de la lista al tamaño de cada dimensión de modo que el primer elemento se corresponde con el tamaño de la primera dimensión, el segundo elemento con el tamaño de la segunda y así sucesivamente. El código de la función pasaría a ser el siguiente:

```
def create_array_rec(dimensions: int, sizes: list,
    initialization_value: int = 0) -> list:
    if dimensions == 1:
        # Just create a list with as many zeros as specified in size
        return [initialization_value for x in range(sizes[0])]
    else:
        # Generate lists with unique sizes per dimension
        return [create_array_rec(dimensions - 1, sizes[1:],
    initialization_value) for x in range(sizes[0])]
```

Con esta nueva definición ya se pueden crear arrays con dimensiones de tamaños arbitrarios de una forma sencilla. No obstante, su usabilidad se ha visto afectada imponiendo una restricción: la lista *sizes* tiene que tener, al menos, tantos elementos como el número de dimensiones deseadas. De no ser así el código mostrado tendría un error de ejecución ya que en la bifurcación *else* se realiza un corte de la lista eliminando el primer elemento para realizar la llamada recursiva. Si no hay elementos que cortar (es el caso de no especificar suficientes elementos) el resultado de *sizes[1:]* devolverá una lista vacía y, en la siguiente iteración fallará al intentar obtener el elemento de la misma. Para arreglar esto basta con añadir una comprobación más sobre esta lista. El siguiente código muestra la versión final de esta función:

```
def create_array_rec(dimensions: int, sizes: list,
    initialization_value: int = 0) -> list:
    if dimensions == 1:
        # Just create a list with as many zeros as specified in
    sizes[0]
        return [initialization_value for x in range(sizes[0])]

    if len(sizes) == 1:
        # Generate lists of the same size per dimension
        return [create_array_rec(dimensions - 1, sizes,
    initialization_value) for x in range(sizes[0])]
    else:
        # Generate lists with unique sizes per dimension
        return [create_array_rec(dimensions - 1, sizes[1:],
    initialization_value) for x in range(sizes[0])]
```

Se ha optado por comprobar cuándo la lista contiene un único elemento. En este caso, la llamada recursiva enviará la misma lista con el único elemento hasta que se creen todas las dimensiones. Haber puesto la comprobación en ese punto tiene otra ventaja añadida y es que ahora al pasar listas con menos elementos que el número de dimensiones el último número de la lista se emplea para el tamaño de cada dimension restante.

Un caso de uso para este tipo de lista se puede ilustrar mediante una matriz. Si la matriz es *MxN*, será necesario especificar tanto *M* como *N* en la lista *sizes*. Sin embargo, si la matriz es *MxM* basta con que la lista *sizes* contenga un único elemento de valor *M*.

El siguiente ejemplo ilustra cómo crear una matriz *MxN* y *MxM*:

```
>>> m = 2; n = 3
>>> mxn = create_array_rec(2, [m, n])  # matriz 2x3
>>> mxm = create_array_rec(2, [m])     # matriz 2x2
```

Hasta aquí se ha visto cómo gestionar la creación de listas de una forma cómoda y sencilla. El siguiente paso, respetando la tabla inicial, consiste en implementar el padding para las listas. En este punto hay que tener en cuenta que el intérprete de Python puede añadir elementos adicionales al final de la lista de forma automática y transparente para el usuario (de hecho, como se ha estudiado al principio del capítulo, CPython y PyPy reservan memoria adicional para intentar mejorar el rendimiento de *append()*), por lo que el número de elementos adicionales que se reservan al final de la lista puede ser mayor al especificado por el usuario. Por otra parte, añadir elementos adicionales al principio de la lista como ocurre en PolyBench/C cuando *POLYBENCH_ENABLE_INTARRAY_PAD* está activado no es posible de implementar sin alterar los índices en la implementación del kernel.

Puesto que sólo se puede implementar el post-padding del array de forma correcta, el lugar más apropiado para hacerlo es en la función de creación de arrays. Ya que el objetivo es, para cada dimensión del array, añadir X elementos. Como se han definido expresamente en una lista los tamaños de las dimensiones basta con modificar estos valores sumando la cantidad de padding que se desea. De este modo pasa a ser interesante una función auxiliar, que será la que utilicen los benchmarks, en la que se realice esta operación. Además es conveniente validar la entrada del usuario (verificar que la lista cumple con unos requisitos mínimos). De este modo, se puede definir la función *create_array()* del siguiente modo:

```
def create_array(self, dimensions: int, sizes: list,
    initialization_value: int = 0) -> list:
    new_sizes = [size + POLYBENCH_PADDING_FACTOR for size in sizes]
    return create_array_rec(dimensions, new_sizes,
    initialization_value)
```

La versión mostrada de la función ha sido simplificada con respecto a la implementación real. Dicha implementación incluye una serie de comprobaciones sobre los datos de entrada debidamente comentadas.

Es importante recordar que, aunque se haya especificado una cantidad de padding, el intérprete internamente puede haber añadido más elementos de los deseados en función de la estrategia interna que implemente. Esto puede tener efectos negativos en el uso de memoria y, muy posiblemente, en los resultados de la ejecución.

Finalmente se dota al método *create_array()* la capacidad de reservar memoria usando otros tipos de datos que no sean listas. Esto es de especial importancia para dar soporte a implementaciones alternativas de algoritmos usando librerías como NumPy que ofrece tipos de datos propios para arrays. Para ello se usa la opción *ARRAY_IMPLEMENTATION* que contiene el tipo de implementación especificado por el usuario. De este modo, el codigo final tiene el siguiente aspecto:

```
def create_array(self, dimensions: int, sizes: list,
    initialization_value: int = 0) -> list:
    # Add post-padding to every array dimension
    new_sizes = [size + self.POLYBENCH_PADDING_FACTOR for size in
    sizes]

    # Expand the new_sizes list to match the number of dimensions
    while len(new_sizes) < dimensions:
        new_sizes.append(sizes[-1])

    # At this point it is safe to say that both dimensions and
    sizes are valid.
    # Use the appropriate "array" implementation.
```

```
11      if self.ARRAY_IMPLEMENTATION == ArrayImplementation.LIST:
12          return self.__create_array_rec(dimensions, new_sizes,
        initialization_value)
13      elif self.ARRAY_IMPLEMENTATION ==
        ArrayImplementation.LIST_FLATTENED:
14          # A flattened list only has one dimension, whose value is
        the product of all dimensions.
15          dimension_size = 1
16          for dim_size in new_sizes:
17              dimension_size *= dim_size
18          return self.__create_array_rec(1, [dimension_size],
        initialization_value)
19      elif self.ARRAY_IMPLEMENTATION == ArrayImplementation.NUMPY:
20          # Create an auxiliary list for creating an initialized
        NumPy array.
21          list_array = self.__create_array_rec(dimensions, new_sizes,
        initialization_value)
22          return numpy.array(list_array, self.DATA_TYPE)
23      else:
24          raise NotImplementedError(f'Unknown internal array
        implementation: "{self.ARRAY_IMPLEMENTATION}"')
```

En el caso de necesitar otras implementaciones de array futuras será necesario modificar este método.

4.3.2 Temporización

El apartado de medición interna de tiempos de PolyBench/C parece inofensivo de traducir a Python, sin embargo al inspeccionar la implementación en C en seguida se puede percibir que puede ser problemático.

En función de la opción *POLYBENCH_CYCLE_ACCURATE_TIMER* se obtiene la medición de referencia de una fuente o de otra. De forma predeterminada esta opción se encuentra desactivada y el resultado de realizar un muestreo temporal proviene del valor devuelto por la llamada al sistema de Linux *gettimeofday()*. Esta función suele devolver el valor leído del *RTC* del sistema.

Cuando la opción *POLYBENCH_CYCLE_ACCURATE_TIMER* se encuentra activa, el muestreo temporal se realiza a través del registro *TSC* específico de la arquitectura x86. Como se ha visto en la sección 3.2.3 en el apartado de las funciones de temporización, este registro mide el número de ciclos de la CPU desde el último reset.

En Python la librería *time* [21] provee distintas funciones de temporización. Una función interesante resulta ser *time()* ya que, en sistemas Unix en dónde esté disponible, utiliza *get-*

timeofday(), permitiendo que el contador predeterminado tenga una implementación equivalente.

El problema viene con el contador *TSC*. La librería *time* de Python provee la función *monotonic()* que, de acuerdo al PEP-418 [22], puede usar (sin garantías) el registro *TSC* en sistemas Unix. No obstante, la librería *time* especifica que todas las funciones relacionadas con mediciones devuelvan resultados en unidades temporales (segundos, microsegundos, etc.) mientras que el registro *TSC* devuelve el número de ciclos entre llamadas, de modo en que la función *monotonic()* deja de ser interesante.

Para poder comparar resultados entre PolyBench/C y PolyBench/Python es necesario acceder al valor de este registro desde Python. Una opción es usar alguna librería externa que implemente el acceso a este contador. El problema es que las librerías que hacen uso del registro TSC suelen ser *frameworks* completos. Otro problema es que estas librerías suelen hacer uso de mecanismos específicos del intérprete CPython para acceder a librerías C y son incompatibles con PyPy. Sabiendo esto, puede ser más interesante implementar dicha librería de forma manual, aunque esta opción es la menos elegante pues añade pasos adicionales previos a la ejecución de un benchmark.

Una opción ideal sería poder incluir el código ensamblador necesario en línea dentro del código Python. Existen varios proyectos que permiten realizar esta tarea de forma muy sencilla. Por ejemplo, está el proyecto *il* [23] que permite definir funciones Python con código ensamblador a través de anotaciones. Sin embargo esta librería no es compatible con PyPy al depender de una característica de la librería estándar *ctypes* (pythonapi). Otra librería que tiene un modo de empleo similar sintácticamente es *inlineasm* [24]. No obstante esta librería tiene dos detalles importantes a tener en cuenta. Uno es que requiere el compilador *nasm* para funcionar. Por otra parte, el código no parece que reciba mantenimiento, por lo que puede dejar de funcionar en futuras versiones de Python.

La opción seleccionada por comodidad ha sido *inlineasm*. Para utilizar esta librería basta con importar el módulo *assemble* que permite llamar al ensamblador de código *nasm* con el fragmento de código deseado y adicionalmente se debe incluir del módulo estándar de Python *ctypes* el tipo de dato C que devuelve la función definida en código ensamblador. Para definir el código a ensamblar basta con definir una cadena de texto multilínea debidamente formateada utilizando la notación aceptada por *nasm*. Una vez definida esta cadena basta con llamar a la función *assemble()* especificando el código a ensamblar y el tipo de dato que debe devolver. La función *assemble()* devuelve una referencia a la función creada en el código ensamblador. Dicha referencia puede ser utilizada como cualquier referencia a una función de Python. El siguiente fragmento de código ilustra cómo se crea la referencia al código ensamblado en el código de inicialización de la clase PolyBench. El código ensamblador se ha extraido del propio código generado por PolyBench/C para leer el TSC aplicando el nivel de optimización *-O2* en

el compilador *gcc*.

```
from inlineasm import assemble
from ctypes import c_ulonglong

class PolyBench:
    def __init__(self):
        ...
        asm_code = """
            bits 64
            RDTSC
            sal     rdx, 32
            mov     eax, eax
            or      rax, rdx
            ret
        """
        self._read_tsc = assemble(asm_code, c_ulonglong)
```

En la última línea del código se crea la referencia *self._read_tsc* que se asigna a la función definida en el código ensamblador. El código ensamblado for la función *assemble()* se almacena en una librería dinámica que se autogenera en cada instanciación de la clase *PolyBench*. Esta librería se almacena en el directorio temporal del sistema.

Para utilizar la función de lectura del TSC basta con "llamar" a la referencia empleando la misma sintaxis que en cualquier otra función de Python. De este modo, siempre en el contexto de la clase PolyBench (y subclases), el código *self._read_tsc()* llamará a la función de lectura del TSC y devolverá el valor actual de dicho registro.

Con lo visto hasta el momento en este apartado de temporización ya se pueden implementar los métodos encargados de iniciar, finalizar e imprimir los valores de los temporizadores. De este modo, se definen los métodos privados *__timer_start()*, *__timer_stop()* y *__timer_print()*. Estos métodos no deben ser llamados directamente por ninguna subclase ni por el usuario ya que se utilizan de forma indirecta a través del método de instrumentación *time_kernel()* que se encarga de realizar el muestreo y la ejecución de un benchmark.

```
def __timer_start(self):
    self.__prepare_instruments()
    if not self.POLYBENCH_CYCLE_ACCURATE_TIMER:
        self.__timer_start_t = time()
    else:
        self.__timer_start_t = self._read_tsc()

def __timer_stop(self):
    if not self.POLYBENCH_CYCLE_ACCURATE_TIMER:
        self.__timer_stop_t = time()
```

```
    else:
        self.__timer_stop_t = self._read_tsc()
    if self.POLYBENCH_LINUX_FIFO_SCHEDULER:
        self.__linux_standard_scheduler()

def __timer_print(self):
    self.polybench_result = self.__timer_stop_t -
    self.__timer_start_t
    if not self.POLYBENCH_CYCLE_ACCURATE_TIMER:
        print(f'{self.polybench_result:0.6f}')
    else:
        print(f'{self.polybench_result:d}')
```

El objetivo del método *__timer_start()* consiste en almacenar en el atributo privado de clase *__timer_start_t* el instante en el que se comienza a medir el tiempo, ya sea en formato de reloj o en ciclos de CPU. Un detalle de este método es que también es responsable de inicializar la instrumentación a través de la llamada al método *__prepare_instruments()*.

De forma análoga al método *__timer_start()*, el método *__timer_stop()* se encarga de almacenar en el atributo privado de clase *__timer_stop_t* el instante en el que finaliza la medición temporal. Adicionalmente, en PolyBench/C, la función *polybench_timer_stop()* se encarga de restaurar el estado del planificador de tiempo de CPU para el proceso actual. En la implementación de PolyBench/Python esta característica se mueve al método *time_kernel()* en la sección de instrumentación. Los detalles sobre el planificador de tiempo de CPU se detallan en la sección de las herramientas del sistema.

Finalmente el método *__timer_print()* se encarga de imprimir la diferencia entre los tiempos marcados utilizando un formato idéntico al empleado en PolyBench/C. En este método se asigna también el resultado de la monitorización del benchmark y se almacena en el atributo *self.polybench_result* para poder devolverlo a través del método *run()*. Un detalle que se ha omitido con respecto a PolyBench/C es el procesamiento de la opción *POLYBENCH_GFLOPS* ya que el comportamiento de esta opción no parece estar debidamente definido.

4.3.3 Métricas hardware a través de PAPI

Las métricas PAPI ofrecen información de rendimiento sobre un proceso directamente desde la CPU. Para acceder a estas métricas en Linux, PolyBench/C utiliza la librería *papi.h* que simplifica el acceso a los distintos contadores de rendimiento del procesador. El uso de este tipo de métricas para un benchmark viene determinado por la opción de configuración *POLYBENCH_PAPI*.

La implementación de manejo de los contadores en PolyBench/C se realiza a muy bajo nivel pese a utilizar una librería, de modo en que se implementan una serie de funciones que permiten inicializar y finalizar la librería, así como inicializar y finalizar la monitorización

de un contador y su impresión por pantalla. Las funciones responsables de estas acciones se encuentran en el fichero *utilities/polybench.c* y son las siguientes: *polybench_papi_init()*, *polybench_papi_close()*, *polybench_papi_start_counter()*, *polybench_papi_stop_counter()* y *polybench_papi_print()*. Ya que el acceso es de muy bajo nivel pese a estar abstraído por una librería, el código de estas funciones es complejo e incluye numerosas comprobaciones de errores, siendo las más comunes abstraídas a través de la función *test_fail()*.

Para el acceso a los contadores PAPI en Python existe una librería, *python_papi*, que simplifica mucho el manejo de los contadores. Esta librería provee dos tipos de interfaces, una de alto nivel, que es la que se usará durante este desarrollo, y otra de bajo nivel, que no provee ninguna funcionalidad interesante para este proyecto. Al utilizar la librería de alto nivel se elimina la necesidad de inicializar y finalizar manualmente la propia librería. De este modo, las funciones *polybench_papi_init()* y *polybench_papi_close()* de PolyBench/C no requieren ser reimplementadas al ser gestionadas de forma transparente por la librería *python_papi*. Por otra parte, las funciones de manejo de contadores de PolyBench/C *polybench_papi_start_counter()* y *polybench_papi_stop_counter()* tampoco requieren ser reescritas ya que la propia librería de Python incluye la misma funcionalidad a través de las funciones *start_counters(events: list)* y *stop_counters() -> list*. En la sección de instrumentación se muestra el uso de estas funciones a través del método *time_kernel()*.

Hasta aquí parece que la librería ha solucionado prácticamente todo el apartado de métricas hardware, quedando pendientes de implementar la impresión por pantalla de los resultados de los contadores y el análisis del fichero *papi_counters.list*. Sin embargo existe un detalle muy importante sobre el manejo de los contadores PAPI. Aunque el hardware pueda permitir la monitorización de varios contadores de forma simultánea esta opción no es recomendable ya que puede causar diferentes problemas. Para evitar los posibles problemas, los diseñadores de PolyBench/C han decidido realizar la monitorización de los distintos contadores especificados por el usuario uno por uno a través de un bucle *for*. Esto se comenta con más detalle junto con el método *time_kernel()* en la sección de instrumentación 4.3.5 en dónde se aplica una solución similar a la utilizada en PolyBench/C.

Analizador del fichero *papi_counters.list*

PolyBench/C utiliza el fichero *papi_counters.list* a través de la directiva de preprocesador *#include* dentro del cuerpo de inicialización de un array que contiene el listado de eventos PAPI del usuario. De este modo se evita tener que realizar un analizador sintáctico para el mismo. El siguiente fragmento de código extraído de PolyBench/C es el responsable de incluir los eventos PAPI.

```
1     char* _polybench_papi_eventlist[] = {
2 #include "papi_counters.list"
```

```
    NULL
    };
```

Una consecuencia de utilizar el preprocesador de este modo es que el contenido del fichero de texto debe ser código C válido dentro de un contexto de inicialización de arrays. Además, en caso de tener contenido, este debe terminar en una coma, ya que se puede observar que el array termina con un elemento nulo.

El siguiente fragmento de código es el contenido del fichero *papi_counters.list*. En él se puede apreciar que la sintaxis es compatible con la inicialización de un array C y que además el último elemento termina con una coma.

```
// Counters must be delimited with ',' including the last one.
// C/C++ comments are allowed.
// Both native and standard PAPI events are supported.
"PAPI_TOT_CYC",
"L1D:REPL",
```

Al realizar la sustitución de la directiva de preprocesador con el contenido del fichero de texto, la definición del array queda del siguiente modo:

```
    char* _polybench_papi_eventlist[] = {
// Counters must be delimited with ',' including the last one.
// C/C++ comments are allowed.
// Both native and standard PAPI events are supported.
"PAPI_TOT_CYC",
"L1D:REPL",
    NULL
    };
```

Los detalles vistos sobre el fichero *papi_counters.list* deben tenerse en cuenta para permitir que la versión Python pueda compartir dicho fichero.

Para poder utilizar el fichero *papi_counters.list* en Python es necesario implementar un analizador sintáctico sencillo. Como este fichero soporta la sintaxis de C y C++ en el contexto de inicialización de los arrays, la combinación de tokens válidos en este contexto puede volverse compleja. Por esta razón PolyBench/Python requiere que el contenido de este fichero sea lo más sencillo posible restringiendo el tipo de tokens que pueden aparecer en él. De este modo se implementa un analizador sintáctico que interpreta el contenido del fichero línea por línea, permitiendo la existencia de los siguientes elementos en una línea cualquiera:

- Comentarios de una línea cuando la línea comienza con un token de comentario simple "//".
- Comentarios multilínea siempre y cuando el token de inicio de comentario "/*" sea el comienzo de la línea y el token de fin de comentario "*/" aparezca el final de la línea.

En caso de que el token de fin de comentario aparezca en una línea diferente a la que se encuentre el token de inicio de comentario, las líneas intermedias se ignoran.

- El resto de líneas se tratan como identificadores de PAPI eliminando los caracteres de comilla doble ” y coma “,”.

