Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Trabajo Fin de Grado

Análisis automático del reuso de datos en aplicaciones de tiempo real con arquitectura x64

Automatic analysis of data reuse in real-time applications with x64 architecture

Autor

Álvaro Moreno Martín Viveros

Directores

Rubén Gran Tejero

Juan Segarra Flor

Grado en Ingeniería Informática

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2022

AGRADECIMIENTOS

Me gustaría transmitir mi más sincero agradecimiento a todas las personas que me han ayudado durante estos años en la universidad y la realización de este trabajo.

En primer lugar, a mis padres por el constante apoyo mostrado que sin duda ha servido como motor para llegar hasta aquí.

En segundo lugar, a aquellos familiares que se han preocupado sinceramente por mi recorrido esperando lo mejor de mí, especialmente a mi abuela Lola.

En tercer lugar, a mis dos queridos compañeros y mi querida compañera por compartir conmigo las aventuras y desventuras de este viaje de cuatro años, y ser capaces de convertir esta experiencia en una especie de sátira donde de los malos y los buenos momentos se puede salir siempre con unas risas.

Por último a Rubén y a Juan por mostrarse totalmente receptivos a ser mis directores desde el primer día que acudí a ellos. Además de la incansable ayuda que me han prestado desde entonces para que este trabajo salga adelante.

Análisis del reuso de datos en aplicaciones de tiempo real con arquitectura x64

RESUMEN

La tarea de estimar de forma precisa el tiempo en el peor caso (WCET) en sistemas de tiempo real estricto con memorias cache de datos es una labor compleja. Las memorias cache aprovechan las propiedades de reuso de los programas almacenando temporalmente ciertos contenidos de memoria cerca del procesador, con idea de que los accesos próximos a ese dato tengan un mayor rendimiento. Por este motivo, realizar un análisis seguro del reuso de datos (en el peor caso) presente en una aplicación de tiempo real puede ser muy útil para determinar el comportamiento que tendrán sus accesos a memoria de forma fiable. Para conseguir que el análisis sea seguro se plantea el reuso de datos como una propiedad del programa, pero el análisis es desarrollado sobre la versión del programa más cercana a la ejecución real, es decir, sobre el programa en bajo nivel. De esta forma, se puede plantear una metodología de análisis que aplicando interpretación abstracta sobre el programa compilado sea capaz de extraer para cada instrucción de acceso a memoria encontrada, una expresión lineal que defina su patrón de acceso a datos de acuerdo a la teoría de reuso de datos en bucles anidados. El grupo de arquitectura de computadores de Zaragoza desarrolló una herramienta (polygaz) [1] que aplica esta metodología para poner en práctica un análisis seguro del reuso de datos en aplicaciones de tiempo real sobre arquitectura ARMv7, pero no se ha encontrado ningún estudio posterior que haya tratado de ampliar la herramienta para cumplir esta misma labor sobre arquitecturas con un conjunto de instrucciones complejas (CISC). Por esta razón, durante este trabajo se busca expandir la herramienta y reproducir el mismo proceso de análisis sobre arquitecturas de 64 bits de Intel y AMD (x64). Para lograr este objetivo, ha sido necesario definir las bases del proyecto seleccionando los bancos de prueba y compiladores sobre los que se iba a trabajar, adaptar polygaz para incluir el análisis de x64 a su funcionalidad y buscar una forma adecuada de representación de resultados para su posterior valoración. Finalmente, se presenta una discusión entre las diferencias y similitudes, ofrecidas por los resultados, en el reuso de datos presentado por los binarios del banco de pruebas para las arquitecturas y compiladores seleccionados. Para las tareas probadas, se puede concluir que la herramienta mantiene su utilidad al trabajar con nuevas arquitecturas y que las características de arquitectura y compiladores tienen una incidencia importante en el reuso de datos presentado por una aplicación.

Índice

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

Hoy en día el tipo de sistemas informáticos que es posible encontrar es muy heterogéneo y amplio: centros de datos, móviles, IOT, sistemas embarcados, etcetera. Dentro de esta creciente variedad se encuentran los sistemas de tiempo real estricto (STRe). Unos sistemas muy importantes por sus aplicaciones en campos como la aviónica, medicina, la automoción o la robótica, este tipo de sistemas se caracterizan por tener, además de requisitos funcionales, requisitos temporales de las tareas que los componen, es decir, restricciones de periodo y plazo de ejecución. La forma de garantizar las restricciones temporales en estos sistemas es realizar la planificación de sus tareas usando el tiempo en el caso peor (WCET), el cual debe ser calculado formalmente. Esto hace que el cálculo de WCET deba ser planteado de una forma analítica, que en la práctica hace imposible incluir detalles micro-arquitecturales como la implementación de memorias cache en STRe.