En el siguiente fragmento de código se ilustran ejemplos de sintaxis válida e inválida.

```
/* Esto es un comentario válido */
/* Esto es
   también
   un comentario válido */
// Esto es un comentario válido
/* Esto es
// también válido */
"PAPI_TOT_CYC",
"L1D:REPL",

// Las siguientes líneas son válidas en C y C++ pero no en Python
"PAPI_TOT_CYC",  // Comentario inválido
"L1D:REPL",  /* Comentario inválido */
/* Comentario inválido */ "PAPI_BRU_IDL",
"PAPI_BR_CN", /* comentario
inválido */
```

Con los requisitos bien definidos sólamente falta implementar el analizador sintáctico. Ya que sólamente realiza un análisis a nivel de línea sobre comentarios, su implementación resulta casi trivial. La función *parse_counters_file()* implementa el código necesario para realizar este análisis y su objetivo es devolver una lista de strings. Siempre y cuando el formato del fichero de entrada sea válido cada string de la lista representa o bien un comentario o bien un identificador de evento PAPI.

```
def parse_counters_file() -> list:
    result = []
    with open('papi_counters.list') as f:
        contents = f.read()
        # Remove both empty lines and whitespaces
        lines = contents.splitlines()
        lines = [line.strip() for line in lines]

        is_in_comment = False
        for line in lines:
            if not is_in_comment:
                if line.startswith('/*'):
                    is_in_comment = True
                    continue
                elif line.startswith('//'):
```

```
                    continue
                else:
                    result.append(line.strip('",'))  # store plain
    counter names
            else:
                if line.endswith('*/'):
                    is_in_comment = False
    return result
```

En este momento se tiene una lista de identificadores en formato texto pero la librería *python_papi* requiere un identificador de evento numérico para funcionar. En el módulo *events* de *python_papi* se definen los eventos PAPI genéricos como referencias a números, es decir, algo del estilo *PAPI_TOT_CYC = 0x3B*. El objetivo ahora es poder convertir la referencia textual obtenida en el fichero de texto en una referencia numérica aceptada por PAPI. Para ello se puede utilizar el módulo *inspect* de Python, el cual ofrece la función *getmembers()* que aplicado sobre un módulo (en este caso el módulo *events*) devuelve una lista de tuplas en la que cada elemento (tupla) representa a un objeto diferente de la clase (en este caso las referencias a los eventos PAPI). Los valores de la tupla ofrecen el nombre de la referencia en formato *string* junto con el valor asociado a dicha referencia. El código responsable de obtener esta lista de tuplas se puede ver en la función *get_available_counters() -> list*. Esta función utiliza a otra función auxiliar *is_number()* para filtrar del módulo los valores de tipo numérico, que es el tipo que utilizan los eventos PAPI. Todo esto se ilustra en el siguiente fragmento de código.

```
def get_available_counters() -> list:
    def is_number(x):
        return isinstance(x, int) or isinstance(x, float) or
    isinstance(x, complex)
    from inspect import getmembers
    return getmembers(papi_events, is_number)
```

En este momento ya se puede construir una lista de eventos PAPI válidos para la librería *python_papi* a partir de los datos obtenidos en el fichero *papi_counters.list*. Para ello basta con comparar los identificadores del fichero de texto con los nombres contenidos en el primer campo de las tuplas. En caso de que los identificadores coincidan, se almacena el valor del segundo elemento de la tupla en la lista de eventos PAPI. Todo este proceso se puede incluir en un método llamado *__papi_init()*, cuyo código se puede ver a continuación. Nótese que este código utiliza las funciones auxiliares vistas anteriormente *get_available_counters()* y *parse_counters_file()*.

```
def __papi_init(self):
    self.__papi_available_counters = get_available_counters()
    user_counters = parse_counters_file()

```

```
    # Check if the user counters exist within the available
    standard set.
    for usr_counter in user_counters:
        is_available = False
        for avlbl_counter in available_counters:
            if usr_counter == avlbl_counter[0]:
                self.__papi_counters.append(avlbl_counter[1])
                is_available = True
                break
        if not is_available:
            print(f'WARNING: counter "{usr_counter}" not
    available.')
```

Impresión de contadores

La impresión por pantalla de los contadores se ciñe al formato definido en PolyBench/C y no tiene muchos detalles que comentar. Su código se muestra a continuación.

```
def __papi_print(self):
    self.polybench_result = {}
    counter_names = papi_counter_names()

    for i in range(0, len(self.__papi_counters)):
        if self.POLYBENCH_PAPI_VERBOSE:
            print(f'{counter_names[i]}=', end='')
        print(f'{self.__papi_counters_result[i]} ', end='')
        if self.POLYBENCH_PAPI_VERBOSE:
            print()  # new line

        # Append key-value to result (name-value)
        self.polybench_result[counter_names[i]] =
    self.__papi_counters_result[i]
    print()  # new line
```

De este código hay que comentar dos cosas. Una de ellas tiene que ver con el atributo *self.polybench_result*. Este atributo es una referencia a un diccionario en el que se almacenan los resultados de la monitorización del benchmark. Esta misma función se ha visto en el método *__timer_print()* y el objetivo es igualmente permitir exportar los resultados de la monitorización a través del método *run()* para que el usuario pueda trabajar con ellos.

El otro detalle en este método es la llamada a la función *papi_counter_names()*. Esta función se encarga de traducir un identificador numérico de evento PAPI a una representación textual que sea entendible por el usuario. El código de esta función se muestra a continuación. Cabe recordar que el atributo *self.__papi_counters* contiene el listado de identificadores

numéricos PAPI y *self.__papi_available_counters* contiene la lista de tuplas generada en *__papi_init()*.

```
def papi_counter_names():
    result = []
    for usr_counter in self.__papi_counters:
        for avlbl_counter in self.__papi_available_counters:
            if usr_counter == avlbl_counter[1]:
                result.append(avlbl_counter[0])
                break
    return result
```

4.3.4 Funciones del sistema

Este apartado muestra el proceso de traducción de las distintas funciones que controlan el comportamiento del sistema. El objetivo de estas funciones es alterar el estado del sistema previo a la ejecución de un kernel para intentar reducir la interferencia que puedan causar el sistema operativo y otros procesos.

La primera de estas funciones es *polybench_flush_cache()*. El objetivo de esta función es simplemente intentar llenar la memoria caché del procesador con datos irrelevantes. Para ello, se crea un array de gran tamaño y se realiza una operación sencilla sobre el mismo. Actualmente el tamaño de este array es de aproximadamente 32MiB y se puede configurar a través de la opción *POLYBENCH_CACHE_SIZE_KB*.

La implementación más aproximada se muestra a continuación. Nótese que el tamaño del tipo de dato se estima en función del conocimiento de la plataforma. En este caso, se sabe que los enteros de punto flotante se implementan como *double* en CPython y PyPy. En el caso de PyPy, el tratamiento de los tipos primitivos se detalla en la documentación de RPython [25].

```
def __flush_cache(self):
    # sizeof(double) is typically 8
    cs = self.POLYBENCH_CACHE_SIZE_KB * 1024 / 8
    flush = [0.0 for _ in range(cs)]
    tmp = 0.0
    for i in range(cs):
        tmp += flush[i]
    assert tmp <= 10
```

Las funciones restantes permiten configurar el planificador de procesos del sistema. Para poder ejecutar estas funciones es necesario tener privilegios elevados.

La función *polybench_linux_fifo_scheduler()* permite establecer el planificador FIFO para el proceso actual mientras que la función *polybench_linux_standard_scheduler()* restaura el planificador predeterminado.

Desde Python 3.3 es posible configurar el planificador a través del módulo *os* [26]. Esto permite realizar una traducción directa de PolyBench/C a Python. No obstante, el intérprete PyPy no implementa todas las funciones de *scheduling*, por lo que es necesario tomar otra dirección. Con vistas a que una versión futura lo implemente, la solución temporal (se podría llamar apaño) consiste en embedir directamente un fragmento de código C, extraído de PolyBench/C, que implemente esas funciones. Para poder realizar esto se utiliza la librería *inline* [27] que permite compilar código C dentro de Python con una sintaxis sencilla. El siguiente fragmento de código muestra cómo se usa esta librería para crear las funciones C. Este fragmento de código se encuentra en la selección de inicialización de la clase abstracta PolyBench.

```
if python_implementation() != 'CPython':
    from inline import c
    # Linux scheduler code snippets taken from PolyBench/C
    linux_shedulers = c('''
        #include <sched.h>
        void polybench_linux_fifo_scheduler() {
            struct sched_param schedParam;
            schedParam.sched_priority = sched_get_priority_max
    (SCHED_FIFO);
            sched_setscheduler (0, SCHED_FIFO, &schedParam);
        }
        void polybench_linux_standard_scheduler() {
            struct sched_param schedParam;
            schedParam.sched_priority = sched_get_priority_max
    (SCHED_OTHER);
            sched_setscheduler (0, SCHED_OTHER, &schedParam);
        }
    ''')
    self.__native_linux_fifo_scheduler =
    linux_shedulers.polybench_linux_fifo_scheduler
    self.__native_linux_standard_scheduler =
    linux_shedulers.polybench_linux_standard_scheduler
```

Lo que realiza este fragmento de código es, en caso de que el intérprete en el que se ejecuta el código no sea CPython, compila los extractos de código que definen las funciones de control del planificador, generando en el directorio temporal del sistema una librería dinámica contra la que trabajar. Finalmente, sobreescribe las referencias a los métodos *self.__native_linux_fifo_scheduler* y *self.__native_linux_standard_scheduler* con las referencias de la librería dinámica generada.

En el siguiente fragmento de código se muestran los métodos *__linux_fifo_scheduler()* y *__linux_standard_scheduler()* de la clase abstracta que implementan el control del planificador. En estos métodos se puede ver cómo, en función del intérprete en el que se esté ejecutando el código, se selecciona la implementación apropiada. En el caso del intérprete CPython se utilizan las funciones del módulo estándar *os*, mientras que para cualquier otro intérprete se utilizan las llamadas a la librería dinámica generada a partir del código C.

```
def __linux_fifo_scheduler(self):
    if python_implementation() == 'CPython':
        param = os.sched_param(os.SCHED_FIFO)
        os.sched_setscheduler(0, os.SCHED_FIFO, param)
    else:
        self.__native_linux_fifo_scheduler()

def __linux_standard_scheduler(self):
    if python_implementation() == 'CPython':
        param = os.sched_param(os.SCHED_OTHER)
        os.sched_setscheduler(0, os.SCHED_OTHER, param)
    else:
        self.__native_linux_standard_scheduler()
```

4.3.5 Instrumentación

Las funciones de instrumentación permiten gestionar los distintos tipos de monitores de benchmarks (temporizadores, contadores PAPI) ofreciendo una pequeña abstracción para el control de los mismos. Esta abstracción es usada por los benchmarks para controlar la inicialización, finalización e impresión de los elementos de monitorización.

PolyBench/C implementa el control de estos elementos a través de funciones y macros. Siendo precisos, sólamente implementa una función, *polybench_prepare_instruments()*. Las macros se "mapean" en funciones de temporización o de control PAPI en función de la presencia de una de las opciones *POLYBENCH_TIME* y *POLYBENCH_PAPI*.

En el caso de que la opción *POLYBENCH_TIME* se encuentre activa, las macros *polybench_start_instruments*, *polybench_stop_instruments* y *polybench_print_instruments* se mapean en las funciones de temporización *polybench_timer_start()*, *polybench_timer_stop()* y *polybench_timer_print()* respectivamente.

Por otra parte, en caso de que la opción *POLYBENCH_PAPI* se encuentre activa las macros se mapean de forma diferente. En lugar de cambiar la macro por una función como en el caso de *POLYBENCH_TIME* lo que ocurre es que las macros se sustituyen por fragmentos de código. En el caso de *polybench_start_instruments* se sustituye por código que se encarga de inicializar la instrumentación y la librería PAPI para después ejecutar el kernel del benchmark utilizando los contadores PAPI especificados en el fichero *papi_counters.list*. La ejecución del kernel se

realiza dentro de un bucle for, seleccionando en cada iteración un contador PAPI diferente e inicializando su monitorización. De este modo, PolyBench/C evita posibles problemas que pueden surgir cuando se monitorizan múltiples eventos PAPI de forma simultánea. Un detalle de este código es que, dentro de la función *main()* del código de un benchmark, se sitúa justo antes de la llamada a la función *kernel()* y después de la inicialización de datos. Esto tiene como consecuencia que los datos con los que trabaja el kernel pueden ser diferentes a lo largo de las ejecuciones ya que los benchmarks pueden modificar los distintos arrays de datos.

Continuando con las sustituciones de código con la opción *POLYBENCH_PAPI*, la macro *polybench_stop_instruments* se sustituye por el código de finalización del bucle que comienza *polybench_start_instruments*, así como el código que detiene la monitorización del contador activo en el bucle. Finalmente la macro *polybench_print_instruments* se mapea directamente con la función *polybench_papi_print()* que se encarga de imprimir por pantalla los resultados de los contadores.

Lo mencionado hasta aquí se ilustra mejor mediante un extracto de código de la función *main()* vista en la sección 3.2.4 y aplicando sustituciones. Basta con observar las líneas *polybench_start_instruments* y *polybench_print_instruments* y aplicar las sustituciones apropiadas con la opción *POLYBENCH_TIME* para obtener el código que se ilustra a continuación.

```
int main(int argc, char** argv)
{
    ...
    /* Start timer. */
    polybench_timer_start();

    /* Run kernel. */
    kernel_template (n, POLYBENCH_ARRAY(C));

    /* Stop and print timer. */
    polybench_timer_stop();
    polybench_timer_print();
    ...
}
```

Utilizando la opción *POLYBENCH_PAPI* la sustitución de código quedaría:

```
int main(int argc, char** argv)
{
    ...
    /* Start timer. */
    polybench_prepare_instruments();
    polybench_papi_init();
    int evid;
    for (evid = 0; polybench_papi_eventlist[evid] != 0; evid++)
    {
```

```
10          if (polybench_papi_start_counter(evid))
11              continue;
12
13          /* Run kernel. */
14          kernel_template (n, POLYBENCH_ARRAY(C));
15
16          /* Stop and print timer. */
17          polybench_papi_stop_counter(evid);
18      }
19      polybench_papi_close();
20      polybench_papi_print();
21      ...
22  }
```

Ya que este comportamiento se tiene que trasladar a PolyBench/Python en dónde no se pueden aplicar reglas similares a las sustituciones de macros de C, la opción que se toma es aglutinar el comportamiento de control de la instrumentación (inicio y fin) en un único método *time_kernel()* que permite con una simple estructura de bifurcación *if* seleccionar el esquema monitorización apropiado en función de las opciones *POLYBENCH_TIME* y *POLYBENCH_PAPI*. Por otra parte, la función *polybench_prepare_instruments()* se implementa con el método de la clase abstracta *__prepare_instruments()* y la macro *polybench_print_instruments*, que se convierte bien en una función de tiempo o bien en una función PAPI, se implementa con el método de la clase abstracta *__print_instruments()*.

A continuación se muestra el código de los métodos *__print_instruments()* y *__prepare_instruments()* por tener una implementación sencilla. El objetivo del primero es simplemente imprimir por pantalla o bien los contadores temporales o bien los contadores PAPI, mientras que el objetivo del segundo método es simplemente realizar una serie de preparativos previos a la ejecución del benchmark.

```
1  def __print_instruments(self):
2      if self.POLYBENCH_TIME or self.POLYBENCH_GFLOPS:
3          self.__timer_print()
4      elif self.POLYBENCH_PAPI:
5          self.__papi_print()
6
7  def __prepare_instruments(self):
8      if not self.POLYBENCH_NO_FLUSH_CACHE:
9          self.__flush_cache()
10     if self.POLYBENCH_LINUX_FIFO_SCHEDULER:
11         self.__linux_fifo_scheduler()
```

Por último en esta sección queda por explicar el funcionamiento del método *time_kernel()*. Como se mencionaba anteriormente, el objetivo de este método consiste en inicializar los mo-

nitores de medición apropiados, ejecutar el algoritmo y finalizar los contadores. A continuación se muestra el código de dicho método.

```
def time_kernel(self, *args, **kwargs):
    if self.POLYBENCH_TIME or self.POLYBENCH_GFLOPS:
        self.__timer_start()
        self.kernel(*args, **kwargs)
        self.__timer_stop()
    elif self.POLYBENCH_PAPI:
        i = 0
        self.__papi_init()
        self.__prepare_instruments()
        for counter in self.__papi_counters:
            if i > 0:
                self.initialize_array(*args, **kwargs)  # force
    initialization
            i += 1
            papi_high.start_counters([counter])
            self.kernel(*args, **kwargs)

    self.__papi_counters_result.extend(papi_high.stop_counters())
    else:
        self.__prepare_instruments()
        self.kernel(*args, **kwargs)

    if self.POLYBENCH_LINUX_FIFO_SCHEDULER:
        self.__linux_standard_scheduler()
```

Lo primero que se puede observar en el código es el tipo de parámetros que admite. Nuevamente se utilizan los parámetros especiales **args* y ***kwargs* para el paso de parámetros de forma arbitraria. La idea es que un benchmark llame al método *time_kernel()* con una serie de parámetros que son desconocidos para este método pero que por otra parte coinciden con los parámetros que debe recibir el método *kernel()*. Presuponiendo que el método del kernel tiene la siguiente firma: *kernel(array1, array2, array3)*, se tiene que el benchmark llamará al método *time_kernel(array1, array2, array3)* y éste pasará esa lista de parámetros al método *kernel()* a través de **args* y ***kwargs*.

Con respecto a la parte del método relacionada con PAPI, en ella se puede ver cómo se llama de forma explícita al método *__prepare_instruments()* mientras que en la parte del temporizador no aparece la llamada al mismo. Esto se debe a que el método *__timer_start()* llama internamente a *__prepare_instruments()*. Otro detalle que se puede observar es el bucle *for* que se encarga de las ejecuciones del kernel. Este bucle depende de los atributos de clase *self.__papi_counters* y *self.__papi_counters_result*. El primero de ellos se inicializa en el método *__papi_init()* tal y como se mostró en la sección 4.3.3 sobre PAPI. El segundo atributo es una

lista en la que se van almacenando los resultados de los distintos contadores en el orden en que se van ejecutando. Estos dos atributos permiten imprimir los resultados de los contadores PAPI más tarde en la ejecución.

Dentro del bucle *for* lo primero que se ve es una diferencia frente a PolyBench/C en la cual se fuerza la inicialización de los arrays del benchmark para cada contador. En PolyBench/C esta inicialización sólo se realiza para el primer contador. Sin embargo, en Python, no reiniciar los arrays de datos causa problemas en la ejecución de algunos benchmarks relacionados con el redondeo de los valores. El problema principal es la división por cero en los benchmarks *cholesky*, *gramschmidt*, *lu* y *ludcmp* que no ocurre en PolyBench/C.

Lo siguiente que se puede observar en el bucle es cómo se utiliza la librería *python_papi* para inicializar y finalizar la monitorización de contadores. Con respecto a la librería, la llamada a *papi_high.start_counters()* espera como argumento una lista de identificadores de contador. Ya que se están midiendo uno a uno para evitar problemas (interferencias entre contadores, incompatibilidades, etc.) es necesario crear una lista de un único elemento en la llamada para que funcione correctamente. Por otra parte, con respecto a *papi_high.stop_counters()*, esta llamada se encarga de finalizar la medición de contadores y devuelve una lista con los resultados de los contadores monitorizados. Estos contadores se añaden a la lista de resultados a través de la llamada a *extend()*.

La rama *else* de la bifurcación implementa una llamada al algoritmo en la que no se pretende realizar una medición temporal ni de contadores. Esta opción es útil, por ejemplo, para validar los resultados de la operación imprimiendo los arrays.

Finalmente el método *time_kernel()* es responsable de restaurar el planificador de CPU en caso de que éste haya sido alterado para realizar el benchmark. En PolyBench/C esta restauración se realiza únicamente en la función *polybench_timer_stop()* mientras que en PolyBench/Python se opta por restaurar el planificador al finalizar el método *time_kernel()* ya que de este modo se restaura siempre que se termine de realizar una monitorización.

4.3.6 Métodos propios de PolyBench/Python

El objetivo de esta sección es comentar los detalles de implementación de los métodos únicos de PolyBench/Python que permiten la implementación de benchmarks de un modo fácil y cómodo.

Inicializador de clase

El primer método que se debe contemplar es el inicializador de la clase *__init__()*. El objetivo de este método consiste en asegurarse de que la clase PolyBench no se instancia directamente para evitar problemas durante la ejecución. Otra tarea de este método consiste en

validar los parámetros de entrada con los que se ha instanciado una subclase. Para ello, es necesario que las subclases reimplementen este método llamando a la versión de la superclase, tal y como se muestra a continuación.

```
class Jacobi_1d(PolyBench):

    def __init__(self, options: PolyBenchOptions, parameters:
    PolyBenchSpec):
        super().__init__(options, parameters)
        ...
```

Una vez validados los parámetros de entrada el método *__init__()* de PolyBench se encarga de inicializar las opciones de configuración que emplean los distintos métodos (métricas PAPI, impresión por pantalla, etc.) utilizando los valores almacenados en el objeto *PolyBenchOptions* pasado como primer parámetro. Las opciones de configuración se almacenan internamente en la clase PolyBench como atributos de clase. Por otra parte, el objeto *PolyBenchSpec* contiene información específica del benchmark como el tipo de datos con el que trabaja, los tamaños de las estructuras de datos, etc. Este objeto lo utilizan principalmente los benchmarks, pero el método *__init__()* lo utiliza para inicializar el formato de impresión de datos en función del tipo de dato.

Finalmente el método de inicialización contiene también el código para las librerías *inline* e *inlineasm* visto en las sección 4.3.4 para la gestión del planificador Linux en intérpretes diferentes a CPython y la sección 4.3.2 referente a la implementación del TSC.

A continuación se muestra un extracto del método para ilustrar los conceptos descritos. Las secciones de código C y ensamblador no se incluyen al haberse visto en las secciones 4.3.2 y 4.3.4.

```
class PolyBench:

    def __init__(self, options: PolyBenchOptions, parameters:
    PolyBenchSpec):
        # Validate instantiation
        if self.__class__ != PolyBench and
    issubclass(self.__class__, PolyBench):
            # Validate input parameters
            ...
            # Setup parameters and options
            self.DATA_TYPE = parameters.DataType
            ...
            self.POLYBENCH_TIME = options.POLYBENCH_TIME
            ...
            # Define C and Assembly functions
            ...
        else:
```

```
16              raise RuntimeError('Abstract classes cannot be
    instantiated.')
```

Métodos auxiliares de impresión

Los métodos auxiliares de impresión permiten imprimir información en el fichero de salida especificado por la opción de configuración *POLYBENCH_DUMP_TARGET* (stdout, stderr, un fichero de texto en el sistema de ficheros, etc.) siempre y cuando el usuario habilite la impresión mediante la opción *POLYBENCH_DUMP_ARRAYS*.

El objetivo de estos métodos es alterar el comportamiento del método *print()* para que se comporte de un modo parecido a la función *fprintf()* de C, a la vez que se ofrecen métodos específicos para imprimir los datos con el formato apropiado según si tipo, simplificando un poco el desarrollo del método *print_array_custom()* en los benchmarks.

Para ello, se desarrolla el método *print_message()*, que abstrae la función *print()* aplicando parámetros para que se comporte como la función *fprintf()* de C y aplicando el fichero de salida apropiado. El código de este método se ilustra a continuación.

```
1 def print_message(self, *args, **kwargs):
2     print(*args, file=self.POLYBENCH_DUMP_TARGET, end='', **kwargs)
```

Al rededor de este método se desarrollan los métodos de impresión de utilidad *print_array()* y *print_value()*. El método *print_array()* implementa la impresión de datos por pantalla añadiendo información relevante para igualar el formato de salida de impresión al utilizado en PolyBench/C y llama al método específico del benchmark *print_array_custom()*, permitiendo al benchmark decidir cómo se imprimen los datos específicos. Este método se usa internamente en el método *run()* de la clase PolyBench y permite imprimir arrays tanto con el formato predeterminado de la función *print()* como con el formato especificado por el benchmark en el método *print_array_custom()*.

El método *print_value()* ofrece una abstracción sobre el método *print_message()* que permite imprimir un único dato del array del benchmark utilizando un formateador de datos adaptado al tipo de dato del benchmark. Este método está pensado para ser utilizado directamente en el código específico del método *print_array_custom()*. El formateador de datos viene representado por una cadena de texto que se inicializa en el método *__init__()* a partir del tipo de dato utilizado en el benchmark y se almacena en el atributo de clase *self.DATA_PRINT_MODIFIER*. El formato para esta cadena es *"{:d}"* para números enteros, mientras que para números en punto flotante es *"{:.2f}"*, en dónde ".2" indica al formateador que debe redondear los datos para que queden dos decimales de precisión. El valor de *DATA_PRINT_MODIFIER* puede ser modificado en el código de inicialización de un benchmark en caso de querer aplicar un formato diferente (por ejemplo, usar más decimales de precision).

A continuación se muestran los códigos de los métodos descritos.

```
def print_array(self, array: list, native_style: bool = True,
    dump_message: str = ''):
    self.print_message(f'begin dump: {dump_message}')
    if native_style:
        print(array)
    else:
        self.print_array_custom(array, dump_message)
    self.print_message(f'\nend   dump: {dump_message}\n')

def print_value(self, value):
    self.print_message(self.DATA_PRINT_MODIFIER.format(value))
```

Ejecución de benchmarks

La ejecución de un benchmark se gestiona a través del método *run()*. Este método es responsable de inicializar la ejecución de un benchmark a través del método *run_benchmark()* y una vez finalizada se encarga de gestionar los datos de salida. Por una parte gestiona los arrays de datos devueltos por *run_benchmark()* para imprimirlos y por otra parte gestiona los resultados de las métricas observadas durante la ejecución del benchmark, devolviéndolas como un valor de retorno de tipo *dict* para que el usuario pueda trabajar con estos valores del modo que estime oportuno. En caso de que ninguna de las opciones *POLYBENCH_TIME* ni *POLYBENCH_PAPI* se encuentren activas el método devolverá un diccionario vacío.

El código del método *run()* se muestra a continuación.

```
def run(self) -> dict:
    outputs = self.run_benchmark()

    self.__print_instruments()
    if self.POLYBENCH_DUMP_ARRAYS:
        self.print_message('==BEGIN DUMP_ARRAYS==\n')
        for out in outputs:
            self.print_array(out[1], False, out[0])
        self.print_message('==END   DUMP_ARRAYS==\n')
        self.POLYBENCH_DUMP_TARGET.flush()

    if self.POLYBENCH_TIME:
        # Return execution time
        return {"POLYBENCH_TIME": self.polybench_result}
    if self.POLYBENCH_PAPI:
        # Return PAPI counters
        return {"POLYBENCH_PAPI": self.polybench_result}
    return {}
```

4.4 Clases auxiliares

Las clases auxiliares se utilizan para agrupar y definir las distintas opciones de configuración con las que trabajan los benchmarks. En el apartado 4.3 se han mencionado las clases auxiliares *PolyBenchOptions* y *PolyBenchSpec*.

La clase PolyBenchOptions es una clase contenedor en la que se definen, mediante atributos de clase, todas las opciones disponibles de PolyBench facilitando su uso en la clase abstracta. Esta clase establece las opciones predeterminadas para cada una de las opciones disponibles.

La clase PolyBenchSpec es una clase de utilidad que contiene los datos relevantes para inicializar un benchmark. Estos datos vienen definidos en el fichero *polybench.spec* para cada benchmark, representando cada línea del fichero a un benchmark diferente. De este modo, una instancia de esta clase representaría a una línea del fichero.

Existen también una serie de enumerados que simplifican el trabajo con distintos elementos de PolyBench. Actualmente las estructuras de tipo enumerado definidas son *DataSetSize* que define los tamaños de los conjuntos de datos del parámetro definido en *PolyBenchSpec*, y el enumerado *ArrayImplementation* que contiene las distintas posibilidades de implementación de arrays.

4.5 Desarrollo de benchmarks

En la sección 4.2.2 se mostró un sencillo ejemplo de desarrollo de un benchmark para aplicar los conceptos de la clase abstracta. En esta sección se pretende proponer e ilustrar con un ejemplo más complejo la implementación de un benchmark aplicando los patrones de diseño factoría y estrategia que permiten realizar implementaciones alternativas de forma transparente para el usuario. El objetivo de las implementaciones alternativas es ofrecer la posibilidad de comparar el rendimiento de las opciones disponibles en Python para trabajar con arrays. Para ilustrar todo esto se utiliza el código de *Jacobi-2D* aprovechando que su implementación es similar a la versión estudiada, con la diferencia de que ahora se trata con datos bidimensionales.

La idea general para aplicar el patrón de diseño consiste en implementar una factoría en el constructor de la clase del benchmark de tal modo que esta factoría devuelva, en el momento de la creación de la clase, una instancia de una subclase del benchmark que implemente el algoritmo con el tipo de array especificado en las opciones de usuario. El siguiente fragmento de código ilustra la implementación definida.

```
class Jacobi_2d(PolyBench):

    def __new__(cls, options: PolyBenchOptions, parameters:
    PolyBenchSpec):
        implementation = options.POLYBENCH_ARRAY_IMPLEMENTATION
        if implementation == ArrayImplementation.LIST:
            return _StrategyList.__new__(_StrategyList, options,
    parameters)
        elif implementation == ArrayImplementation.LIST_FLATTENED:
            return
    _StrategyListFlattened.__new__(_StrategyListFlattened, options,
    parameters)
        elif implementation == ArrayImplementation.NUMPY:
            return _StrategyNumPy.__new__(_StrategyNumPy, options,
    parameters)
```

El método *__new__()* en Python es un método especial que define el constructor de una clase. En la implementación de un benchmark es importante definir explícitamente este método para aplicar el patrón de diseño y es necesario que la firma del método contenga a los mismos parámetros que se vayan a utilizar en el método de inicialización *__init__()*. Aprovechando el parámetro *PolyBenchOptions* se puede consultar el tipo de implementación para los arrays y a partir de ella seleccionar la estrategia apropiada. Nótese que la forma de crear una instancia de una estrategia es a través del constructor de dicha estrategia ya que lo que se pretende cuando se instancia una clase *Jacobi_2D* es devolver la subclase que implemente la estrategia apropiada.

Continuando con la implementación de la clase *Jacobi_2D*, el método de inicialización *__init__()* debe ser implementado de forma en que se encargue de seleccionar los parámetros *PolyBenchSpec* que definen al benchmark, como el tamaño de los arrays de datos. De este modo el método queda definido como se ilustra a continuación.

```
def __init__(self, options: PolyBenchOptions, parameters:
    PolyBenchSpec):
    super().__init__(options, parameters)

    # Get the size of the parameters
    params = parameters.DataSets.get(self.DATASET_SIZE)

    # Set up problem size from the given parameters
    self.TSTEPS = params.get('TSTEPS')
    self.N = params.get('N')
```

Otro método que puede implementarse en la clase es *run_benchmark()*. En él deben inicializarse los datos y realizar la llamada al método *time_kernel()* para comenzar la ejecución

del benchmark. El código de este método es parecido al de la función *main()* de PolyBench/C aplicando las diferencias específicas de PolyBench/Python. Su código se muestra a continuación.

```
def run_benchmark(self):
    # Create data structures (arrays, auxiliary variables, etc.)
    A = self.create_array(2, [self.N, self.N], self.DATA_TYPE(0))
    B = self.create_array(2, [self.N, self.N], self.DATA_TYPE(0))

    # Initialize data structures
    self.initialize_array(A, B)

    # Benchmark the kernel
    self.time_kernel(A, B)

    # Return computed data as a list of tuples ('name', value).
    return [('A', A)]
```

Se puede observar cómo en la llamada a *create_array()* en el último parámetro se fuerza una conversión de tipos del valor de inicialización. Esto es necesario para evitar que el comportamiento predeterminado inicialice los datos del array al tipo predeterminado *int*, que puede ser incorrecto en algoritmos que esperan datos de otro tipo y que a parte de poder emitir resultados incorrectos, puede causar una pérdida de rendimiento considerable en intérpretes como PyPy que optimizan el acceso a las listas siempre y cuando se inicialicen con tipos de datos optimizables y homogéneos (por ejemplo, *floats*).

El método print_array_custom() puede implementarse en esta clase siempre que su código sea lo suficientemente genérico como para ser universal entre implementaciones (como ocurre en *Jacobi-1D* al trabajar con datos unidimensionales). En este caso como los datos que se imprimen son multidimensionales este método debe implementarse en las subclases que implementan las distintas estrategias. Esto es cierto también para los métodos *initialize_array()* y *kernel()*.

La forma de implementar una estrategia es muy sencilla. Basta con crear una nueva clase de estrategia que herede directamente del benchmark en cuestión. En este caso, para crear una estrategia que implemente el algoritmo utilizando listas de listas como representación de arrays multidimensionales, basta con crear la clase *_StrategyList(Jacobi_2D)* que hereda del benchmark de Jacobi. En la implementación de la estrategia es importante redefinir los métodos *__new__()* para generar una instancia correcta de clase e *__init__()* para llamar al código de inicialización de la clase padre, propagando la inicialización al resto de clases padres. La estrategia es responsable también de implementar todos los métodos abstractos de la clase *PolyBench* que no se hayan implementado por el camino. En este caso se implementan los métodos previamente citados *print_array_custom()*, *initialize_array()* y *kernel()*. De este modo,

la implementación de la estrategia para las listas queda definida en el siguiente fragmento.

```
class _StrategyList(Jacobi_2d):

    def __new__(cls, options: PolyBenchOptions, parameters:
    PolyBenchSpec):
        return object.__new__(_StrategyList)

    def __init__(self, options: PolyBenchOptions, parameters:
    PolyBenchSpec):
        super().__init__(options, parameters)

    def initialize_array(self, A: list, B: list):
        for i in range(0, self.N):
            for j in range(0, self.N):
                A[i][j] = (self.DATA_TYPE(i) * (j + 2) + 2) / self.N
                B[i][j] = (self.DATA_TYPE(i) * (j + 3) + 3) / self.N

    def print_array_custom(self, A: list, name: str):
        for i in range(0, self.N):
            for j in range(0, self.N):
                if (i * self.N + j) % 20 == 0:
                    self.print_message('\n')
                self.print_value(A[i][j])

    def kernel(self, A: list, B: list):
# scop begin
        for t in range(0, self.TSTEPS):
            for i in range(1, self.N - 1):
                for j in range(1, self.N - 1):
                    B[i][j] = 0.2 * (A[i][j] + A[i][j-1] +
    A[i][1+j] + A[1+i][j] + A[i-1][j])

            for i in range(1, self.N - 1):
                for j in range(1, self.N - 1):
                    A[i][j] = 0.2 * (B[i][j] + B[i][j-1] +
    B[i][1+j] + B[1+i][j] + B[i-1][j])
# scop end
```

Como se puede observar la implementación basada en listas ofrece una sintaxis muy natural a la hora de trabajar con arrays multidimensionales, similar a la sintaxis de otros lenguajes como C.

A modo ilustrativo se muestra a continuación el método *initialize_array()* de las estrategias para listas aplanadas y NumPy para que se pueda apreciar que la única diferencia en la implementación se encuentra en la sintaxis del código y en las anotaciones de los tipos de

datos.

```
class _StrategyListFlattened(Jacobi_2d):

    def initialize_array(self, A: list, B: list):
        for i in range(0, self.N):
            for j in range(0, self.N):
                A[self.N * i + j] = (self.DATA_TYPE(i) * (j+2) + 2)
    / self.N
                B[self.N * i + j] = (self.DATA_TYPE(i) * (j+3) + 3)
    / self.N

class _StrategyNumPy(Jacobi_2d):

    def initialize_array(self, A: ndarray, B: ndarray):
        for i in range(0, self.N):
            for j in range(0, self.N):
                A[i, j] = (self.DATA_TYPE(i) * (j+2) + 2) / self.N
                B[i, j] = (self.DATA_TYPE(i) * (j+3) + 3) / self.N
```

La implementación basada en listas aplanadas tiene como objetivo reducir un problema multidimensional en otro unidimensional para poder comparar si existe alguna diferencia de rendimiento frente a la implementación basada en listas de listas.

Finalmente la implementación NumPy se realiza con el fin de comparar el rendimiento de las estructuras que ofrece Python frente a las que implementa la librería de alto rendimiento *NumPy*.

4.5.1 Observaciones

Aunque en general el proceso de implementación de benchmarks implique cierto esfuerzo (traducir código C a Python es casi trivial para la implementación de listas) existen detalles que han sido necesario tener en cuenta para la correcta implementación y verificación de los mismos. Por ejemplo en el caso de los benchmarks que utilizan funciones matemáticas en PolyBench/C los desarrolladores han tenido que seleccionar a través de macros la implementación apropiada en función del tipo de dato con el que trabaja el problema (recuérdese que este tipo de dato es configurable por el usuario a través del fichero *polybench.spec*). Por suerte en Python las funciones equivalentes a través de la librería *math* admiten distintos tipos de datos usando la misma firma. El problema de esta comodidad viene si se utilizan estas funciones con números enteros ya que Python convierte este tipo de dato en punto flotante para realizar la operación, causando una ligera degradación en el rendimiento del algoritmo.

En las funciones de PolyBench/C *init_array()* y *kernel()* se emplean valores paramétricos y escalares, en función de la opción *POLYBENCH_USE_SCALAR_LB*. Sin embargo como se co-

mentó en la sección 3.2.1 en Python no existen los valores escalares, por lo que únicamente se contemplan valores paramétricos. Tampoco tiene sentido pasar las dimensiones de los arrays de datos en las firmas de los métodos ya que estos valores se encuentran disponibles en los atributos de la clase.

4.5.2 Menciones especiales

La traducción de benchmarks en general es muy similar entre los 30 benchmarks disponibles en PolyBench. Sin embargo, algunos benchmarks presentan particularidades que merecen ser comentadas. Esta sección trata de comentar brevemente estos detalles.

linear-algebra/solvers/cholesky

Este algoritmo, junto con *lu* y *ludcmp* sacan a la luz una particularidad del operador módulo de C (%). En Python, el operador módulo define su rango de salida como [0, N). Sin embargo, en C el rango de salida es (-N, N). En estos benchmarks salta a la luz esta peculiaridad ya que utilizan el operador módulo con valores de entrada negativos que devuelven valores también negativos. Para solucionar este problema, basta con realizar, de modo específico para estos benchmarks, una negación al resultado del operador módulo.

linear-algebra/solvers/durbin

Este algoritmo, junto a otros, declara un array de pila dentro del código del kernel. La solución empleada ha sido utilizar la funcion de creación de arrays. Esto no debería ser problemático ya que el array se declara fuera del SCoP.

medley/deriche

Es el único benchmark que utiliza el tipo float dentro del kernel y por ello la validación de los resultados falla si no se aplican cambios. Basta con recompilar el benchmark de PolyBench/C cambiando el tipo de dato por double para que los resultados coincidan. Esto es una limitación por parte de Python al no implementar como tipo primitivo los flotantes de 32 bit.

medley/nussinov

Este benchmark utiliza dos macros, *match* y *max_score* que se han sustituido por su código equivalente en línea. La implementación se ve un poco más larga debido a ello.

En este benchmark también se define un tipo de dato *base* con el objetivo de definir un rango numérico entre [0, 3]. En la implementación Python simplemente se ignora este hecho y se delega en la implementación del código los valores que tome en dónde se usa, durante la inicialización de arrays en init y run.

stencils/fdtd-2d

La validación de datos no funciona de forma predeterminada por el formato de salida. Basta con modificar el fichero generado por PolyBench/C para mover la línea que incluye el texto *==END DUMP_ARRAYS==* al final del fichero, dejando una línea en blanco después del dump, para que la validación sea correcta.

stencils-seidel-2d

La validación falla por errores de precisión de una décima. Incrementando el número de decimales a la salida para ambas versiones (C y Python) los resultados coinciden.

4.6 Uso de PolyBench/Python

Para trabajar con PolyBench/Python se han desarrollado dos herramientas auxiliares. Una de ellas permite lanzar ejecuciones de benchmarks y la otra permite crear nuevos benchmarks a partir de plantillas de código. Los detalles de implementación de estas herramientas se describen en los anexos A.4.2 y A.4.1.

El objetivo de esta sección es simplemente ilustrar cómo se pueden emplear dichas herramientas.

4.6.1 Creación de nuevos benchmarks

La herramienta *create-benchmark.py* ofrece la posibilidad de crear nuevos benchmarks de forma muy sencilla. Para ello basta con especificar un nombre de benchmark y una categoría y la propia herramienta se encarga de realizar toda la gestión necesaria para que este benchmark esté disponible para su ejecución. La sintaxis de esta herramienta es muy sencilla y sólamente requiere un intérprete de Python para funcionar.

Para crear un nuevo benchmark, por ejemplo, *atax2*, basta con ejecutar el siguiente comando:

```
$> python create-benchmark.py --name atax2 --category
    linear-algebra/kernels/atax2
```

Tras la ejecución del mismo, en el directorio *benchmarks/linear-algebra/kernels/atax2* se habrá creado el fichero *atax2.py*. Este fichero contiene una plantilla de código en la que figuran los conceptos citados en la sección 4.5 del desarrollo de los benchmarks. Basta con modificar el contenido de los métodos con los detalles específicos de la implementación para tener un benchmark funcional.

4.6.2 Ejecución de benchmarks

La herramienta *run-benchmark.py* permite lanzar benchmarks de forma sencilla simplemente indicando la ruta relativa del mismo. Adicionalmente esta herramienta ofrece, mediante opciones configurables a través de línea de comandos, numerosas opciones de ejecución que permiten evaluar el benchmark de distintas maneras, así como gestionar los datos de salida del mismo.

Para poder consultar las opciones disponibles basta con utilizar la opción de línea de comandos *–help* para que la herramienta muestre un texto de ayuda descriptivo para cada opción disponible. Entre las opciones más destacadas se encuentran la posibilidad de activar y configurar las opciones de PolyBench, verificar los resultados de salida frente a otros resultados previamente computados, y la posibilidad de especificar cuántas veces repetir la ejecución completa de un benchmark para obtener múltiples resultados consecutivos.

A continuación se muestra un ejemplo de uso que permite lanzar el benchmark *Jacobi-1D* con la opción *POLYBENCH_TIME* y la implementación de arrays basada en listas aplanadas, repitiendo el benchmark cinco veces.

```
$> python run-benchmark.py benchmarks/stencils/jacobi_1d/jacobi_1d
    --polybench-options POLYBENCH_TIME --array-implementation 1
    --iterations 5
```

4.7 Planificación temporal y estimación de costes

Vista la estructura e implementación de PolyBench el siguiente paso consiste en detallar cómo se ha organizado la realización del trabajo para poder realizar una estimación del coste total del proyecto.

En la primera etapa de desarrollo se ha realizado un estudio del estado del arte, PolyBench/C, que se ha visto durante el desarrollo del capítulo 3. En esta etapa también se han estudiado las posibles soluciones a los problemas no triviales (como el manejo de funciones del sistema y acceso a los contadores PAPI) mediante la búsqueda y evaluación de librerías externas.

En una segunda etapa, se ha realizado el desarrollo completo de PolyBench, comenzando por la clase abstracta y, mediante un ciclo de desarrollo basado en diseño-implementación-validación, se implementaron los 30 benchmarks. En esta segunda etapa se han implementado también las herramientas auxiliares, permitiendo agilizar la implementación y validación de los benchmarks.

Como etapa final se realizan las pruebas de rendimiento para todos y cada uno de los benchmarks que permiten comparar empíricamente los resultados de las diferentes implementaciones existentes para las pruebas de PolyBench.

La figura 4.1 muestra el diagrama de Gantt perteneciente al desarrollo de este proyecto, dividido en las tres etapas descritas.

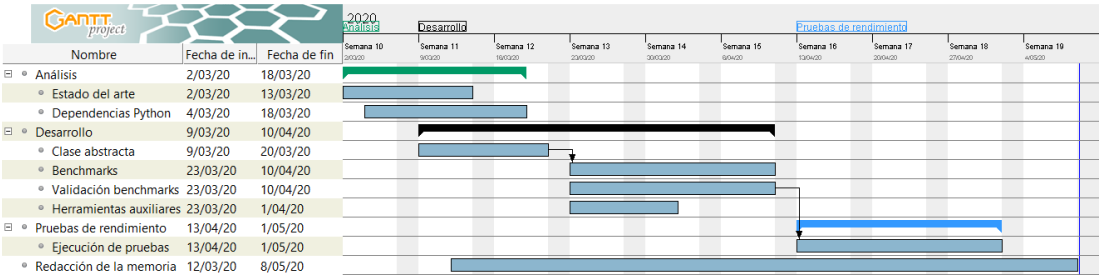

Figura 4.1: Diagrama de Gantt mostrando la planificación estimada

Para el análisis de costes es necesario tener en cuenta los recursos que han sido necesarios para el desarrollo del proyecto. Por una parte están los recursos humanos, en este caso el estudiante y el director de proyecto, y por otra parte estarían otros costes asociados. En este proyecto no es necesario contabilizar otros costes adicionales ya que se dispone de los recursos necesarios para el desarrollo completo del proyecto, por una parte el equipo de desarrollo (un ordenador personal) y por otra las herramientas software, que son herramientas libres y gratuitas.

En el diagrama de Gantt de la figura 4.1 se muestra una estimación temporal del trabajo que abarca un total de 10 semanas. El tiempo de trabajo se distribuye en 6 horas diarias durante 5 días a la semana, computando un total de 300 horas de trabajo. Este trabajo se distribuye entre los recursos humanos, de tal forma que al alumno le corresponden 240 horas y al director del proyecto 60. Para calcular el coste económico de este proyecto, se estima el coste por hora del alumno y del director del proyecto, siendo este de 10€ y 25€ respectivamente, teniendo en cuenta los costes sociales.