El grupo de arquitectura de computadores de Zaragoza (gaZ) ha desarrollado una herramienta [1] capaz de analizar el reuso de datos para luego formalizar el WCET de las tareas ejecutadas en un procesador con memoria cache, a partir de esta información. Esta herramienta solo es capaz de trabajar con arquitectura ARMv7, lo que abre la posibilidad de ampliar su funcionalidad con otra arquitectura mayoritaria como x64 (Intel y AMD) para comprobar que sigue siendo efectiva sobre arquitecturas diferentes.

1.1. Objetivos y Alcance

El objetivo principal de este trabajo es el de conseguir un análisis del reuso de datos presente en aplicaciones de tiempo real ejecutadas sobre arquitecturas x64, es decir, arquitecturas de 64 bits con un conjunto de instrucciones complejo (CISC) utilizadas por Intel y AMD. Al estar planteado con vistas al ámbito del tiempo real, este trabajo también permitirá adquirir los conocimientos necesarios para poder desarrollar una metodología de análisis de reuso de datos enfocada desde el bajo

nivel que permita asegurar una mayor fiabilidad en los resultados. La realización de este trabajo establece varios pasos a seguir: (1) El estudio de la metodología de interpretación abstracta utilizada en [1] para desarrollar el análisis de reuso de datos de forma segura. (2) La modificación de la herramienta desarrollada en el artículo [1] por el gaZ, para hacerla capaz de analizar binarios de arquitecturas x64. (3) La preparación del entorno necesario para poder llevar a cabo el análisis, incluyendo la discusión y selección de todos los elementos imprescindibles para ello (bancos de pruebas y compiladores). (4) Un estudio de los resultados obtenidos que permita definir las diferencias experimentales observadas entre los diferentes casos de estudio planteados.

La realización del trabajo se ha cerrado completando todos los puntos anteriormente expuestos, realizando un análisis comparativo del reuso de datos obtenido en la serie de programas de prueba escogidos, tras modificar la herramienta para que fuese capaz de completar su funcionalidad analítica en todos ellos.

1.2. Estructura de la Memoria

La memoria del proyecto se divide en capítulos. El capítulo 2 presenta las bases teóricas sobre las que se sustenta la metodología de análisis aplicada, así como, la versión original de la herramienta antes de ser modificada. En el capítulo 3 se expone todo el recorrido de trabajo realizado desde las fases más tempranas, donde se definió el entorno, hasta las más tardías, donde se obtuvieron los resultados. En el capítulo 4 se explica como han sido representados los resultados y se plantean las observaciones que se pueden concluir de ellos. Por último, en el capítulo 5 se plantea el trabajo a futuro que podría ampliar el proyecto junto a las conclusiones extraídas del mismo.

Capítulo 2

Contexto y trabajo previo

En este capítulo se describe el contexto necesario para entender el trabajo realizado. De esta forma se explican los detalles más importantes relacionados con el reuso de datos y los principios de interpretación abstracta para comprender el funcionamiento de la herramienta con la que se ha trabajado. Además se presenta el trabajo previo [1], polygaz, para conocer su alcance y el punto de inicio del trabajo desempeñado.

2.1. Reuso de datos

Esta sección ofrece una visión conceptual del reuso de datos, exponiendo en que consiste, por qué es importante, como se clasifica y su relación con la jerarquía de memorias cache en un computador siguiendo el conocimiento extraído del artículo de referencia [1].

El reuso de datos presente en un programa se puede entender como la frecuencia con la que el acceso a un mismo dato o bloque de datos, se ve repetido dentro de alguno de los caminos presentes en su flujo de ejecución. Es posible realizar un análisis que estudie la relación entre las distintas instrucciones load/store presentes en un binario, para determinar cuando están accediendo a un dato que ya había sido accedido previamente. Estos accesos a memoria se pueden catalogar según el tipo de reuso que presentan para determinar a su vez, de una forma precisa, el potencial de producir un acierto o un fallo en una jerarquía de cache de datos.