La tabla 4.2 muestra un resumen de esta estimación de costes y añade el cómputo total para poder estimar el coste total del proyecto.

Tabla 4.2: Coste total del proyecto a partir de los recursos necesarios

Recurso	**Horas**	**Coste/hora**	**Coste total**
Alumno	240	10 €	2400.00 €
Director	60	25 €	1500.00 €
		Coste total del proyecto	3900.00 €

Capítulo 5

Resultados experimentales

El objetivo final de PolyBench/Python es evaluar el rendimiento de los benchmarks desarrollados para poder comparar las medidas obtenidas entre las diferentes implementaciones con las medidas de referencia generadas en PolyBench/C.

Para comparar los resutados de las implementaciones se han elaborado una serie de tablas en las que figuran los benchmarks en cada fila y las distintas implementaciones en cada columna. El significado de los valores de la tabla viene definido en el título de cada tabla.

Por motivos de espacio disponible, los nombres de las diferentes implementaciones en Python se han abreviado. Los significados de las abreviaturas se listan a continuación.

- CPython-LL: representa al intérprete CPython con la implementación lista de listas.
- CPython-FL: representa al intérprete CPython con la implementación de lista aplanada.
- CPython-NP: representa al intérprete CPython con la implementación de NumPy.
- PyPy-LL: representa al intérprete PyPy con la implementación lista de listas.
- PyPy-FL: representa al intérprete PyPy con la implementación de lista aplanada.
- PyPy-NP: representa al intérprete PyPy con la implementación de NumPy.

Para la obtención de resultados los benchmarks se ejecutan en una máquina con las siguientes características: Procesador Intel Core i7 920 @ 3,37GHz, 24GiB DDR3@1600MHz. Sistema operativo Fedora 32, Linux Kernel 5.7.10-201. Intérprete de CPython versión 3.8.5. Intérprete de PyPy 7.3.1 (soporte Python 3.6.9).

Durante la ejecución de las pruebas se ha intentado reducir el número de servicios activos para reducir las interferencias que estos puedan causar durante la toma de muestras.

5.1 Muestreo temporal

La tabla 5.1 muestra los tiempos de ejecución promedio, en segundos, de cinco iteraciones consecutivas del benchmark. Para la obtención de los resultados se han utilizado los parámetros predeterminados de ejecución del benchmark (conjunto de datos “LARGE”, sin padding,

Tabla 5.1: Resultados de **POLYBENCH_TIME** en segundos utilizando **LARGE_DATASET** y promediando los resultados tras 5 ejecuciones.

Benchmark	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
correlation	12.2207	346.4770	570.4570	1007.0471	18.2919	12.3417	N/A
covariance	12.2575	687.9100	1135.7380	2007.8955	39.7629	25.1356	N/A
gemm	1.7388	414.4155	652.1741	1481.8701	10.9791	4.5210	N/A
gemver	0.0447	3.7042	4.8569	14.3411	0.1026	0.0719	130.5723
gesummv	0.0054	0.7911	0.9423	2.8759	0.0146	0.0086	22.9465
symm	9.1529	373.1414	567.0815	1078.0396	17.7286	9.7115	N/A
syrk	4.4593	242.2616	405.7015	810.2212	20.3897	16.6324	N/A
syr2k	10.2914	413.1509	701.0182	1541.2308	36.9615	27.0299	N/A
trmm	8.3369	210.6721	333.5399	591.5617	11.2302	8.3819	N/A
2mm	12.5310	548.1357	911.6014	1732.1234	24.8594	13.0922	N/A
3mm	20.5976	875.7078	1435.8113	2652.2360	38.3356	22.1256	N/A
atax	0.0090	1.8783	2.2093	8.4503	0.0787	0.0826	63.0289
bicg	0.0139	1.7211	2.2455	7.1180	0.0856	0.0809	60.1891
doitgen	0.7654	141.6387	273.4359	515.0038	4.8546	1.9190	N/A
mvt	0.0375	2.1306	2.5245	7.1846	0.0698	0.0503	66.2415
cholesky	1.4653	400.4689	644.6122	1358.6350	10.6981	4.2832	N/A
durbin	0.0040	1.0235	0.9794	3.3448	0.0260	0.0257	21.2337
gramschmidt	23.2378	511.3847	827.7921	1468.8693	31.6510	22.2101	N/A
lu	17.9113	862.0416	1411.4571	2679.3059	45.2575	20.6817	N/A
ludcmp	17.9189	669.4888	1055.6803	1845.5015	31.5057	19.8718	N/A
trisolv	0.0022	0.4821	0.5771	1.8112	0.0123	0.0116	16.4972
deriche	0.3871	24.1749	37.2235	75.4414	0.8891	0.5483	N/A
floyd-warshall	28.6126	8297.8912	16 436.8849	26 904.0758	330.8755	115.1629	N/A
nussinov	17.0496	1247.3076	2164.5455	3230.1243	50.7486	22.4432	N/A
adi	19.1327	1511.8442	2500.2806	4831.9528	41.2643	22.2600	N/A
fdtd-2d	3.2234	771.6370	1456.1075	2615.0186	22.3908	7.8038	N/A
heat-3d	5.8595	2455.1341	5780.0231	7615.4649	99.7752	24.3207	N/A
jacobi-1d	0.0024	0.5152	0.5172	2.1124	0.0087	0.0087	14.0553
jacobi-2d	3.6458	1147.1888	2015.0203	3421.7912	26.3812	11.8251	N/A
seidel-2d	27.8356	2697.3664	4306.8342	7650.9793	49.9749	28.0820	N/A

liberando 32MiB de caché) y utilizando el planificador FIFO de Linux. Se ha garantizado que el error promedio entre ejecuciones es menor al 5%. La línea de comandos para la compilación de las versiones C ha sido la siguiente:

```
$ gcc -O3 -lpapi -lm -DPOLYBENCH_PAPI -D[DATASET_SIZE]
    -DPOLYBENCH_LINUX_FIFO_SCHEDULER -lc <benchmark paths and files>
```

Para la ejecución en Python, la línea de comandos ha sido la siguiente:

```
$ python run-benchmark.py --polybench-options POLYBENCH_PAPI
    --array-implementation [0|1|2] --iterations 5 --dataset-size
    [SIZE] <benchmark>
```

De los resultados obtenidos se pueden observar varias características de comportamiento. La primera y más evidente es que los tiempos de ejecución de los algoritmos en Python es generalmente más lento que en C. Esto es lógico ya que Python se ejecuta detrás de un in-

térprete (CPython o PyPy en este experimento), no obstante, se pueden apreciar diferencias significativas entre ambos intérpretes.

Seleccionando un algoritmo cualquiera, como por ejemplo *atax*, se puede apreciar la gran diferencia de tiempo de ejecución entre la implementación de referencia en C, que es la más rápida de todas, frente a la más lenta obtenida con el intérprete PyPy y la librería NumPy. El tiempo promedio de ejecución del kernel en C ha sido de una fracción de segundo, 0.0090 segundos, mientras que en PyPy-NumPy ha sido superior a un minuto, 63.0289 segundos. Si a esto se suma que en el siguiente tiempo más lento se encuentra de nuevo la librería NumPy, esta vez con el intérprete CPython, con un total de 8.4503 segundos, se tiene una sorpresa inesperada. La librería NumPy es una librería utilizada para acelerar cálculos numéricos en Python, especialmente con datos representables en matrices. Sin embargo en estas pruebas particulares se ha obtenido una regresión en el rendimiento. La razón de este problema de rendimiento se encuentra en cómo se está usando la librería. Por simplicidad, se ha decidido implementar el código de los algoritmos basados en NumPy de forma parecida a las listas, accediendo a los elementos a través de subíndices (por ejemplo, *ARRAY[i, j]*) de forma en que la sintaxis sea clara. Sin embargo esta forma de utilizar los arrays de NumPy es inapropiada, ya que esta librería ofrece sus propios métodos de acceso a través de funciones como *item()* e *itemset()*, la primera para leer un valor y la segunda para escribirlo. En el caso de atax, una implementación en la que simplemente se cambien los accesos a través de subíndices por las llamadas a estas funciones ofrece un tiempo promedio total de ejecución para cinco ejecuciones de 4.4083 segundos con CPython y 23.2671 segundos con PyPy, reduciéndose los tiempos en ambos casos aproximadamente a la mitad a costa de un código fuente menos legible. Los resultados de NumPy se podrían mejorar aún más en caso de poder utilizar las funciones de manejo de matrices a costa de tener que reescribir los algoritmos de una forma vectorizada.

La columna *PyPy-NP* contiene múltiples elementos N/A (no aplica). En estos casos los test se omitieron manualmente por tardar un tiempo excesivo.

Continuando con el ejemplo de *atax*, se puede observar a través de las implementaciones basadas en listas Python la diferencia de rendimiento entre los intérpretes CPython y PyPy. En el caso de las implementaciones basadas en listas de listas, los tiempos promedio de CPython y PyPy fueron 1.8783 y 0.0787 segundos respectivamente. La diferencia de rendimiento es notable y se debe en gran parte a que el intérprete de CPython es un intérprete de código puro mientras que PyPy implementa un recompilador *just-in-time* que permite optimizar el código justo antes de ser ejecutado. Otra razón para esta diferencia de rendimiento se encuentra en el propio diseño de los intérpretes, siendo CPython un intérprete orientado a ofrecer la mayor funcionalidad posible mientras que PyPy pretende ser un intérprete de mayor rendimiento ofreciendo una serie de optimizaciones en su implementación orientadas al cálculo numérico.

Otra observación que se puede hacer con el benchmark de *atax* es la diferencia de rendimiento entre las implementaciones lista de listas y la lista aplanada. En un principio, el sentido común dice que la implementación basada en listas aplanadas debe ser más rápida, sin embargo esto no es así. De forma consistente entre todos los benchmarks la implementación basada en listas de listas es siempre más rápida que la implementación de listas aplanadas para el intérprete CPython (excepto para benchmarks con datos unidimensionales como *durbin* y *jacobi-1d* en dónde los resultados son idénticos para ambas implementaciones atendiendo al margen de error del 5%). La causa de esta "lentitud" en la implementación aplanada tiene que ver con el código generado para el intérprete en el que se realizan más operaciones en la versión aplanada. En la figura 5.1 se muestra la diferencia del código byte generado por CPython del siguiente fragmento de código. A la izquierda se muestra el código referente a la asignación del método *initialize_array()* (línea 5) mientras que a la derecha se muestra el código de la asignación en *initialize_array_flat()* (línea 10).

```
1     class X:
2     def initialize_array(self, A: list, x: list):
3         for i in range(0, self.M):
4             for j in range(0, self.N):
5                 A[i][j] = 0
6
7     def initialize_array_flat(self, A: list, x: list):
8         for i in range(0, self.M):
9             for j in range(0, self.N):
10                A[self.N * i + j] = 0
```

```
5           36 LOAD_CONST               1 (0)
            38 LOAD_FAST                1 (A)
            40 LOAD_FAST                3 (i)
            42 BINARY_SUBSCR
            44 LOAD_FAST                4 (j)
            46 STORE_SUBSCR
            48 JUMP_ABSOLUTE           32
      >>    50 POP_BLOCK
      >>    52 JUMP_ABSOLUTE           14
      >>    54 POP_BLOCK
      >>    56 LOAD_CONST               0 (None)
            58 RETURN_VALUE

sassembly of <code object initialize_array_flat at 0x559
8           0 SETUP_LOOP               60 (to 62)
```

```
92  10      36 LOAD_CONST               1 (0)
93          38 LOAD_FAST                1 (A)
94          40 LOAD_FAST                0 (self)
95          42 LOAD_ATTR                2 (N)
96          44 LOAD_FAST                3 (i)
97          46 BINARY_MULTIPLY
98          48 LOAD_FAST                4 (j)
99          50 BINARY_ADD
100         52 STORE_SUBSCR
101         54 JUMP_ABSOLUTE           32
102   >>    56 POP_BLOCK
103   >>    58 JUMP_ABSOLUTE           14
104   >>    60 POP_BLOCK
105   >>    62 LOAD_CONST               0 (None)
106         64 RETURN_VALUE
```

Figura 5.1: Diferencia de código generado entre lista de listas y lista aplanada

En la figura 5.1 se puede observar como el código de la versión aplanada a la derecha necesita realizar las cargas de atributos adicionales *self* y *N* (bytes 40 y 42 en la imagen a la derecha) y además requiere realizar una multiplicación (byte 46). Estas operaciones adicionales son la causa principal del deterioro de rendimiento en CPython.

En PyPy la situación se parece más a la esperada ya que los tiempos de ejecución de las versiones de lista aplanadas son habitualmente más rápidas que las versiones basadas en listas

de listas (salvo nuevamente en los benchmarks unidimensionales, en dónde los resultados no varían salvo por el margen de error). La ventaja de PyPy en este caso proviene de su implementación orientada al cálculo numérico combinada con la capacidad del recompilador JIT para optimizar esa región de código y evitar los accesos indirectos a los atributos.

Una observación curiosa sobre los resultados reside en los tiempos de ejecución de los benchmarks *trmm* y *gramschmidt*, para los cuales los tiempos de referencia en C y el tiempo obtenido en la implementación de lista aplanada de PyPy son idénticos en el caso de *trmm* y sorprendentemente más rápidos en PyPy para *gramschmidt*. La causa de esto no se debe a un error en los resultados ya que se ha repetido posteriormente esta prueba para estos benchmarks obteniendo resultados similares. La causa puede estar en la localización de los datos en memoria y la capacidad del recompilador JIT de PyPy para optimizar este código en concreto.

5.2 Muestreo de contadores PAPI

En la tabla 5.1 se puede observar que en general los tiempos de ejecución son demasiado altos. Por ejemplo, el benchmark *floyd-warshall*, que es el benchmark que más tiempo de ejecución requiere en general, ha necesitado 26 904.0758 segundos para completar su ejecución con CPython+NumPy (aproximadamente 7 horas y media). Esta situación es inapropiada para elaborar cualquier medición compleja, especialmente para las mediciones de contadores de rendimiento. Por esta razón, las tablas de rendimiento de contadores PAPI se elaboran utilizando tamaños del conjunto de datos adaptados al tiempo de ejecución de la implementación más lenta. De este modo, se ha decidido que la referencia temporal la define la combinación de intérprete-implementación PyPy+NumPy y el tamaño del conjunto de datos será aquel para el cual el tiempo total de ejecución de una iteración no supere los dos minutos, ya que para cada benchmark se van a monitorizar doce contadores y se ejecutarán un total de cinco veces cada uno de tal modo que el resultado que se muestre en las tablas será la mediana y no la media, ya que la mediana representa un valor real de contador. La mediana puede ser una mejor medida que la media en esta ocasión ya que algunos contadores, como los relacionados con los fallos de caché, se ven fuertemente afectados por la carga del sistema (algún proceso ajeno, interrupciones del sistema, etc.) incluso cuando se establece el nivel de prioridad FIFO para el proceso y esta carga puede generar valores en ciertos contadores anormalmente altos.

La tabla 5.2 muestra los resultados obtenidos para el contador *PAPI_TOT_CYC*, que mide el número total de ciclos de CPU que se han consumido durante la ejecución del algoritmo. Esta medida puede utilizarse para compararla con los tiempos de ejecución obtenidos en la tabla 5.1. En esta tabla de contadores de ciclos se sigue la misma tendencia que en la tabla de

Tabla 5.2: Resultados de **POLYBENCH_PAPI** para el contador **POLYBENCH_TOT_CYC**. Los resultados se expresan en millones de ciclos. Los algoritmos con * requieren la reinicialización de los valores de los arrays entre ejecuciones.

Benchmark	SIZE	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
correlation	MEDIUM	51.6345	7200.9103	12 687.6684	24 270.8350	514.9720	429.3966	213 973.2412
covariance	SMALL	1.1655	619.0143	1099.4830	2048.3860	19.4820	8.8624	18 792.8280
gemm	SMALL	0.5651	347.0238	545.0614	1239.0442	10.1131	4.5411	10 219.2287
gemver	MEDIUM	0.1567	456.9488	615.7481	1825.0441	10.0613	6.5780	13 594.4226
gesummv	MEDIUM	18.7021	2617.0362	3228.1864	9334.3721	45.6850	24.6381	64 908.5188
symm	MEDIUM	44.0960	8412.7835	13 840.4750	27 903.8523	297.7074	129.0503	248 041.4384
syrk	MEDIUM	14.1915	5915.1703	9711.1755	20 788.4682	183.3854	81.2296	187 400.9288
syr2k	SMALL	0.8427	333.4756	552.6690	1306.4289	10.2070	3.5058	10 943.0040
trmm	MEDIUM	21.4633	4481.8937	7976.5635	15 054.7210	182.6911	73.1741	143 137.2449
2mm	SMALL	1.0019	293.3906	485.6481	1030.8695	9.7208	4.3306	9038.9258
3mm	SMALL	1.7649	514.8754	922.5163	1708.8123	28.3240	20.7734	15 021.9303
atax	LARGE	21.9014	5802.0667	7619.4756	26 639.0851	257.7649	271.9102	159 409.0512
bicg	LARGE	46.5316	5856.2305	7598.8657	22 352.5763	283.2432	270.9967	152 016.6113
doitgen	MEDIUM	22.5376	6521.8547	12 567.6107	21 970.6100	229.2786	92.6841	170 640.4964
mvt	LARGE	125.9983	6972.7567	9133.9825	22 036.2599	227.6032	164.9377	166 815.4979
*cholesky	SMALL	1.0247	270.4117	476.8951	908.7173	8.7290	4.0843	8660.1047
durbin	LARGE	11.4110	3343.5040	3326.0153	10 231.9056	77.7626	77.9131	51 126.9440
*gramschmidt	SMALL	1.5332	379.7833	696.2365	1229.8797	14.0146	5.4849	11 738.0250
*lu	SMALL	1.9533	526.9421	962.2841	1818.8373	16.6593	7.6443	16 285.6236
*ludcmp	SMALL	1.7442	374.4085	638.6976	1267.2548	12.3468	7.1458	11 473.9682
trisolv	LARGE	7.8808	1452.3141	1929.3060	5602.4262	37.7121	34.9137	41 815.8160
deriche	MEDIUM	30.3071	2898.6987	4841.2224	9432.2978	102.0416	69.0492	79 541.9878
floyd-warshall	SMALL	30.3062	7039.2325	13 773.9895	22 044.1820	300.2980	104.3837	227 098.4801
nussinov	SMALL	4.3857	1380.6424	2553.3451	3840.3470	51.0241	23.1010	38 456.1205
adi	SMALL	14.1102	1277.3118	2114.2770	4063.5870	32.7894	17.8051	34 514.7081
fdtd-2d	SMALL	1.5269	795.0930	1494.9402	2597.2520	30.5991	15.5298	24 654.5292
heat-3d	SMALL	2.6725	2450.4821	5249.5304	6835.9918	95.4250	26.1789	45 768.8190
jacobi-1d	LARGE	6.3840	1808.4942	1778.3265	6449.0324	22.3534	22.3608	35 749.3203
jacobi-2d	SMALL	2.4793	1264.5380	2306.2996	3879.0928	33.0193	15.4090	35 509.2603
seidel-2d	SMALL	22.8598	2129.4407	3608.5307	6246.3337	51.3967	26.3323	53 981.9390

tiempos en la que la implementación de referencia es siempre la más rápida, seguida de las implementaciones de listas de PyPy y finalmente las implementaciones de listas de CPython, quedando las implementaciones basadas en NumPy como las más lentas.

El anexo A.5 muestra los resultados de ejecutar los 30 benchmarks utilizando una serie de contadores diferentes. La tendencia general observada en los resultados es la misma comentada hasta el momento, siendo la implementación C la más eficiente y la implementación PyPy+NumPy la menos eficiente actualmente. Una observación importante es que los intérpretes de Python actuales no optimizan el código de tal forma que se usen las unidades de procesamiento vectorial, mientras que la mayoría de los benchmarks se benefician del uso de las mismas como se puede ver para las implementaciones en C. Sería interesante repetir las ejecuciones en el momento en el que alguno de los intérpretes de Python diese soporte a este tipo de extensiones hardware y poder ver cuál es el speedup frente a las implementaciones actuales.

Capítulo 6

Conclusiones

A lo largo de este trabajo se han visto las distintas etapas del proceso de traducción de un proyecto software a otro lenguaje. Durante este proceso se han estudiado las distintas arquitecturas de las herramientas empleadas, concretamente el funcionamiento de los lenguajes de programación C y Python y de los problemas que pueden surgir al emplear distintos compiladores e intérpretes aunque sean de un mismo lenguaje.

Los resultados finales obtenidos en el proyecto han resultado satisfactorios. El objetivo principal de convertir las pruebas de PolyBench a Python se ha podido implementar y validar en su totalidad, pese a que durante el proceso surgieron una serie de problemas y limitaciones que se han podido solventar. Por una parte ha sido necesario el uso de librerías externas para poder acceder a funciones del sistema operativo ya que el intérprete PyPy no implementa por completo las interfaces de acceso al mismo. No obstante, se espera que en un futuro los desarrolladores de PyPy puedan implementar las características faltantes. Por otra parte, para el acceso al registro TSC del procesador ha sido necesario el uso de una librería externa que permitiese embedir código ensamblador en Python, añadiendo una nueva dependencia al proyecto. La solución alternativa en este caso no era una opción ya que implicaría tener que realizar acciones adicionales por parte del usuario y añadirían pasos innecesarios para alcanzar la misma meta.

Con respecto a la comparativa de los resultados de ejecución, estos no resultaron ser siempre los esperados. La librería NumPy para cálculo numérico ha mostrado ser poco eficiente al manejar datos de forma arbitraria a través de subíndices. No obstante, en su defensa, esta librería no se ha usado de la forma más apropiada ya que en un principio se espera que se utilicen sus funciones internas para trabajar con los datos.

Queda plantear como trabajo futuro una serie de mejoras o características deseables. La primera de ellas consiste en realizar para cada benchmark una serie de implementaciones alternativas utilizando la librería NumPy de formas más correctas, una implementación cambiando el acceso a los elementos mediante las funciones *item()* y *getitem()*, y otra implemen-

tación más correcta reescribiendo los benchmarks de forma vectorizada.

Otra mejora interesante consistiría en poder personalizar a través de las herramientas auxiliares los nombres y rutas para los ficheros *papi_counters.list* y *polybench.spec*, para los cuales actualmente es un requisito que se encuentren en la raíz del proyecto al tener sus rutas codificadas manualmente.

Finalmente queda planteada una notación de marcado para las partes de control estático que, en caso de que una futura herramienta lo requiera, deberá ser adaptada al formato requerido por la misma.

Apéndices

Apéndice A

Material adicional

En este capítulo se incluye información complementaria al trabajo que resulta interesante mencionar.

A.1 Obtención del código fuente PolyBench/C

Este apartado trata cómo obtener el código fuente del proyecto PolyBench/C para su análisis y estudio.

El primer paso para obtener el código fuente es dirigirse al sitio web en dónde se encuentra hospedado el código fuente. Para ello, basta con abrir un navegador web y dirigirse a la dirección https://sourceforge.net/projects/polybench/. Una vez en el sitio web, hacer click sobre el enlace de descarga que aparecerá en pantalla. Esto descargará el fichero **polybench-c-4.2.1-beta.tar.gz** al directorio de descargas seleccionado en el navegador. Una vez obtenido el fichero será necesario desempaquetarlo para poder trabajar con su contenido.

Alternativamente a la descarga directa del fichero empaquetado, es posible realizar una copia del proyecto desde el sistema de control de versiones. Para ello será necesario dirigirse a la página previamente citada y en lugar de hacer click en el botón de descarga se deberá hacer click en el botón "Code" para acceder al repositorio de código en dónde se mostrará un enlace de descarga y un explorador de código en línea. En el momento de realización de este trabajo el repositorio de software es de tipo Subversion.

A.1.1 Desempaquetado del fichero *polybench-c-4.2.1-beta.tar.gz*

Desempaquetado en sistemas Unix

Serán necesarias las herramientas *tar* y *gunzip* para poder desempaquetar el fichero. En caso de no estar instaladas en el sistema se deberá consultar la documentación del mismo para conocer cómo obtener dichas herramientas (típicamente mediante algún gestor de paquetes

como “apt” en Debian, “dnf” en Fedora o “zypper” en SUSE).

Para desempaquetar el fichero basta con dirigirse al directorio en donde se ha descargado previemante y ejecutar el siguiente comando:

```
tar -xf polybench-c-4.2.1-beta.tar.gz -C .
```

Este comando indica a *tar* que descomprima (-x) el fichero (-f <fichero>) en el directorio seleccionado (-C). En este caso se desempaqueta el fichero *polybench-c-4.2.1-beta.tar.gz* en el directorio actual “.” por conveniencia. Durante el proceso de extracción, el directorio “*polybench-c-4.2.1-beta*” será creado y en el deberá figurar un listado de ficheros similar al mostrado en la tabla 3.1.

A.1.2 Desempaquetado en sistemas Windows

Será necesaria una herramienta de extracción que entienda el formato de empaquetado *tar.gz* (tar+gzip). Una herramienta útil, de código abierto, gratis y que soporta este tipo de formatos es *7-zip* y se puede obtener en el siguiente enlace https://www.7-zip.org/.

Asumiendo la previa instalación de *7-zip* y de forma análoga al desempaquetado en sistemas Unix, será necesario dirigirse al directorio en dónde se encuentre el fichero descargado *polybench-c-4.2.1-beta.tar.gz* y en este caso, desde la interfaz gráfica de usuario se selecciona el fichero y se realiza click derecho en el para desplegar el menú contextual. En este menú deberá aparecer la opción *7-zip*, se selecciona para que aparezca el despliegue del submenú y en este último se selecciona la opción *Extraer aquí*. De esta forma se genera en el directorio actual el directorio “*polybench-c-4.2.1-beta*” en el cual deberá aparecer un listado de ficheros similar al mostrado en la tabla 3.1.

A.1.3 Obtención del código fuente mediante Subversion

Para descargar el código fuente mediante Subversion será necesario tener el software *Subversion*, que provee el comando *svn*. Dicha herramienta se puede obtener desde el sitio web oficial https://subversion.apache.org/.

Una vez instalado (y registrado en el entorno del sistema), basta con abrir una consola y ejecutar el siguiente comando:

```
svn checkout https://svn.code.sf.net/p/polybench/code/trunk
polybench-code
```

Esto descargará el contenido más actual del proyecto en el directorio “polybench-code”. Dentro de “bolybench-code”, el subdirectorio con el mismo nombre (polybench-code) contiene el proyecto PolyBench/C con el que se tratará en este trabajo.

Debido a que el código fuente obtenido de esta forma se encuentra, a fecha de este documento, en un estado de desarrollo más actual que la versión descargable, existen diferencias en el código fuente. La versión disponible para descarga directa se encuentra en la *revisión*

99 del repositorio. Los cambios existentes hasta el momento (*revisión 108*) son principalmente correcciones que no alteran el estudio del código.

A.2 Rendimiento de iteradores de listas Python

En este apartado se ilustran las diferencias de tiempo y uso de memoria para las distintas formas de recorrer una lista en Python. Los resultados de este experimento se realizaron con la herramienta IPython usando las opciones por defecto del entorno. El intérprete de Python utilizado es CPython 3.8.5. Para el cálculo de tiempo de ejecución se emplea el comando mágico de IPython *%%timeit* y para el uso de memoria se emplea la librería *tracemalloc*. En cualquiera de los casos los datos obtenidos son aproximados.

Los resultados del experimento se muestran en las tablas A.1 y A.2.

El siguiente fragmento de código importa la librería *tracemalloc* que permite obtener información sobre las reservas de memoria realizadas por el intérprete entre dos puntos de medición. En este fragmento también se declara una lista de un tamaño considerable para trabajar con ella.

```
import tracemalloc
lst = [x for x in range(1000000)]
```

Los siguientes fragmentos de código muestran el cálculo del tiempo promedio y del uso de memoria para el iterador de listas predeterminado.

```
%%timeit
r = 0
for item in lst:
    r += item
```

```
tracemalloc.start()
r = 0
for item in lst:
    r += item
current, peak = tracemalloc.get_traced_memory()
print(f"Current memory usage is {current / 10**6}MB; Peak was {peak
    / 10**6}MB")
tracemalloc.stop()
```

Los siguientes fragmentos de código muestran el cálculo del tiempo promedio y del uso de memoria para el iterador de listas *range()*.

```
%%timeit
r = 0
for x in range(500000):
    r += lst[x]
```

```
tracemalloc.start()
r = 0
for x in range(500000):
    r += lst[x]
current, peak = tracemalloc.get_traced_memory()
print(f"Current memory usage is {current / 10**6}MB; Peak was {peak
    / 10**6}MB")
tracemalloc.stop()
```

Los siguientes fragmentos de código muestran el cálculo del tiempo promedio y del uso de memoria para el iterador de listas mediante *slice*.

```
%%timeit
r = 0
for item in lst[:500000]:
    r += item
```

```
tracemalloc.start()
r = 0
for item in lst[:500000]:
    r += item
current, peak = tracemalloc.get_traced_memory()
print(f"Current memory usage is {current / 10**6}MB; Peak was {peak
    / 10**6}MB")
tracemalloc.stop()
```

Tabla A.1: Tiempo medio de ejecución

Implementación	media (ms)	Des. típica (µs)	Ejecuciones	Iteraciones
Iterador de lista	59.5	849	7	10
Rango	61.8	1863	7	10
Slice	36	316	7	10

Tabla A.2: Uso de memoria

Implementación	Memoria al finalizar (MiB)	Memoria máxima (MiB)
Iterador de lista	0.00107	0.011953
Rango	0.001098	0.011581
Slice	0.00107	4.001725

A.3 Jacobi-1D

En esta sección se muestra la implementación completa del benchmark *Jacobi-1D* implementado en Python.

```python
class Jacobi_1d(PolyBench):

    def __init__(self, options: dict, parameters:
    PolyBenchParameters):
        super().__init__(options)

        # The parameters hold the necessary information obtained
    from "polybench.spec"
        params = parameters.DataSets.get(self.DATASET_SIZE)

        # Set up problem size
        self.TSTEPS = params.get('TSTEPS')
        self.N = params.get('N')

    def initialize_array(self, A: list, B: list):
        for i in range(0, self.N):
            A[i] = (self.DATA_TYPE(i) + 2) / self.N
            B[i] = (self.DATA_TYPE(i) + 3) / self.N

    def print_array_custom(self, A: list, name: str):
        for i in range(0, self.N):
            if i % 20 == 0:
                self.print_message('\n')
            self.print_value(A[i])

    def kernel(self, A: list, B: list):
# scop begin
        for t in range(0, self.TSTEPS):
            for i in range(1, self.N - 1):
                B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1])

            for i in range(1, self.N - 1):
                A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1])
# scop end

    def run_benchmark(self):
        # Create data structures (arrays, auxiliary variables, etc.)
        A = self.create_array(1, [self.N], self.DATA_TYPE(0))
        B = self.create_array(1, [self.N], self.DATA_TYPE(0))

        # Initialize data structures
```

```
40          self.initialize_array(A, B)
41
42          # Benchmark the kernel
43          self.time_kernel(A, B)
44
45          # Return printable data as a list of tuples ('name', value).
46          return [('A', A)]
```

A.4 Herramientas auxiliares

A.4.1 Creador de benchmarks

El objetivo de esta herramienta es facilitar al usuario, más concretamente a los desarrolladores de benchmarks, el proceso de creación de nuevos benchmarks.

Durante el estudio de la herramienta de ejecución de benchmarks se ha detallado cómo funciona uno de sus componentes encargado de buscar de forma automática los distintos benchmarks dentro del paquete *benchmarks*. En ese apartado se mencionaban los requisitos para que el sistema funcionase correctamente, entre ellos, la necesidad de crear ficheros *__init__.py* auxiliares. Es fácil ver que si este proceso se realiza manualmente es probable cometer errores como olvidar incluir en el fichero adjunto al benchmark la importación del mismo. Además, a la hora de implementar la clase para un nuevo benchmark es posible olvidarse de implementar alguno de los métodos definidos como abstractos en la clase PolyBench. Por estos inconvenientes surge la necesidad de crear una herramienta que se encargue de automatizar esta tarea manual y repetitiva, permitiendo al desarrollador centrarse en la implementación del nuevo benchmark sin preocuparse de tener que respetar ciertas normas fáciles de olvidar.

Plantillas de código

Puesto que esta herramienta tiene un uso limitado, la opción más sencilla para implementarla es basarla en plantillas de código. Estas plantillas no son más que ficheros de código Python con un contenido predefinido. El objetivo de la herramienta pasa a ser determinar dónde copiar estos ficheros, copiarlos y modificarlos para adaptarlos al nuevo benchmark.

El primer paso entonces es crear las plantillas de código. Estas plantillas se sitúan en el directorio *util* con respecto a la raíz del proyecto. Las plantillas serán dos:

- Una plantilla para los ficheros *__init__.py* sin código, cuyo contenido será simplemente la cabecera de licencia predeterminada y un comentario explicativo.
- Una plantilla para subclases de PolyBench en donde figuren los métodos abstractos que se deben reimplementar y cualquier documentación y/o código fuente que pueda servir

de ayuda.

La plantilla para los ficheros *__init__.py* se define como se ha mencionado, un fichero sin código fuente, con comentarios y licencia. Esta plantilla se encuentra en *utilities/template-package-init.py* y será copiada directamente a todos los directorios que sea necesario crear. En caso de que el directorio sea el que contiene al módulo de implementación, esta plantilla será editada para contener la sentencia de importación del módulo.

La plantilla de benchmark se define con la importación de módulos requeridos para cualquier benchmark y la implementación parcial de los métodos con código de ejemplo de una subclase de PolyBench con el nombre genérico *TemplateClass*. Este nombre será modificado automáticamente durante el proceso de copia de la plantilla en su destino de acuerdo al nombre especificado por el usuario. Esta plantilla se encuentra en *utilities/template-benchmark.py*.

Durante el proceso de creación de un nuevo paquete y modificación del nombre del benchmark, se respetan las normas definidas en el PEP8[19] para el nombrado de paquetes, módulos y clases. De este modo, cualquier aparición de un guión en el nombre será reemplazado por una barra baja y, en el caso de benchmarks cuyos nombres comienzan por un número, se les añadirá como prefijo un guión bajo.

Implementación

Como se ha visto, esta herramienta debe, a partir de un nombre y una categoría especificados por el usuario, crear la estructura de ficheros y directorios necesaria para definir un nuevo benchmark respetando las normas de los paquetes Python.

Para alcanzar este objetivo, la herramienta debe realizar en primer lugar la petición de los parámetros de entrada del nuevo benchmark al usuario. Esto se realiza de forma sencilla a través de parámetros de línea de comandos. En este caso no es necesario pensar mucho y lo más apropiado parece definir dos comandos por separado, uno para el nombre del benchmark y otro para la categoría en donde se clasifica. De este modo se definen los parámetros `--name` y `--category` (con sus versiones abreviadas `-N` y `-C` respectivamente). La sintaxis del programa quedaría definida, para crear un benchmark en *benchmarks/linear-algebra/kernels/atax*, del siguiente modo:

```
$> python create-benchmark.py --name atax --category
    linear-algebra/kernels/atax
```

El comando anterior debe crear un módulo llamado *atax.py* que contenga la implementación de la clase *Atax* en el directorio *benchmarks/linear_algebra/kernels/atax*. Por el camino debe crear los ficheros *__init__.py* en todos los directorios que se generen y para el directorio fi-

nal, donde reside el módulo, modificar el fichero *__init__.py* para que incluya la directiva de importación de la clase *Atax*.

De este modo la implementación concreta se divide en dos partes. En una se delega la creación de una estructura de paquetes válida en la que se crean los directorios necesarios junto con los ficheros *__init__.py* sin ningún contenido. Esta parte es fácil de implementar a través de las librerías estándar de Python. Para crear la estructura de directorios basta con usar la clase *Path* de la librería *pathlib* [28]. Esta clase es capaz de, dada una estructura de directorios completa, crear todos los directorios que falten en el árbol. Por otra parte, crear los ficheros *__init__.py* es también sencillo. Basta con comprobar si existen y, en caso de que no existan, copiar la plantilla de código al directorio objetivo. La copia se realiza a través de la función *copy2()* de la librería *shutil* [29].

Lo único que falta en este momento es copiar la plantilla de código, modificarla para que tenga el nombre del benchmark correctamente capitalizado y modificar el fichero *__init__.py* situado junto al benchmark para que lo incluya. Para resolver este problema se opta por una solución sencilla. Ya que el tamaño de las plantillas es pequeño (unos pocos Kilo Bytes) se pueden cargar en memoria por completo y realizar las modificaciones necesarias en memoria para después escribir el fichero completo de vuelta al soporte de almacenamiento. Con este esquema ya no es necesario copiar el fichero *__init__.py* durante el proceso de preparación del paquete junto al módulo, de hecho, de copiarlo, supondría una ineficiencia ya que en este momento se debe modificar su contenido.

Modificar el contenido de las plantillas resulta, en este momento, una tarea muy sencilla. En el caso del fichero *__init__.py* basta con cargar el contenido de la plantilla en memoria y añadir la sentencia de importación al final de la misma. Una vez hecho esto, se escribe en el fichero de destino el contenido modificado. De forma similar, se carga el contenido de la plantilla *template-benchmark.py* en memoria como un string y, a través del método *replace()* de los strings, se reemplaza el nombre de la clase plantilla por el nombre de la clase de destino. Una vez finalizado, se escribe el contenido modificado en el fichero de destino.

La implementación completa de lo descrito se puede ver en el fichero *create-benchmark.py*, que se sitúa en el directorio raíz del proyecto PolyBench.

A.4.2 Lanzador de benchmarks

El objetivo de esta herramienta es implementar un lanzador de benchmarks universal (dentro del contexto PolyBench/Python) de modo que cualquier ejecución que necesite realizar un usuario sea a través de esta herramienta.

Para alcanzar este objetivo la herramienta debe:

- ser capaz de localizar el benchmark que el usuario desea ejecutar

- interpretar el contenido del fichero *polybench.spec*
- aceptar opciones de configuración que alteren el comportamiento de PolyBench/Python de forma similar a las vistas en PolyBench/C
- realizar múltiples ejecuciones de benchmarks y promediar el tiempo entre ejecuciones

Búsqueda de benchmarks

La localización automática de benchmarks se realiza mediante el uso de una herramienta interna de Python: la librería *pkgutil* [30]. Esta librería provee, entre otros, a un método muy conveniente llamado *walk_packages()*. Este método realiza una búsqueda de paquetes y módulos Python a partir de un directorio determinado por parámetro. Durante el proceso de búsqueda, este método también se encarga de cargar y ejecutar los ficheros *__init__.py* existentes. Es importante destacar que este método no detecta módulos que no tengan consigo en el sistema de ficheros al fichero *__init__.py*.

Por otra parte, ya que el objetivo final es automatizar la carga y ejecución de benchmarks, será necesario obtener una instancia del benchmark deseado. Para ello se explota una característica disponible desde Python 3 que consiste en que todos los objetos en Python pasan a tener un método privado llamado *__subclasses__()*. Este método devuelve una lista en la que cada elemento es una referencia débil a las subclases que extienden a la clase desde la que se invoca el método. Por ejemplo, en CPython 3.7, el tipo *bool* se implementa como una subclase del tipo *int*:

```
>>> int.__subclasses__()
[<class 'bool'>]
```

No obstante, el método *__subclasses__()* sólo devuelve aquellas subclasses que han sido ejecutadas por el intérprete. En el caso de PolyBench es necesario importar explícitamente todos los benchmarks presentes en los distintos módulos encontrados para que *PolyBench.__subclasses__()* devuelva los valores deseados. Para ello basta con incluir, en los ficheros *__init__.py* adyacentes a los módulos (ficheros *.py* también) una sentencia de importación que incluya específicamente al benchmark que acompaña. Por ejemplo, el fichero *__init__.py* adyacente al fichero *correlation.py* debe tener el siguiente contenido:

```
from benchmarks.datamining.correlation.correlation import
    Correlation
```

La estructura de ficheros y directorios dentro del paquete para que el sistema funcione quedaría de la siguiente forma (nótese que sólo se ilustra en profundidad la categoría datamining ya que el proceso para las otras categorías se realiza de forma similar):

```
benchmarks
└── __init__.py
```

```
├── polybench.py
└── datamining
    ├── __init__.py
    ├── correlation
    │   ├── __init__.py
    │   └── correlation.py
    └── covariance
        ├── __init__.py
        └── covariance.py
```

Todo este proceso puede resultar ineficiente, tanto para el desarrollador que tiene que crear para cada nuevo benchmark un fichero de inicio como para el intérprete que necesita recorrer el sistema de ficheros y guardar estructuras adicionales en memoria. Estos problemas se pueden agravar especialmente si se tiene un número elevado de implementaciones de benchmarks.

El código responsable de realizar esta búsqueda se encuentra en *benchmarks/__init__.py* e implementa una función llamada *__build_module_list__()*. Esta función devuelve una tupla con dos conjuntos. El primer conjunto es la lista de los módulos encontrados dentro del paquete *benchmarks* y el segundo conjunto es el listado de referencias débiles a las distintas implementaciones de PolyBench. Estas referencias débiles pueden usarse directamente para crear instancias de objetos como se verá más adelante. Es necesario que la localización de este módulo sea precisamente la vista ya que ello permite que exista una referencia válida a la referencia de paquetes *__path__*, que sólo existe dentro del contexto de los paquetes en Python.

Si se desease poner el cuerpo de la función *__build_module_list__()* directamente en el programa principal sería necesario modificar la forma de obtener el valor de *from_path* para que tuviese como valor el nombre del directorio raíz del proyecto.

En este momento, si se ejecutase el intérprete de Python para importar el paquete *benchmarks* lo que ocurriría es que se definiría la función vista y se podría usar del siguiente modo (a modo ilustrativo, conviene asumir que se ha implementado un benchmark en *benchmarks/datamining/correlation/correlation.py*):

```
polybench-python$> python
>>> import benchmarks
>>> modules, classes = __build_module_list__()
>>> print(modules)
['benchmarks.datamining.correlation.correlation']

>>> print(classes)
[<class
    'benchmarks.datamining.correlation.correlation.Correlation'>]
```

Un detalle final importante sobre el contenido real del fichero *__init__.py* del directorio *benchmarks* es que en el momento de carga del paquete se exportan dos referencias que son consecuencia directa de ejecutar *__build_module_list__()*. Estas referencias son *benchmark_modules*, que contiene la lista final de módulos y *benchmark_classes*, que contiene las referencias débiles de las subclases de PolyBench encontradas. Estas referencias serán accesibles siempre y cuando se importen directamente.

Procesamiento de línea de comandos

Otro objetivo del programa principal es admitir la entrada de parámetros de configuración por parte del usuario. Este objetivo se implementa de forma muy sencilla a través de la librería *argparse* [31]. El código en dónde se definen los parámetros se puede consultar en la función *parse_command_line* del fichero *run-benchmark.py*. Esta función se encarga de reasignar las opciones definidas en el diccionario *polybench_default_options* que se encuentra en el fichero *benchmarks/polybench_options.py*. Este diccionario, una vez procesados todos los parámetros de línea de comandos y reasignados los valores pertinentes, se pasa directamente al constructor del benchmark para que éste pueda establecer su estado interno en función de estos parámetros.

Dado que crear parámetros de línea de comandos es tan sencillo como asignar etiquetas textuales no resulta interesante mostrar el código de esta parte. Como detalle importante, el código del parser es responsable de hacer una validación de todas las opciones disponibles y de establecer en valores válidos todas las posibles combinaciones. Para los parámetros no especificados basta con seleccionar un valor predeterminado. Los valores que el usuario introduce como inválidos deben generar un error durante esta etapa y finalizar el programa.