La importancia de esta información radica en que puede ser utilizada para mejorar la eficiencia en términos de latencia y energía de las memorias cache, priorizando la presencia de los datos más reusados en los niveles de cache más próximos al procesador. Más allá del alcance de este trabajo, esta información tan precisa sobre el reuso se puede utilizar para mejorar el tiempo de ejecución de peor caso (WCET) en el ámbito de sistemas de tiempo real estricto [2].

2.1.1. Tipos de reuso

Como se ha comentado anteriormente, el resultado del análisis ofrece la clasificación del reuso observado en diferentes categorías. En primer lugar, se puede diferenciar el reuso dependiendo de si es provocado por una misma instrucción de memoria (*reuso individual*) o en cambio se debe a accesos provocados por un conjunto de instrucciones diferentes (*reuso de grupo*). En segundo lugar se puede observar el reuso desde la perspectiva de la dirección del dato accedido en memoria, hablando en este caso de *reuso temporal* cuando un mismo dato es reutilizado en el tiempo, la misma dirección de memoria es accedida constantemente, o de *reuso espacial* en caso de que se accedan elementos próximos, cómo en el recorrido de un vector dentro de un bucle.

2.1.2. Caches de datos

Los computadores modernos incluyen jerarquías de memorias cache que permiten mejorar el rendimiento del sistema en el acceso a datos de memoria. Estas caches son pequeñas memorias que se sitúan en el camino de datos entre el procesador y la memoria principal (RAM) del computador. La capacidad de almacenamiento se estructura en líneas o bloques que agrupan decenas de bytes (64 B en Intel y AMD) consecutivos del espacio de direccionamiento del computador. El reuso de datos presente en los programas, ya sea temporal o espacial, es aprovechado por las memorias cache, es decir, las direcciones de memoria accedidas por el procesador encuentran el dato buscado en la jerarquía de memorias cache sin necesidad de acceder a memoria principal. Por lo tanto, la latencia esperada para obtener el dato proveniente del acceso a memoria se reduce.

En un sistema con jerarquía de memoria cache, el análisis del reuso debe centrarse en los bloques o líneas reutilizados por las instrucciones load/store. A partir de ahora hablaremos de reuso temporal cuando una instrucción load/store presente un acceso a una línea de cache que ya había sido consultada previamente. Además, en términos de reuso espacial se encontrarán instrucciones que presenten un patrón de acceso con un *stride* pequeño que reincida numerosas veces sobre la misma línea de cache, ocasionando una buena tasa de acierto, e instrucciones con un *stride* tan grande que no se pueda garantizar el acierto en memoria cache por estar accediendo a bloques diferentes de forma constante, ocasionando una tasa de acierto pobre.

2.2. Compiladores

Aunque el reuso de datos en un programa podría entenderse como una característica propia del código de alto nivel, las técnicas de optimización aplicadas por los compiladores al compilar el programa pueden afectar en como se manifiesta este reuso. Por este motivo, si se quiere realizar un análisis de reuso de datos preciso orientado a aplicaciones de tiempo real y estimación de WCET, es necesario enfocar el análisis desde el nivel de lenguaje máquina para ceñirse lo máximo posible a la realidad de ejecución del programa. En capítulos posteriores se experimentará en este aspecto aplicando el análisis a binarios compilados por *clang* [3] y *gcc* [4], con el objetivo de ilustrar el grado de diferencia que se puede llegar a obtener en el análisis de reuso al utilizar dos compiladores diferentes.

2.3. Interpretación Abstracta

El proceso de interpretación abstracta se fundamenta sobre una conexión de Galois [5] establecida entre un dominio concreto y un dominio abstracto, dónde el dominio abstracto representa una sobre-aproximación de los subconjuntos en el dominio concreto. En el análisis planteado el dominio concreto queda definido por el conjunto de vectores $(r_0, ..., r_{n-1}) \in \mathbb{W}^n$ que representa el estado de los registros del programa analizado, con todos los posibles valores que pueden tomar. Siendo el conjunto $\mathbb{W} = \{0, ..., 2^{w-1}\}$ la representación de todos los valores que puede tomar cada uno de los registros con una palabra de w bits. En el caso del dominio abstracto éste se ve representado por un conjunto de invariantes que se mantienen para todo el conjunto de registros. Para resolver la representación de los elementos en el dominio abstracto se puede usar *apron* [6], una biblioteca que ayuda al uso de interpretación abstracta para el análisis estático de las variables numéricas de un programa. De entre los diferentes dominios numéricos abstractos con los que *Apron* permite trabajar, se selecciona el dominio de los poliedros para representar los elementos en el dominio abstracto con una serie de inecuaciones lineales, también llamadas restricciones lineales, con la siguiente forma [1]:

$$\sum_{i \in \{0,...,n-1\}} c_i r_i \leq k, \quad c_i, k \in \mathbb{Z}. \tag{2.1}$$

En la figura 2.1 se puede observar un ejemplo donde se considera un subconjunto de dos registros. En cierto punto de la ejecución del programa los registros pueden contener los valores en el conjunto S, estos valores representados en la figura por puntos sólidos compondrían los estados concretos del dominio. Por otra parte el área

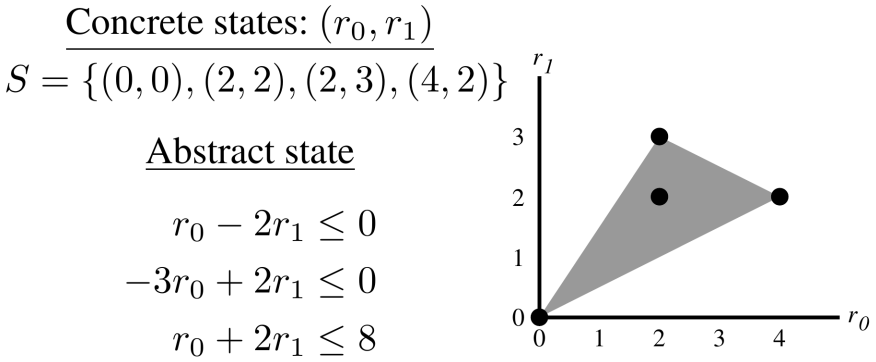


Figura 2.1: Dominios concreto (puntos sólidos) y abstracto (zona sombreada) en 2.3 [1].

sombreada representaría el estado abstracto, dónde se produce una sobre-aproximación de S, es decir, aparecen estados como $(1, 1)$ que también satisfacen los tres invariantes a pesar de no pertenecer a S.

Dentro del ámbito del estado abstracto se incluyen también dos valores especiales $\bot$ o vacío y $\top$. El símbolo $\bot$ hace referencia al valor más bajo posible en el dominio abstracto (ningún valor), siendo utilizado para inicializar todos los estados abstractos al comienzo del proceso de interpretación. Dado que durante el proceso de interpretación abstracta se calculan los diferentes estados abstractos en cada punto del programa visitando iterativamente todas sus instrucciones, encontrar un valor $\bot$ (ningún valor) en un estado indicará que este no se ha visto actualizado hasta ese punto. Por otra parte el símbolo $\top$ representa el valor más alto posible en el dominio abstracto (cualquier valor), este valor se utiliza en el proceso de interpretación para representar que el valor correspondiente a un estado en ese punto de ejecución no puede ser conocido por las características concretas del programa. Por ejemplo, esto ocurriría si se ha realizado una instrucción que aplica una transformación no expresable linealmente (ver sección 2.3.1), provocando que no pueda ser evaluada en el dominio de los poliedros del álgebra euclídea.

En resumen la técnica de interpretación abstracta permite garantizar que cualquier propiedad segura presente en el dominio abstracto, se mantiene también en el dominio concreto. Esta propiedad permite que la técnica utilizada encuentre patrones de acceso a datos en el programa de forma segura, mediante el análisis del contenido de los registros representado por cada uno de los estados abstractos.

2.3.1. Funciones de transferencia

La herramienta de interpretación abstracta requiere una serie de funciones que transformen los distintos estados abstractos durante el recorrido del programa. A estas funciones se las conoce como funciones de transferencia, las cuales capturan

el comportamiento de cada instrucción y transforman el estado abstracto en el que resultaría tras la ejecución de la instrucción.

Para realizar la captura del comportamiento de una instrucción, se realiza una micro-decodificación de cada una de ellas que las descompone en una secuencia de micro-operaciones básicas, es decir, operaciones que contienen como máximo un sólo registro destino y operaciones aritmético lógicas unarias o binarias, por ejemplo $r_j = r_i + r_k$.