Para decidir qué parametros son interesantes, su sintáxis y su comportamiento, es necesario reflexionar un poco e intentar pensar qué puede resultar útil y, en la medida de lo posible, cómodo para el usuario encargado de ejecutar benchmarks.

El comando mínimo, presuponiendo que los ficheros Python no se pueden ejecutar directamente en el sistema (esto no es necesariamente cierto), para ejecutar un programa Python es el siguiente:

```
1 $> python run-benchmark.py
```

Con tan sólo eso ya surge la primera pregunta: ¿qué debe hacer el programa en este caso? Una opción lógica y común a muchos programas de línea de comandos es mostrar la ayuda del programa. Otra opción también típica es realizar una acción predeterminada. Ambas opciones tienen sus ventajas e inconvenientes.

Si se opta por imprimir la ayuda del programa el usuario podrá tomar las decisiones apro-

piadas mediante una segunda invocación con los parámetros necesarios. Esto implicaría para el usuario realizar un mínimo de dos llamadas. Sin embargo, si se desconoce la sintaxis para invocar a un benchmark y los benchmarks existentes, el usuario tendrá que hacer una llamada adicional para conocer los benchmarks existentes (o buscarlos manualmente por otros medios).

Por otra parte, si se opta por realizar una acción predeterminada el problema surge en determinar qué acción realizar. Una buena opción puede parecer mostrar un listado con los benchmarks disponibles junto con un ejemplo de ejecución y, adicionamente, un texto que indique la existencia de un comando de ayuda del programa. En este caso el usuario podrá realizar, mediante una segunda llamada, la ejecución del benchmark que desee o invocar a la ayuda del programa para conocer más detalles. Sin embargo, dado que PolyBench tiene en este momento 30 pruebas únicas, la pantalla del usuario mostrará todas estas pruebas en una lista y puede afectar al flujo de trabajo al ocupar espacio en pantalla. El número mínimo de llamadas con esta opción resulta ser de dos llamadas también.

En el supuesto de que el usuario ya conozca la sintaxis del programa, puede resultar más útil mostrar el listado de pruebas. Sin embargo, para nuevos usuarios resulta más útil mostrar la ayuda. Las figuras A.1 y A.2 muestran los diagramas de flujo para un usuario nuevo y uno con experiencia respectivamente. De ellos se puede deducir que, en general, parece más útil optar por la opción de mostrar de forma predeterminada el listado de benchmarks ya que esta información parece más útil para un usuario que trabaje habitualmente con la herramienta. En los diagramas mostrados, las cajas denotan acciones del usuario y las flechas el conocimiento aportado por el programa hasta el momento.

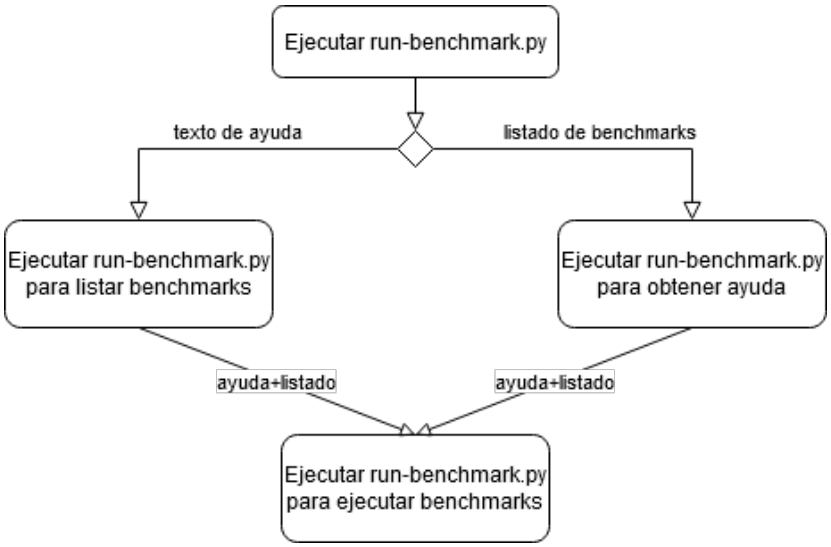

Figura A.1: Diagrama de flujo para la ejecución de run-benchmark.py por parte de un usuario nuevo

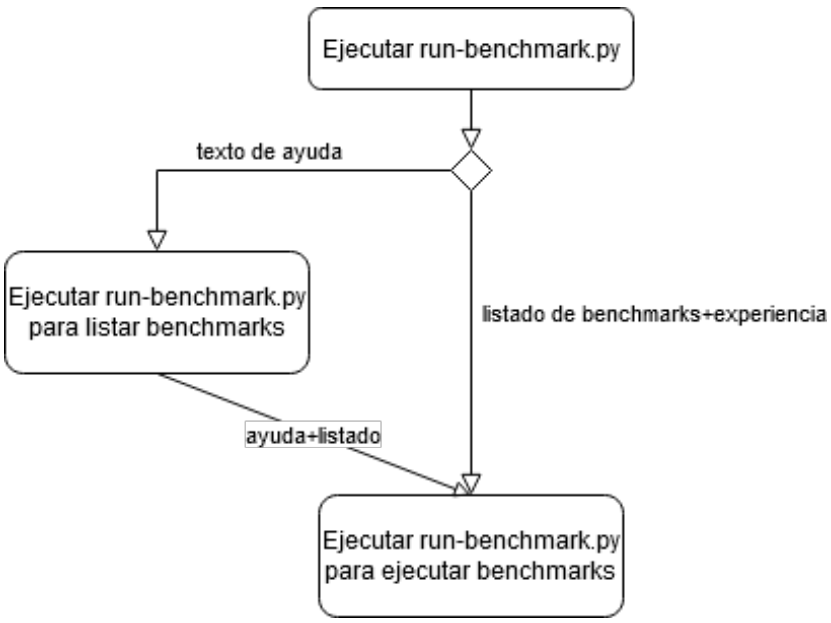

Figura A.2: Diagrama de flujo para la ejecución de run-benchmark.py por parte de un usuario con experiencia

Procesamiento del fichero polybench.spec

Otra de las responsabilidades del lanzador de benchmarks es el procesamiento del fichero *polybench.spec*. Este fichero contiene, para cada benchmark, los parámetros que lo definen, como el tipo de datos que utiliza, los parámetros de control del búcle (tanto su nombre como su valor) y una serie de conjuntos de valores predefinidos categorizados por tamaño.

El procesamiento de este fichero es importante para poder indicar de forma cómoda y rápida cambios en los parámetros que definen a un kernel dentro de un benchmark sin necesidad de alterar el código fuente.

Dado que la naturaleza de estos datos es diferente a las opciones de configuración del usuario, los datos obtenidos por este fichero se tratan por separado. Se debe pensar en el contenido de este fichero como "partes del código fuente modificables" y no como opciones de configuración propiamente dichas. Uno de los motivos para esta distinción es que el usuario puede, a través de las opciones de configuración, seleccionar qué tamaño del conjunto de datos definido en este fichero se va a emplear durante la ejecución del benchmark. En este caso es como si el benchmark ya tuviese implementadas todas las posibles opciones.

El contenido de este fichero, una vez procesado, se almacena en un diccionario para su posterior acceso.

Ejecución de programas

Hasta ahora se han visto las tareas necesarias que debe cumplir esta herramienta antes de poder ejecutar un benchmark. A continuación se detalla cómo utilizar la información previamente obtenida para ejecutar benchmarks concretos.

Antes de comenzar, es necesario expandir la clase PolyBench vista al principio del capítulo añadiendo un método de ejecución y una subclase que implemente dicho método. Dicho esto, la nueva definición de PolyBench pasa a ser:

```
class PolyBench:
    def __init__(self, options):
        ...  # Do stuff with options

    def run():
        raise NotImplementedError()
```

Por conveniencia, se van a definir dos benchmarks de la categoría *datamining*: *correlation* y *covariance*.

Fichero *benchmarks/datamining/correlation/__init__.py*

```
from benchmarks.datamining.correlation.correlation import
    Correlation
```

Fichero *benchmarks/datamining/correlation/correlation.py*

```
from benchmarks.polybench import PolyBench

class Correlation(PolyBench):
    def __init__(self, options, parameters):
        super().__init__(options)
        ...  # Do stuff with parameters

    def run():
        print('Imprimir: Correlation')
```

Fichero *benchmarks/datamining/covariance/__init__.py*

```
from benchmarks.datamining.covariance.covariance import Covariance
```

Fichero *benchmarks/datamining/covariance/covariance.py*

```
from benchmarks.polybench import PolyBench

class Covariance(PolyBench):
    def __init__(self, options, parameters):
        super().__init__(options)
        ...  # Do stuff with parameters

```

```
    def run():
        print('Imprimir: Covariance')
```

Nótese que las definiciones de las subclases reciben un parámetro adicional en el constructor. Este parámetro contiene los datos relativos al benchmark obtenidos en el fichero *polybench.spec*.

Estas nuevas definiciones permiten realizar dos cosas. Por una parte permiten al sistema de búsqueda automática encontrar los módulos *correlation* y *covariance* gracias a los ficheros *__init__.py*. Por otra parte, permiten probar de forma visual que el sistema de ejecución funciona ya que el objetivo es llamar al método *run()* y obtener por pantalla el nombre de la clase, tal cual se define dentro del método mediante la llamada a *print()*.

Para ilustrar más cómodamente la implementación de esta parte, se omiten detalles como el procesamiento de parámetros, excepto el parámetro *benchmark* que contiene el valor del benchmark a ejecutar. El siguiente fragmento de código se corresponde con una versión simplificada de la función *run()* de este programa, que es la encargada de llamar al método *run()* específico de cada benchmark:

```
def run(options: dict, parameters) -> None:
    module_name = options['benchmark']

    # Search the module within available implementations
    instance = None
    for implementation in benchmark_classes:
        if implementation.__module__ == module_name:
            # Create a new instance of the benchmark
            instance = implementation(options['polybench_options'],
    parameters)

    # Check if the module was found
    if instance is None:
        module = module_name.replace(".", "/") + '.py'
        raise NotImplementedError(f'Module {module} not
    implemented.')

    # When the module was found, run it.
    if isinstance(instance, PolyBench):
        instance.run()
```

Lo primero que se realiza es la asignación de *module_name*. El valor de esta referencia se corresponde con el parámetro especificado por el usuario para seleccionar el benchmark que se va a ejecutar tras ser procesado. Para este ejemplo se va a suponer que el valor de este parámetro es la cadena de texto *benchmarks.datamining.correlation.correlation*. Lo siguiente que se realiza en el código es crear una referencia para *instance*. El valor predeterminado

se establece en *None* para poder validar posteriormente si se ha encontrado el benchmark o no. El bucle que aparece a continuación recorre la lista de referencias a las clases que implementan los benchmarks. Esta lista se mencionó anteriormente en el apartado de búsqueda de benchmarks al principio de esta sección. Cuando se encuentra una referencia cuyo módulo se corresponde con el valor especificado por el usuario, se utiliza esta referencia para crear una instancia del objeto. Esto se puede ver en la línea 9 con el contenido `instance = implementation(...)`.

Para describir qué ocurre en este momento, es necesario recordar que en Python todo son referencias y que, en este instante del bucle, *implementation* es una referencia a la clase especificada por el usuario, en este caso, la clase *Correlation*. Para ser más correctos, las etiquetas *implementation* y *Correlation* son referencias a la clase interna de Python *<class 'benchmarks.datamining.correlation.correlation.Correlation'>*. Esto es fácil de comprobar desde el intérprete con la siguiente secuencia de comandos:

```
1  >>> from benchmarks.datamining.correlation.correlation import
       Correlation
2  >>> Correlation
3  <class 'benchmarks.datamining.correlation.correlation.Correlation'>
4
5  >>> etiqueta = Correlation
6  >>> etiqueta
7  <class 'benchmarks.datamining.correlation.correlation.Correlation'>
8
9  >>> instancia = etiqueta()
10 >>> instancia.run()
11 Imprimir: Correlation
```

De este modo, crear una instancia de clase es tan sencillo como invocar al constructor. Normalmente se realiza con la etiqueta predeterminada, pero en este caso se realiza con la etiqueta *implementation*.

Finalmente se comprueba si la instancia obtenida es una instancia de la clase abstracta PolyBench (una instancia de subclase es también instancia de la superclase) y, en caso de ser una instancia válida, se invoca al método *run()* para ejecutar el benchmark. En este caso concreto se imprime por pantalla el texto *Imprimir: Correlation* demostrando que la implementación de búsqueda y ejecución es correcta.

Validación de resultados

Una característica original de este lanzador de benchmarks es la posibilidad de comparar los resultados de la ejecución de un benchmark de PolyBench/Python con los resultados previamente guardados de un benchmark de PolyBench/C. La validación que se realiza es simple y susceptible a errores. Simplemente comorueba que el contenido del fichero generado

por PolyBench/Python coincide con el contenido del fichero guardado de PolyBench/C. Esta operaci.on puede fallar por muchos motivos como diferencias en el redondeo de decimales, diferencias en el formato de impresión, carácter de fin de línea, etc.

Esta función se implementa a través de *validate_benchmark_results()* y requiere que exista un fichero de resultados de PolyBench/C válido.

Promediado de tiempos

El lanzador de benchmarks es responsable también de realizar múltiples ejecuciones de un benchmark para promediar sus resultados como hace el script bash en PolyBench/C.

Esta operación se realiza dentro de *run()*.

A.5 Resultados de ejecución de contadores PAPI

En este apartado se incluyen los resultados de la ejecución de los 30 benchmarks utilizando los siguientes doce contadores PAPI y seleccionando la mediana del valor obtenido en cinco ejecuciones.

- PAPI_TOT_CYC: ciclos totales de ejecución
- PAPI_L1_DCH: aciertos en la caché de nivel 1
- PAPI_L1_DCM: fallos en la caché de nivel 1
- PAPI_L1_ICH: aciertos en la caché de instrucciones de nivel 1
- PAPI_L1_ICM: fallos en la caché de instrucciones de nivel 1
- PAPI_L2_TCM: fallos totales en la caché de nivel 2
- PAPI_L3_TCM: fallos totales en la caché de nivel 3
- PAPI_BR_MSP: instrucciones de ramificación condicional mal predichas
- PAPI_FP_OPS: operaciones de punto flotante
- PAPI_VEC_DP: instrucciones vectoriales de doble precisión
- PAPI_LD_INS: instrucciones de carga
- PAPI_SR_INS: instrucciones de almacenamiento

En general las tablas que se muestran a continuación tienen una serie de características que interesa comentar. Una de ellas es que independientemente del intérprete de Python, ningún benchmark ejecuta instrucciones vectoriales de doble precisión en Python. Esta puede ser una causa diferenciadora muy importante en la diferencia de rendimiento observado.

Con respecto al muestreo realizado para calcular la mediana, los resultados presentaron también características que deben ser comentadas. Por ejemplo, el contador *PAPI_L1_DCM* presentó de forma continua cierta variabilidad en sus resultados, al igual que los contadores *PAPI_L2_TCM* y *PAPI_L3_TCM*, siendo este último el que más variabilidad presentaba entre ejecuciones. Ya que estos dos últimos representan los fallos totales de la caché, es de suponer

que la variabilidad de los resultados es dependiente de la actividad del sistema. Por otra parte los contadores con la menor variabilidad (habitualmente presentando resultados iguales entre ejecuciones) son los contadores de instrucciones de carga y almacenamiento *PAPI_LD_INS* y *PAPI_SR_INS*. Este comportamiento es el más esperable y deseable para este tipo de instrucciones.

Tabla A.3: Resultados de **POLYBENCH_PAPI** para el benchmark **CORRELATION** y tamaño del conjunto de datos **MEDIUM**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	51 634 546	7 200 910 303	12 687 668 371	24 270 835 023	514 972 039	429 396 603	213 973 241 220
PAPI_L1_DCH	13 655 768	6 523 194 189	13 405 558 271	21 934 868 733	513 507 963	436 363 958	99 129 554 541
PAPI_L1_DCM	9 410 968	56 927 903	45 687 227	20 422 771	27 434 244	20 558 781	1 884 826 895
PAPI_L1_ICH	1 092 309	5 313 257 347	8 939 961 535	17 009 639 792	359 245 506	374 010 485	89 463 996 377
PAPI_L1_ICM	140	2 498 988	3 011 616	202 266 197	40 750	41 749	3 217 553 899
PAPI_L2_TCM	733 569	8 064 855	8 543 437	1 974 693	1 506 313	1 494 903	1 157 394 953
PAPI_L3_TCM	530	235 843	694 349	194 697	84 104	59 889	697 876 982
PAPI_BR_MSP	29 738	54 338 919	31 238 995	78 894 102	36 869	35 329	563 193 178
PAPI_FP_OPS	15 268 134	16 201 288	16 238 147	15 972 730	16 088 062	15 565 800	445 360 115
PAPI_VEC_DP	93 630	0	0	0	0	0	0
PAPI_LD_INS	15 194 723	4 717 193 874	9 414 329 353	15 477 134 748	409 865 784	333 863 006	68 325 312 856
PAPI_SR_INS	7 702 853	2 330 265 837	4 733 349 577	8 614 124 281	129 598 661	167 066 062	38 662 202 378

Tabla A.4: Resultados de **POLYBENCH_PAPI** para el benchmark **COVARIANCE** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	1 165 545	619 014 251	1 099 482 966	2 048 385 982	19 482 014	8 862 357	18 792 827 997
PAPI_L1_DCH	982 912	556 625 261	1 139 273 754	1 865 069 660	18 387 485	5 670 160	8 495 449 150
PAPI_L1_DCM	35 963	4 962 460	3 551 091	1 631 941	441 728	122 470	173 243 909
PAPI_L1_ICH	141 190	454 385 168	781 609 118	1 424 819 877	12 616 343	7 411 040	7 680 352 181
PAPI_L1_ICM	103	244 926	362 525	16 080 720	15 689	14 641	294 198 701
PAPI_L2_TCM	1154	471 798	451 239	114 406	17 845	14 096	116 125 473
PAPI_L3_TCM	1	3788	5865	2371	1186	426	69 664 960
PAPI_BR_MSP	3421	4 705 251	2 766 218	3 743 548	7702	7455	51 432 250
PAPI_FP_OPS	666 319	1 358 929	1 363 714	1 355 731	1 348 226	1 315 442	38 105 089
PAPI_VEC_DP	4000	0	0	0	0	0	0
PAPI_LD_INS	664 063	404 526 176	815 612 119	1 319 931 516	16 977 458	4 042 947	5 834 953 311
PAPI_SR_INS	345 918	200 400 250	404 032 493	735 949 331	2 957 808	3 013 877	3 298 931 922

Tabla A.5: Resultados de **POLYBENCH_PAPI** para el benchmark **GEMM** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	565 109	347 023 816	545 061 429	1 239 044 209	10 113 058	4 541 104	10 219 228 659
PAPI_L1_DCH	559 186	323 381 437	597 344 590	1 090 917 258	10 263 646	3 242 186	4 819 905 045
PAPI_L1_DCM	41 450	309 460	352 674	197 948	60 503	52 014	98 162 279
PAPI_L1_ICH	533 542	254 810 067	387 409 484	840 100 367	6 495 069	3 814 209	4 321 063 285
PAPI_L1_ICM	102	223 164	184 859	11 405 040	8736	9653	162 584 929
PAPI_L2_TCM	439	45 621	69 845	27 589	8363	6697	64 122 354
PAPI_L3_TCM	65	1523	2261	1837	716	696	37 001 851
PAPI_BR_MSP	82	2 583 049	809 036	4 333 006	5614	5525	27 752 021
PAPI_FP_OPS	342 922	1 042 326	1 045 109	1 046 204	1 023 096	1 013 244	21 973 659
PAPI_VEC_DP	338 120	0	0	0	0	0	0
PAPI_LD_INS	342 961	228 666 798	430 075 681	756 402 239	9 055 492	2 022 149	3 315 643 961
PAPI_SR_INS	170 138	115 095 584	219 137 848	425 803 453	1 876 148	1 576 384	1 876 467 565

Tabla A.6: Resultados de **POLYBENCH_PAPI** para el benchmark **GEMVER** y tamaño del conjunto de datos **MEDIUM**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	156 690	456 948 754	615 748 143	1 825 044 127	10 061 314	6 577 971	13 594 422 597
PAPI_L1_DCH	114 741	440 411 436	621 051 858	1 591 421 113	8 233 032	4 477 341	6 076 109 710
PAPI_L1_DCM	5466	2 038 178	1 883 441	731 871	400 976	220 005	118 147 009
PAPI_L1_ICH	42 165	337 723 925	428 970 112	1 241 334 224	7 111 772	2 717 357	5 425 065 301
PAPI_L1_ICM	125	184 643	200 355	11 221 919	15 303	17 151	211 605 823
PAPI_L2_TCM	508	502 142	419 296	93 647	37 203	34 963	79 440 611
PAPI_L3_TCM	4	106 631	125 038	22 668	1075	619	49 479 911
PAPI_BR_MSP	265	2 835 316	1 827 845	5 541 051	2565	2189	31 103 627
PAPI_FP_OPS	115 268	1 935 568	1 942 077	1 607 679	1 773 734	1 761 183	28 714 346
PAPI_VEC_DP	28 868	0	0	0	0	0	0
PAPI_LD_INS	79 873	312 749 181	445 113 025	1 119 156 722	7 305 939	3 472 425	4 262 871 202
PAPI_SR_INS	36 100	158 899 742	226 666 205	637 327 355	2 303 700	2 627 944	2 372 763 188

Tabla A.7: Resultados de **POLYBENCH_PAPI** para el benchmark **GESUMMV** y tamaño del conjunto de datos **MEDIUM**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	18 702 098	2 617 036 182	3 228 186 408	9 334 372 095	45 684 999	24 638 125	64 908 518 774
PAPI_L1_DCH	9 727 647	2 342 019 017	3 231 131 329	8 411 055 849	42 052 351	20 070 831	31 343 089 541
PAPI_L1_DCM	497 416	7 559 379	7 637 451	1 403 069	569 579	544 415	566 172 968
PAPI_L1_ICH	102 611	1 872 315 100	2 248 194 855	6 621 810 682	30 635 282	22 165 145	27 646 422 764
PAPI_L1_ICM	101	391 881	393 813	44 412 051	8702	7897	1 249 741 119
PAPI_L2_TCM	18 384	5 410 845	5 532 338	346 052	30 207	29 737	357 093 500
PAPI_L3_TCM	14 937	723 430	722 568	93 875	25 953	26 362	210 293 600
PAPI_BR_MSP	1835	20 224 820	16 928 642	26 927 833	2535	2516	155 869 533
PAPI_FP_OPS	6 772 350	6 790 129	6 773 172	6 771 833	6 766 430	6 772 749	141 588 494
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	10 141 368	1 622 114 112	2 311 487 391	5 945 709 441	38 991 977	17 028 951	21 984 833 094
PAPI_SR_INS	3 383 943	821 797 471	1 179 990 245	3 364 594 035	11 885 953	10 202 675	12 278 262 135

Tabla A.8: Resultados de **POLYBENCH_PAPI** para el benchmark **SYMM** y tamaño del conjunto de datos **MEDIUM**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PPAPI_TOT_CYC	44 095 955	8 412 783 474	13 840 475 031	27 903 852 256	297 707 416	129 050 346	248 041 438 367
PAPI_L1_DCH	16 856 860	7 366 714 738	14 297 849 374	24 520 467 609	229 942 193	74 713 880	112 095 957 474
PAPI_L1_DCM	7 025 821	45 057 733	33 273 305	14 005 388	17 973 167	9 362 887	2 389 772 346
PAPI_L1_ICH	9 264 737	6 308 195 348	9 794 848 752	19 335 609 335	170 658 443	98 372 094	102 228 620 189
PAPI_L1_ICM	233	3 167 956	4 989 778	177 796 043	25 421	18 570	4 030 638 229
PAPI_L2_TCM	1 188 549	13 573 885	13 525 381	2 455 635	1 231 199	1 194 740	1 577 186 851
PAPI_L3_TCM	889	1 517 488	2 278 131	367 699	66 105	34 009	934 457 826
PAPI_BR_MSP	18 721	83 165 827	46 126 946	63 569 610	36 345	36 304	706 939 804
PAPI_FP_OPS	12 657 706	24 549 420	24 523 153	24 554 537	24 962 479	24 259 899	523 033 338
PAPI_VEC_DP	7 238 202	0	0	0	0	0	0
PAPI_LD_INS	14 905 277	5 090 305 311	10 031 214 340	17 367 587 240	224 496 515	64 116 415	77 026 208 152
PAPI_SR_INS	4 825 856	2 556 043 761	5 099 355 883	9 724 770 069	31 357 293	36 228 935	43 501 832 798

Tabla A.9: Resultados de **POLYBENCH_PAPI** para el benchmark **SYRK** y tamaño del conjunto de datos **MEDIUM**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	14 191 515	5 915 170 334	9 711 175 543	20 788 468 205	183 385 392	81 229 559	187 400 928 773
PAPI_L1_DCH	10 820 160	5 532 893 355	10 326 314 519	18 382 517 353	168 011 817	57 634 768	82 667 266 040
PAPI_L1_DCM	1 001 951	27 266 751	19 943 091	7 775 178	6 235 873	1 201 326	1 670 553 657
PAPI_L1_ICH	3 990 068	4 455 651 554	6 793 153 906	14 140 632 086	113 442 873	68 030 435	75 540 603 667
PAPI_L1_ICM	162	1 687 214	2 746 229	192 861 008	13 135	12 357	2 672 373 888
PAPI_L2_TCM	330 048	4 320 274	4 374 908	1 247 253	522 052	493 346	1 128 773 912
PAPI_L3_TCM	19	153 367	319 353	107 237	39 010	22 219	706 514 890
PAPI_BR_MSP	23 384	46 091 121	12 971 153	39 781 254	40 667	38 288	509 027 987
PAPI_FP_OPS	5 901 209	17 721 869	17 730 798	17 700 831	17 676 338	17 399 003	377 341 650
PAPI_VEC_DP	5 779 983	0	0	0	0	0	0
PAPI_LD_INS	8 872 949	3 895 402 755	7 485 859 836	12 984 092 520	152 597 943	37 966 365	56 914 465 648
PAPI_SR_INS	2 919 966	1 960 335 722	3 798 247 161	7 292 552 021	30 593 686	25 377 642	32 114 337 485

Tabla A.10: Resultados de **POLYBENCH_PAPI** para el benchmark **SYR2K** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	842 685	333 475 616	552 668 963	1 306 428 895	10 206 983	3 505 811	10 943 004 041
PAPI_L1_DCH	621 322	307 950 437	588 238 074	1 115 329 187	10 285 138	2 875 308	4 912 040 367
PAPI_L1_DCM	41 852	1 500 607	1 148 958	567 965	144 166	56 879	104 482 498
PAPI_L1_ICH	655 579	248 182 296	398 543 719	878 074 987	6 421 760	2 940 430	4 454 235 714
PAPI_L1_ICM	115	186 026	171 053	9 145 866	10 642	10 939	204 766 524
PAPI_L2_TCM	569	240 817	284 635	28 721	7951	6643	63 998 032
PAPI_L3_TCM	2	2354	3848	1900	535	726	39 678 653
PAPI_BR_MSP	494	2 696 600	1 655 567	6 429 627	10 904	10 448	29 965 954
PAPI_FP_OPS	592 203	1 203 242	1 201 242	1 591 332	1 177 776	1 170 031	22 917 891
PAPI_VEC_DP	577 755	0	0	0	0	0	0
PAPI_LD_INS	557 445	216 161 557	416 397 049	772 800 467	9 013 530	2 057 235	3 385 181 319
PAPI_SR_INS	101 124	110 259 499	211 074 276	430 300 514	1 742 617	1 206 412	1 902 216 766

Tabla A.11: Resultados de **POLYBENCH_PAPI** para el benchmark **TRMM** y tamaño del conjunto de datos **MEDIUM**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	21 463 336	4 481 893 693	7 976 563 549	15 054 720 997	182 691 093	73 174 112	143 137 244 926
PAPI_L1_DCH	12 659 760	4 099 784 822	8 258 067 012	13 689 291 803	132 946 451	39 518 873	62 387 664 440
PAPI_L1_DCM	1 781 137	39 695 286	27 181 383	10 842 234	14 828 674	5 193 720	1 327 380 702
PAPI_L1_ICH	3 247 791	3 314 664 621	5 655 080 729	10 456 244 588	99 674 562	56 966 199	56 852 014 519
PAPI_L1_ICM	153	1 796 760	3 520 096	113 192 338	13 504	11 901	2 104 455 217
PAPI_L2_TCM	304 032	4 209 055	3 885 192	1 063 554	480 031	398 882	878 215 705
PAPI_L3_TCM	32	133 585	272 065	113 019	43 398	19 138	543 512 102
PAPI_BR_MSP	32 847	34 315 813	19 705 969	36 328 314	37 509	37 143	384 935 732
PAPI_FP_OPS	9 626 198	9 935 743	9 996 513	9 901 926	10 187 086	9 600 533	279 442 268
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	9 600 067	2 957 153 743	5 965 207 388	9 653 956 145	132 391 300	32 089 114	42 880 332 711
PAPI_SR_INS	4 824 045	1 464 763 261	2 989 547 036	5 384 230 573	30 253 532	21 037 187	24 267 700 371

Tabla A.12: Resultados de **POLYBENCH_PAPI** para el benchmark **2MM** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	1 001 898	293 390 596	485 648 106	1 030 869 488	9 720 817	4 330 583	9 038 925 835
PAPI_L1_DCH	903 728	275 660 194	514 117 080	912 842 466	9 143 741	2 780 834	4 106 256 593
PAPI_L1_DCM	7812	1 050 122	733 653	356 352	63 941	44 143	86 985 122
PAPI_L1_ICH	596 851	216 789 628	341 803 214	695 604 888	6 146 078	3 537 788	3 714 490 321
PAPI_L1_ICM	105	280 001	185 574	9 083 847	12 048	11 579	146 982 027
PAPI_L2_TCM	599	108 105	117 768	30 317	10 286	7329	56 689 650
PAPI_L3_TCM	55	3732	4530	1989	1354	888	35 994 811
PAPI_BR_MSP	2074	2 246 194	1 049 592	3 804 948	11 513	11 157	24 517 571
PAPI_FP_OPS	743 453	776 079	777 771	778 428	765 629	743 293	18 489 924
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	603 265	195 542 954	372 068 798	636 754 639	8 320 596	1 915 478	2 804 906 241
PAPI_SR_INS	305 238	97 662 690	188 413 115	358 562 514	1 838 040	1 434 383	1 587 572 082

Tabla A.13: Resultados de **POLYBENCH_PAPI** para el benchmark **3MM** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	1 764 949	514 875 397	922 516 317	1 708 812 308	28 323 971	20 773 350	15 021 930 252
PAPI_L1_DCH	1 594 962	464 832 343	927 512 716	1 545 797 473	29 629 351	23 093 917	7 060 395 211
PAPI_L1_DCM	38 159	2 147 134	1 464 385	596 253	197 897	146 724	138 330 822
PAPI_L1_ICH	973 315	375 913 663	661 168 740	1 177 054 940	19 480 293	18 146 009	6 408 310 136
PAPI_L1_ICM	94	392 759	651 904	13 453 864	17 310	17 440	252 579 252
PAPI_L2_TCM	679	199 512	236 786	59 116	24 103	19 394	85 714 798
PAPI_L3_TCM	11	8728	11 344	2823	3591	1878	53 003 976
PAPI_BR_MSP	3628	3 900 376	4 306 843	6 605 464	5910	6290	42 699 129
PAPI_FP_OPS	1 080 074	1 133 350	1 219 850	1 130 842	1 113 121	1 080 628	31 523 822
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	1 080 067	336 006 611	655 743 588	1 078 568 015	24 202 168	16 478 271	4 809 664 638
PAPI_SR_INS	548 339	166 327 397	331 011 646	608 213 360	6 556 067	7 424 185	2 740 109 794

Tabla A.14: Resultados de **POLYBENCH_PAPI** para el benchmark **ATAX** y tamaño del conjunto de datos **LARGE**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	21 901 356	5 802 066 725	7 619 475 588	26 639 085 098	257 764 861	271 910 170	159 409 051 175
PAPI_L1_DCH	19 598 763	5 875 063 770	8 203 208 959	23 821 815 945	229 743 697	227 918 878	75 508 998 279
PAPI_L1_DCM	1 652 683	12 245 463	12 184 009	3 443 757	1 892 902	1 844 799	1 222 432 190
PAPI_L1_ICH	202 228	4 317 647 567	5 542 829 559	18 151 973 384	200 469 642	216 076 864	65 829 593 853
PAPI_L1_ICM	145	1 697 290	2 690 170	225 697 605	14 817	16 945	3 439 566 492
PAPI_L2_TCM	42 918	2 412 778	2 272 900	982 434	45 625	59 121	844 410 724
PAPI_L3_TCM	24 555	305 042	312 929	93 284	37 630	50 177	508 295 136
PAPI_BR_MSP	4401	32 089 868	16 300 289	45 029 216	5351	5742	334 062 435
PAPI_FP_OPS	11 981 148	16 010 203	16 096 848	15 985 137	15 971 775	15 970 005	352 218 647
PAPI_VEC_DP	3 986 647	0	0	0	0	0	0
PAPI_LD_INS	11 977 669	4 128 606 259	5 852 064 101	16 608 673 852	195 769 959	183 803 679	53 209 715 863
PAPI_SR_INS	5 989 900	2 106 462 916	2 952 211 610	9 679 707 351	63 969 795	87 915 275	29 317 999 010

Tabla A.15: Resultados de **POLYBENCH_PAPI** para el benchmark **BICG** y tamaño del conjunto de datos **LARGE**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	46 531 554	5 856 230 542	7 598 865 693	22 352 576 260	283 243 241	270 996 749	152 016 611 319
PAPI_L1_DCH	27 012 118	5 538 226 670	7 772 818 380	19 930 370 651	309 343 126	321 358 255	74 391 404 324
PAPI_L1_DCM	1 061 579	8 285 194	8 419 341	2 846 906	1 497 788	1 491 306	1 348 706 650
PAPI_L1_ICH	165 041	4 284 629 460	5 453 843 980	15 942 678 867	199 773 163	212 100 022	65 536 870 163
PAPI_L1_ICM	128	1 165 865	2 194 484	103 459 850	13 589	12 948	2 954 006 264
PAPI_L2_TCM	21 987	1 702 265	1 676 177	704 880	45 122	41 914	809 001 028
PAPI_L3_TCM	18 684	338 911	363 920	97 095	36 748	45 212	502 070 635
PAPI_BR_MSP	2806	47 852 753	32 042 606	62 461 806	3719	3819	401 729 179
PAPI_FP_OPS	15 965 932	16 169 189	16 164 036	15 974 748	15 968 824	16 010 417	334 688 870
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	23 940 077	3 830 248 746	5 481 903 971	13 934 648 883	259 391 335	247 406 980	51 911 268 042
PAPI_SR_INS	7 983 104	1 945 878 078	2 791 638 129	7 947 063 327	83 855 519	115 768 706	28 960 435 841

Tabla A.16: Resultados de **POLYBENCH_PAPI** para el benchmark **DOITGEN** y tamaño del conjunto de datos **MEDIUM**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	22 537 589	6 521 854 727	12 567 610 740	21 970 610 032	229 278 594	92 684 090	170 640 496 375
PAPI_L1_DCH	21 775 216	6 255 843 846	12 718 446 014	19 740 944 150	215 104 751	64 484 724	77 350 833 451
PAPI_L1_DCM	84 173	23 753 202	17 704 536	7 198 196	1 884 519	849 541	1 535 955 767
PAPI_L1_ICH	21 961 938	4 814 124 624	8 987 106 761	15 469 394 672	140 251 105	81 676 250	69 841 133 385
PAPI_L1_ICM	121	8 498 781	8 670 611	128 711 328	29 760	16 540	2 919 908 600
PAPI_L2_TCM	2341	1 627 923	2 541 428	815 511	48 183	27 729	1 015 730 162
PAPI_L3_TCM	7	23 555	27 200	6402	14 787	7526	618 062 534
PAPI_BR_MSP	138	44 946 217	44 135 054	54 964 638	25 112	11 956	524 133 161
PAPI_FP_OPS	14 400 850	15 171 932	15 197 045	15 145 205	15 585 307	14 404 196	330 009 664
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	14 460 069	4 511 591 829	9 011 765 942	13 966 362 765	196 349 713	44 567 918	53 750 719 775
PAPI_SR_INS	7 380 044	2 234 718 667	4 468 964 324	7 798 524 907	40 776 160	34 425 817	30 034 426 943

Tabla A.17: Resultados de **POLYBENCH_PAPI** para el benchmark **MVT** y tamaño del conjunto de datos **LARGE**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	125 998 309	6 972 756 677	9 133 982 529	22 036 259 908	227 603 224	164 937 733	166 815 497 865
PAPI_L1_DCH	21 368 873	5 654 304 184	7 983 833 651	20 243 773 369	98 945 338	53 593 832	76 411 274 649
PAPI_L1_DCM	6 979 076	57 009 065	42 658 347	7 453 137	11 571 268	7 266 953	1 447 126 191
PAPI_L1_ICH	230 943	4 356 284 705	5 486 893 536	15 483 114 568	93 312 782	896 229	68 034 741 668
PAPI_L1_ICM	188	996 693	1 981 282	106 155 884	12 243	12 307	2 685 682 297
PAPI_L2_TCM	3 594 284	44 507 636	32 164 295	4 387 015	6 571 689	4 022 407	1 024 781 285
PAPI_L3_TCM	506 344	3 509 768	3 377 068	561 552	518 249	522 653	624 372 870
PAPI_BR_MSP	5422	32 246 207	16 194 761	39 820 413	6397	6481	398 644 173
PAPI_FP_OPS	16 020 256	16 762 775	16 260 679	16 024 331	16 003 867	16 005 564	347 791 629
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	16 004 100	4 136 600 925	5 816 363 382	14 174 214 469	88 218 324	40 239 393	53 045 113 145
PAPI_SR_INS	8 000 076	2 110 302 527	2 958 176 710	8 126 307 315	32 135 171	32 147 638	29 584 244 908

Tabla A.18: Resultados de **POLYBENCH_PAPI** para el benchmark **CHOLESKY** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	1 024 722	270 411 697	476 895 097	908 717 321	8 729 026	4 084 272	8 660 104 669
PAPI_L1_DCH	862 431	254 281 905	503 360 330	831 580 663	8 476 290	2 713 220	3 761 428 151
PAPI_L1_DCM	28 984	354 236	417 101	114 746	61 570	53 590	79 555 894
PAPI_L1_ICH	349 837	201 526 819	336 842 768	638 346 098	5 623 457	3 303 758	3 440 136 498
PAPI_L1_ICM	113	216 437	202 583	4 882 328	15 154	16 676	141 781 568
PAPI_L2_TCM	526	109 108	132 383	39 728	7586	8215	49 903 984
PAPI_L3_TCM	0	734	1169	964	310	928	30 720 486
PAPI_BR_MSP	7306	2 085 891	1 226 620	1 726 124	12 938	13 373	22 587 290
PAPI_FP_OPS	583 373	634 909	635 647	627 291	601 691	590 986	16 979 625
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	583 279	181 021 720	360 161 474	583 993 387	7 632 282	1 896 638	2 603 465 390
PAPI_SR_INS	295 278	89 821 573	182 897 737	329 220 174	1 390 572	1 437 543	1 473 904 579

Tabla A.19: Resultados de **POLYBENCH_PAPI** para el benchmark **DURBIN** y tamaño del conjunto de datos **LARGE**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	11 411 002	3 343 503 965	3 326 015 292	10 231 905 557	77 762 604	77 913 051	51 126 944 045
PAPI_L1_DCH	7 063 048	3 690 817 068	3 689 539 925	9 729 973 384	34 605 540	34 658 238	29 757 062 507
PAPI_L1_DCM	229 360	13 038 121	13 044 271	2 515 650	347 842	342 776	415 309 398
PAPI_L1_ICH	400 455	2 416 297 885	2 415 880 474	7 034 391 618	48 700 352	48 670 909	25 547 039 854
PAPI_L1_ICM	174	753 110	718 419	34 113 518	34 549	35 130	855 473 293
PAPI_L2_TCM	345	1 276 575	877 018	232 065	17 499	16 001	201 639 764
PAPI_L3_TCM	0	2448	2318	3147	1738	2227	100 752 425
PAPI_BR_MSP	5877	3 393 071	3 428 618	4 737 294	8313	8861	110 398 885
PAPI_FP_OPS	5 010 118	12 092 391	12 090 919	8 047 133	10 015 579	10 015 753	146 503 596
PAPI_VEC_DP	2 997 036	0	0	0	0	0	0
PAPI_LD_INS	5 019 042	2 639 993 353	2 641 060 666	6 832 904 596	18 321 128	18 328 518	21 105 853 427
PAPI_SR_INS	2 011 039	1 357 639 865	1 363 391 475	3 890 644 956	22 224 066	22 223 216	11 630 419 496

Tabla A.20: Resultados de **POLYBENCH_PAPI** para el benchmark **GRAMSCHMIDT** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	1 533 170	379 783 341	696 236 532	1 229 879 695	14 014 607	5 484 869	11 738 024 960
PAPI_L1_DCH	1 147 402	349 380 883	730 789 030	1 118 057 760	13 958 945	4 141 854	5 315 433 006
PAPI_L1_DCM	11 783	2 378 071	1 405 052	622 435	101 603	37 034	110 671 562
PAPI_L1_ICH	1 163 351	282 868 292	499 471 596	865 436 966	9 289 694	4 935 163	4 836 615 875
PAPI_L1_ICM	128	165 217	223 292	11 274 044	14 021	13 905	185 759 632
PAPI_L2_TCM	1029	206 265	199 140	51 864	11 726	9743	74 930 002
PAPI_L3_TCM	3	1995	2215	979	819	115	43 594 848
PAPI_BR_MSP	117	2 985 924	2 054 469	3 859 708	7078	2856	33 664 021
PAPI_FP_OPS	585 143	819 379	820 095	812 872	805 800	778 570	24 184 709
PAPI_VEC_DP	192 327	0	0	0	0	0	0
PAPI_LD_INS	774 621	249 452 633	522 470 879	785 978 376	12 334 588	2 920 930	3 645 100 424
PAPI_SR_INS	387 357	123 360 932	262 085 518	440 608 779	2 702 747	2 362 062	2 074 384 069

Tabla A.21: Resultados de **POLYBENCH_PAPI** para el benchmark **LU** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	1 953 282	526 942 107	962 284 076	1 818 837 260	16 659 288	7 644 264	16 285 623 641
PAPI_L1_DCH	1 675 868	495 112 049	987 203 055	1 639 688 428	16 472 047	5 742 089	7 432 392 178
PAPI_L1_DCM	63 503	1 907 596	1 545 505	311 059	108 571	101 811	150 854 441
PAPI_L1_ICH	739 690	395 831 664	668 072 676	1 276 731 122	10 754 451	6 801 694	6 740 103 192
PAPI_L1_ICM	92	293 857	458 009	9 514 370	13 401	14 031	245 433 042
PAPI_L2_TCM	620	395 119	453 958	57 853	10 408	9173	96 168 452
PAPI_L3_TCM	0	2594	8721	1812	1206	904	57 466 188
PAPI_BR_MSP	9494	4 085 277	2 445 229	4 468 957	16 177	15 896	44 725 248
PAPI_FP_OPS	1 145 146	1 240 553	1 241 216	1 230 746	1 169 852	1 152 121	33 301 591
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	1 159 239	354 758 490	707 737 204	1 144 417 489	14 873 634	4 133 092	5 135 899 706
PAPI_SR_INS	575 998	176 023 381	356 365 928	645 117 017	2 719 487	2 804 104	2 893 947 538

Tabla A.22: Resultados de **POLYBENCH_PAPI** para el benchmark **LUDCMP** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	1 744 226	374 408 464	638 697 589	1 267 254 761	12 346 758	7 145 846	11 473 968 183
PAPI_L1_DCH	849 083	357 108 133	682 505 807	1 177 544 358	12 311 227	4 829 838	5 307 841 344
PAPI_L1_DCM	65 885	1 945 907	1 512 533	328 514	117 328	110 429	112 414 962
PAPI_L1_ICH	1 210 167	280 440 639	457 541 769	906 725 337	9 288 038	3 609 952	4 853 427 257
PAPI_L1_ICM	108	136 772	314 848	5 384 887	21 462	23 530	152 430 678
PAPI_L2_TCM	734	426 325	414 063	35 836	12 152	11 835	74 393 607
PAPI_L3_TCM	3	1755	4070	1903	2219	1597	44 162 114
PAPI_BR_MSP	12 011	2 518 343	1 319 674	2 076 037	16 608	11 884	31 149 252
PAPI_FP_OPS	886 980	1 272 759	1 272 225	1 261 004	1 180 939	1 180 990	25 722 009
PAPI_VEC_DP	288 084	0	0	0	0	0	0
PAPI_LD_INS	893 167	256 285 346	495 284 474	823 225 966	10 736 979	3 216 201	3 662 725 047
PAPI_SR_INS	14 681	126 900 413	250 732 867	462 973 577	2 220 357	2 295 897	2 072 989 042

Tabla A.23: Resultados de **POLYBENCH_PAPI** para el benchmark **TRISOLV** y tamaño del conjunto de datos **LARGE**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	7 880 767	1 452 314 121	1 929 306 027	5 602 426 200	37 712 088	34 913 696	41 815 816 020
PAPI_L1_DCH	5 903 827	1 434 498 897	2 014 044 257	5 028 365 820	26 003 211	14 075 701	18 982 425 439
PAPI_L1_DCM	257 405	2 899 327	2 955 429	931 528	299 213	293 055	376 579 029
PAPI_L1_ICH	121 863	1 070 065 265	1 379 382 163	3 807 848 556	30 315 873	24 278 949	16 734 565 737
PAPI_L1_ICM	237	214 931	341 008	19 264 734	18 169	19 382	658 163 890
PAPI_L2_TCM	20 179	264 415	316 185	173 109	30 254	37 589	250 621 659
PAPI_L3_TCM	18 686	134 791	151 224	51 207	20 656	26 769	155 834 251
PAPI_BR_MSP	2388	8 046 202	4 055 026	9 195 516	3278	3452	92 742 431
PAPI_FP_OPS	4 005 631	4 019 466	4 020 666	4 012 413	4 004 992	4 004 636	87 000 865
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	4 002 061	1 025 944 262	1 458 184 330	3 540 695 129	22 176 616	10 210 369	13 261 919 276
PAPI_SR_INS	2 003 040	522 159 647	734 265 726	2 028 793 835	8 075 355	8 102 491	7 397 540 912

Tabla A.24: Resultados de **POLYBENCH_PAPI** para el benchmark **DERICHE** y tamaño del conjunto de datos **MEDIUM**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	30 307 138	2 898 698 684	4 841 222 397	9 432 297 814	102 041 635	69 049 223	79 541 987 834
PAPI_L1_DCH	3 510 219	2 562 712 362	4 569 774 115	8 218 961 638	77 990 980	27 922 779	35 483 218 429
PAPI_L1_DCM	1 779 295	9 682 532	8 042 804	3 760 945	4 212 366	2 740 623	766 659 831
PAPI_L1_ICH	243 833	2 063 243 171	3 299 459 487	6 534 234 472	51 585 296	25 219 389	32 462 803 056
PAPI_L1_ICM	176	470 542	1 229 235	73 275 231	39 806	40 568	1 266 476 808
PAPI_L2_TCM	559 895	5 411 388	4 832 684	1 942 455	1 378 572	1 218 229	491 950 377
PAPI_L3_TCM	8761	1 127 730	1 266 333	251 943	247 916	237 825	304 953 491
PAPI_BR_MSP	3998	21 341 838	20 669 481	20 161 495	8846	9303	223 071 165
PAPI_FP_OPS	10 912 188	12 487 005	12 518 253	12 472 515	11 145 470	11 151 848	171 956 474
PAPI_VEC_DP	172 980	0	0	0	0	0	0
PAPI_LD_INS	2 419 277	1 785 079 681	3 215 263 142	5 724 643 936	59 791 603	12 153 141	24 554 896 174
PAPI_SR_INS	1 555 247	907 868 013	1 640 127 888	3 190 165 557	17 805 273	18 169 308	13 774 405 241

Tabla A.25: Resultados de **POLYBENCH_PAPI** para el benchmark **FLOYD-WARSHALL** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	30 306 166	7 039 232 452	13 773 989 499	22 044 182 042	300 298 037	104 383 652	227 098 480 136
PAPI_L1_DCH	23 057 043	6 558 482 127	14 945 450 895	20 093 184 519	297 769 214	77 947 959	101 124 978 415
PAPI_L1_DCM	356 920	1 626 146	1 789 446	3 561 241	856 228	783 151	2 144 519 267
PAPI_L1_ICH	2 047 249	5 325 811 462	9 846 835 961	15 464 639 839	177 667 882	87 617 735	90 942 747 591
PAPI_L1_ICM	139	1 119 413	2 201 448	176 678 089	22 293	17 683	4 111 408 305
PAPI_L2_TCM	1119	308 397	334 987	509 877	231 056	109 521	1 346 966 504
PAPI_L3_TCM	0	2198	6455	9892	5491	3157	901 633 661
PAPI_BR_MSP	32 633	57 484 780	27 264 551	50 305 640	537 442	519 135	588 065 284
PAPI_FP_OPS	0	195 605	196 134	195 787	60	54	445 000 239
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	17 496 067	4 722 345 976	10 788 093 685	13 993 814 455	265 488 148	53 412 283	69 096 694 370
PAPI_SR_INS	5 832 047	2 211 069 022	5 359 848 454	7 824 461 692	26 354 379	30 014 171	39 599 877 136

Tabla A.26: Resultados de **POLYBENCH_PAPI** para el benchmark **NUSSINOV** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	4 385 724	1 380 642 371	2 553 345 114	3 840 346 980	51 024 125	23 100 988	38 456 120 492
PAPI_L1_DCH	3 005 455	1 224 851 669	2 634 115 719	3 456 130 042	45 145 416	10 280 104	16 960 299 802
PAPI_L1_DCM	52 800	1 574 554	727 614	1 274 517	266 041	162 278	364 734 949
PAPI_L1_ICH	1 024 618	1 056 732 150	1 841 614 789	2 694 557 331	26 589 976	12 567 322	15 257 552 956
PAPI_L1_ICM	113	347 806	414 932	33 282 659	21 520	20 883	693 310 670
PAPI_L2_TCM	698	73 176	108 535	212 294	27 001	20 139	229 396 286
PAPI_L3_TCM	0	1511	1758	3512	2549	1528	146 175 220
PAPI_BR_MSP	15 926	13 627 876	8 723 297	10 305 599	26 841	26 546	103 607 013
PAPI_FP_OPS	0	96 637	97 399	162 116	70	69	73 784 176
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	2 040 483	869 070 398	1 890 067 578	2 403 952 327	41 087 558	8 216 149	11 549 539 678
PAPI_SR_INS	1 004 229	403 343 818	925 289 415	1 333 497 112	4 412 095	4 494 454	6 613 201 298

Tabla A.27: Resultados de **POLYBENCH_PAPI** para el benchmark **ADI** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	14 110 174	1 277 311 839	2 114 277 036	4 063 586 955	32 789 447	17 805 082	34 514 708 110
PAPI_L1_DCH	2 002 106	1 178 045 452	2 147 281 587	3 485 968 383	31 175 333	7 948 006	15 313 035 908
PAPI_L1_DCM	142 072	3 565 758	3 208 864	1 023 246	324 972	174 353	333 683 569
PAPI_L1_ICH	2 597 313	946 000 585	1 509 338 798	2 680 317 750	21 122 609	12 758 298	13 955 001 982
PAPI_L1_ICM	199	3 376 855	5 023 806	30 774 435	27 761	20 297	701 227 310
PAPI_L2_TCM	875	694 777	684 448	191 267	41 925	17 011	213 243 407
PAPI_L3_TCM	1	892	1855	1572	1510	671	125 659 639
PAPI_BR_MSP	219	11 445 838	9 808 086	11 208 531	10 708	4165	101 306 373
PAPI_FP_OPS	3 711 313	5 816 266	5 448 489	5 170 936	4 710 574	4 586 827	74 396 315
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	1 888 797	806 072 685	1 501 479 061	2 422 765 204	27 688 046	6 749 076	10 545 463 566
PAPI_SR_INS	826 008	411 946 154	777 241 286	1 346 294 501	3 505 656	3 816 060	5 934 233 156

Tabla A.28: Resultados de **POLYBENCH_PAPI** para el benchmark **FDTD-2D** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	1 526 899	795 092 977	1 494 940 222	2 597 251 984	30 599 083	15 529 831	24 654 529 198
PAPI_L1_DCH	1 346 983	739 750 206	1 501 975 054	2 278 454 947	32 187 079	14 881 465	10 606 952 897
PAPI_L1_DCM	168 112	1 603 864	1 822 949	502 471	269 375	229 368	225 376 494
PAPI_L1_ICH	1 248 525	599 214 049	1 079 592 842	1 777 677 083	20 663 349	14 019 167	9 746 393 238
PAPI_L1_ICM	120	264 805	931 268	19 950 596	13 614	13 741	378 449 340
PAPI_L2_TCM	630	273 976	441 204	95 078	48 088	15 549	144 103 859
PAPI_L3_TCM	0	403	832	938	2704	439	93 746 475
PAPI_BR_MSP	63	6 822 433	8 056 911	9 101 416	12 214	8464	67 146 670
PAPI_FP_OPS	1 043 824	2 115 226	2 121 560	2 111 170	2 068 313	2 067 525	49 879 615
PAPI_VEC_DP	1 024 774	0	0	0	0	0	0
PAPI_LD_INS	1 043 425	513 884 778	1 053 915 244	1 585 147 314	27 170 452	10 450 393	7 302 927 025
PAPI_SR_INS	286 442	256 787 170	537 676 181	889 150 693	5 087 820	5 483 704	4 118 247 872

Tabla A.29: Resultados de **POLYBENCH_PAPI** para el benchmark **HEAT-3D** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
API_TOT_CYC	2 672 536	2 450 482 106	5 249 530 438	6 835 991 785	95 425 040	26 178 940	45 768 818 991
API_L1_DCH	2 041 174	2 252 087 440	4 880 428 656	6 079 175 062	88 867 823	17 744 733	22 944 230 763
API_L1_DCM	146 522	2 184 410	2 039 513	753 003	320 615	202 620	416 342 020
API_L1_ICH	1 796 744	1 824 519 963	3 747 732 501	4 766 943 650	49 567 221	21 067 585	20 440 545 964
API_L1_ICM	241	1 323 909	2 068 151	37 596 375	27 355	26 311	1 060 904 348
API_L2_TCM	683	368 190	406 388	213 231	72 629	23 155	209 368 284
API_L3_TCM	0	1884	3620	1861	2502	868	113 191 318
API_BR_MSP	57	23 041 196	38 292 934	22 700 280	10 148	8828	144 417 853
API_FP_OPS	3 036 711	7 182 704	7 172 434	7 163 452	7 012 633	7 006 584	100 797 138
API_VEC_DP	3 035 174	0	0	0	0	0	0
API_LD_INS	1 636 064	1 542 371 658	3 372 372 648	4 271 361 966	84 119 957	15 066 561	15 983 530 736
API_SR_INS	233 325	777 151 717	1 675 236 058	2 331 731 034	4 617 049	2 959 643	8 823 177 153

Tabla A.30: Resultados de **POLYBENCH_PAPI** para el benchmark **JACOBI-1D** y tamaño del conjunto de datos **LARGE**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	6 384 022	1 808 494 217	1 778 326 540	6 449 032 391	22 353 357	22 360 822	35 749 320 281
PAPI_L1_DCH	4 494 287	1 985 232 744	1 982 491 700	6 019 450 944	16 107 807	16 095 140	19 248 514 931
PAPI_L1_DCM	1520	2 858 774	2 838 061	802 959	180 448	169 208	309 387 096
PAPI_L1_ICH	64 488	1 294 871 669	1 297 533 316	4 609 772 071	22 077 222	22 079 719	16 849 030 394
PAPI_L1_ICM	115	212 724	218 804	18 065 851	12 688	12 868	555 500 413
PAPI_L2_TCM	216	238 733	220 250	158 636	6099	6312	145 368 718
PAPI_L3_TCM	1	594	755	1947	320	307	78 334 584
PAPI_BR_MSP	1024	4 078 432	4 130 512	13 990 309	1860	1893	88 134 262
PAPI_FP_OPS	2 997 599	6 027 371	6 026 440	6 000 222	5 994 298	5 994 197	96 071 897
PAPI_VEC_DP	2 997 576	0	0	0	0	0	0
PAPI_LD_INS	2 997 064	1 384 799 790	1 384 796 785	4 190 601 360	10 050 951	10 050 972	13 592 566 157
PAPI_SR_INS	999 041	732 536 970	732 536 965	2 390 878 321	8 033 259	8 033 264	7 483 178 347

Tabla A.31: Resultados de **POLYBENCH_PAPI** para el benchmark **JACOBI-2D** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	2 479 288	1 264 538 042	2 306 299 569	3 879 092 849	33 019 338	15 409 033	35 509 260 308
PAPI_L1_DCH	2 019 460	1 179 131 150	2 318 251 741	3 432 502 034	34 676 179	10 442 304	15 438 531 212
PAPI_L1_DCM	160 455	2 280 921	2 449 547	580 038	204 361	175 406	335 302 488
PAPI_L1_ICH	1 585 602	958 303 015	1 647 653 908	2 637 185 580	21 285 376	14 205 679	14 068 820 745
PAPI_L1_ICM	132	340 989	444 844	33 039 140	14 593	15 179	570 187 871
PAPI_L2_TCM	462	249 179	265 035	189 654	33 177	14 150	220 430 062
PAPI_L3_TCM	58	779	1380	1291	837	338	138 856 039
PAPI_BR_MSP	109	11 924 312	12 144 909	14 217 937	8439	8103	97 349 618
PAPI_FP_OPS	1 553 411	3 141 854	3 141 343	3 140 572	3 104 728	3 097 962	72 774 253
PAPI_VEC_DP	1 553 336	0	0	0	0	0	0
PAPI_LD_INS	1 548 866	807 870 130	1 611 839 505	2 392 560 416	31 290 468	8 419 395	10 550 022 589
PAPI_SR_INS	309 800	403 080 778	826 457 729	1 329 883 473	3 321 584	3 999 085	5 983 324 955

Tabla A.32: Resultados de **POLYBENCH_PAPI** para el benchmark **SEIDEL-2D** y tamaño del conjunto de datos **SMALL**

Counter	C	CPython-LL	CPython-FL	CPython-NP	PyPy-LL	PyPy-FL	PyPy-NP
PAPI_TOT_CYC	22 859 780	2 129 440 707	3 608 530 701	6 246 333 653	51 396 742	26 332 348	53 981 939 011
PAPI_L1_DCH	2 232 228	1 939 145 137	3 668 089 197	5 437 051 234	46 622 775	9 577 635	23 499 071 229
PAPI_L1_DCM	72 188	894 902	1 104 702	624 864	94 034	82 514	547 011 610
PAPI_L1_ICH	3 094 338	1 568 474 353	2 526 238 916	4 199 336 007	28 321 735	16 174 532	21 603 109 254
PAPI_L1_ICM	155	598 798	1 322 527	43 556 131	15 697	15 529	1 025 568 169
PAPI_L2_TCM	719	115 972	161 964	158 998	29 940	13 164	342 268 446
PAPI_L3_TCM	0	2490	3495	1800	1113	779	206 144 685
PAPI_BR_MSP	121	21 176 807	16 563 726	22 132 809	6455	6268	155 298 469
PAPI_FP_OPS	5 526 475	5 459 278	5 460 146	5 458 592	4 898 112	4 887 808	111 515 140
PAPI_VEC_DP	0	0	0	0	0	0	0
PAPI_LD_INS	1 755 906	1 324 547 717	2 566 366 674	3 803 480 458	44 201 344	8 374 875	16 074 642 971
PAPI_SR_INS	557 005	659 605 355	1 321 535 629	2 099 738 058	2 325 900	2 347 672	9 100 081 481

Lista de acrónimos

API *Application Programming Interface*
DCE *Dead Code Elimination*
FFI *Foreign Function Interface*
FIFO *First In First Out*
JIT *Just In Time*
PAPI *Performance Application Programming Interface.*
PEP *Python Enhancement Proposals*
RDTSC *Read Time Stamp Counter*
RTC *Real Time Clock*
TSC *Time Stamp Counter*
SCoP *Static Control Part*

Glosario

Azúcar sintáctico En lenguajes de programación hace referencia a una serie de añadidos al lenguaje para hacer algunas construcciones más fáciles de leer o expresar.

Compilador Programa que transforma un código escrito en un lenguaje de programación en otro código (otro lenguaje de programación, código máquina, bytecode, etc.)

Código abierto Modelo de desarrollo de software basado en la colaboración abierta que se enfoca en los beneficios prácticos de acceso público al código fuente.

Código fuente Conjunto de líneas de texto que indica a un ordenador qué pasos debe seguir para ejecutar un programa.

Directorio raíz Primer directorio o carpeta dentro de una jerarquía.

Gestor de paquetes Colección de herramientas que sirven para automatizar el proceso de instalación, actualización, configuración y eliminación del software de un sistema.

Repositorio software Lugar de almacenamiento del cual pueden ser recuperados e instalados paquetes de software en un ordenador. Los gestores de paquetes suelen valerse de los repositorios de software.

Subversion Herramienta de control de versiones centralizada de código abierto empleada en el desarrollo de software para guardar los cambios que se van produciendo en el tiempo y que permite compartir el código con otros usuarios.

Bibliografía

[1] T. G. team, "Gcc, the gnu compiler collection," https://gcc.gnu.org/, 2020.

[2] llvm-admin team, "The llvm compiler infrastructure," https://llvm.org/, 2020.

[3] IBM, "Ibm c and c++ compiler family," https://www.ibm.com/products/c-and-c-plus-plus-compiler-family, 2020.

[4] R. Labs, "R-stream - advanced polyhedral compiler," https://www.reservoir.com/rstream/.

[5] C. Draft, "Iso/iec 9899:tc3," http://open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf, 2007.

[6] M. Kerrisk, "Linux man pages: gettimeofday," https://www.man7.org/linux/man-pages/man2/gettimeofday.2.html, 2019.

[7] I. Corporation, *Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, I. Corporation, Ed. Intel Corporation, May 2020.

[8] "Linux man pages: posix_memalign," https://linux.die.net/man/3/posix_memalign.

[9] L.-N. Pouchet, "Pocc, the polyhedral compiler collection package," https://sourceforge.net/projects/pocc/, 2018.

[10] P. S. Foundation, "Python," https://www.python.org/, 2020.

[11] T. P. Team, "Pypy," https://www.pypy.org/, 2020.

[12] U. of Tennessee, "Papi," http://icl.cs.utk.edu/papi/, 2020.

[13] "Python 3 documentation: standard types," https://docs.python.org/3.7/library/stdtypes.html#numeric-types-int-float-complex, June 2020.

[14] "Python 3 documentation: decimal fixed point and floating point arithmetic," https://docs.python.org/3.7/library/decimal.html, June 2020.

[15] "Differences between pypy and cpython," https://doc.pypy.org/en/latest/cpython_differences.html, 2020.

[16] P. S. Foundation, "Design and history faq," https://docs.python.org/3/faq/design.html#how-are-lists-implemented-in-cpython, 2020.

[17] L. Diekmann and C. F. Bolz, "More compact lists with list strategies," https://morepypy.blogspot.com/2011/10/more-compact-lists-with-list-strategies.html, November 2011.

[18] "Python 3 documentation: standard types," https://docs.python.org/3.7/library/stdtypes.html#range, June 2020.

[19] G. van Rossum, B. Warsaw, and N. Coghlan, "Pep 8 – style guide for python code," https://www.python.org/dev/peps/pep-0008/, August 2013.

[20] "Python 3 documentation: Expressions," https://docs.python.org/3.7/reference/expressions.html, 2020.

[21] "Python 3 documentation: Time access and conversions," https://docs.python.org/3.7/library/time.html, 2020.

[22] C. Simpson, J. Jewett, S. J. Turnbull, and V. Stinner, "Pep 418 – add monotonic time, performance counter, and process time functions," https://www.python.org/dev/peps/pep-0418/, March 2012.

[23] A. Kervinen, "Inline assembly in python," https://github.com/askervin/python-il, January 2020.

[24] faineance, "Inline assembly/machine code in python," https://github.com/faineance/inlineasm, April 2016.

[25] "The rpython typer," https://doc.pypy.org/en/release-1.9/rtyper.html#primitive-types, 2011.

[26] "Python 3 documentation: miscellaneous operating system interfaces," https://docs.python.org/3.7/library/os.html#os.sched_param, April 2020.

[27] R. Ito, "Embed inline c / c++ source codes in python." https://pypi.org/project/inline/, November 2015.

[28] "Python 3 documentation: Package extension utility," https://docs.python.org/3.7/library/pathlib.html, 2020.

[29] "Python 3 documentation: Package extension utility," https://docs.python.org/3.7/library/shutil.html, 2020.

[30] "Python 3 documentation: Package extension utility," https://docs.python.org/3.7/library/pkgutil.html, 2020.

[31] "Python 3 documentation: Parser for command-line options, arguments and sub-commands," https://docs.python.org/3.7/library/argparse.html, 2020.