La función de transformación del estado abstracto para la ejecución de cualquier instrucción I siguiendo la forma $r_{dst} \leftarrow ld$ (operandos en el lado derecho), estaría definida de la siguiente manera [1]:

$$Transfer(I) = \begin{cases} ld & \text{si } ld \in \{r_i, k, r_i \pm r_j, r_i \pm k, r_i \cdot k\} \\ \top & \text{otro caso} \end{cases} \tag{2.2}$$

Donde r_i y r_j hacen referencia a los registros fuente de la instrucción y k representa un valor entero constante. En la figura 2.2 se puede observar como sería expresada una micro-operación de suma de dos registros y la transformación que supondría desde el estado abstracto previo a su ejecución, hasta el estado abstracto posterior.

Por otra parte, debido a que el dominio de los poliedros empleado para representar el dominio abstracto solamente permite relaciones lineales para expresar sus estados, la función de transferencia asigna el valor más alto del dominio abstracto $\top$ al registro destino, siempre que la instrucción procesada no pueda ser expresada con una relación lineal entre los registros, como pasaría con $r_{dst} \leftarrow r_i \cdot r_j$.

2.3.2. Contenido de memoria

La base teórica expuesta en las secciones previas 2.3 y 2.3.1 trata la interpretación abstracta centrándose únicamente en el seguimiento de los valores en los registros. Sin embargo, para asegurar un correcto análisis cuando se trabaja sobre programas con código no optimizado, es necesario ampliar estos conceptos a la memoria para seguir los casos donde el valor de un registro es volcado en memoria y recuperado constantemente. Para ello se puede considerar la memoria como un gran banco de registros, dónde un acceso a memoria con una instrucción load/store puede verse similar a una instrucción mov sobre dos registros convencionales. Este comportamiento es cierto excepto en los

$$\begin{array}{|r|} \hline r_0 - 2r_1 \leq 0 \\ -3r_0 + 2r_1 \leq 0 \\ r_0 + 2r_1 \leq 8 \\ \hline \end{array} \xrightarrow{r_0 \leftarrow r_0 + r_1} \begin{array}{|r|} \hline r_0 - 3r_1 \leq 0 \\ -3r_0 + 5r_1 \leq 0 \\ r_0 + r_1 \leq 8 \\ \hline \end{array}$$

Figura 2.2: Función de transferencia para la instrucción `add r0,r1` [1]

siguientes casos: (1) la ejecución de una instrucción load sobre una dirección de memoria desconocida provoca que se asigne el valor $\top$ al registro destino, y (2) la ejecución de una instrucción store sobre una dirección desconocida asigna el valor $\top$ a todas las posiciones de memoria [1].

2.3.3. Variables de inducción

Las variables de inducción son aquellas variables que se ven incrementadas o decrementadas de forma fija a lo largo de las iteraciones de un bucle. Estas variables son críticas a la hora de estudiar la forma en que los vectores son accedidos en un bucle.

Es posible explotar el uso de las técnicas de interpretación abstracta para detectar las variables de inducción almacenadas en los registros dentro de un bucle natural. Para ello se crea primero una variable r^{prev} para almacenar el valor de cada registro r en la iteración anterior del bucle. Posteriormente en la entrada al bucle se produce la asignación $r^{prev} = r$ y en cada transición de retorno del bucle se realiza el calculo $r^{step} = r - r^{prev}$. De esta forma, se consigue detectar una variable de inducción en el bucle cuando para la unión de los estados presentes en todas las transiciones de retorno se cumple que $r^{step} = k,\ k \in \mathbb{Z}$.

La captura de estas variables de inducción es útil en relación a la captura de los patrones de acceso a memoria de las diferentes instrucciones load/store en un programa, pudiendo existir accesos constantes (accesos a variables escalares), accesos secuenciales (lineales a una variable de inducción) o no lineales [1].

2.4. Teoría de reuso

El análisis de reuso se basa en la teoría de reuso de datos en bucles anidados [7], para conseguir una función simplificada $f(\vec{i}) = \vec{h} \cdot \vec{i} + c$ que define la indexación de cualquier estructura de datos mapeada en memoria, vista como un espacio de una sola dimensión. $\vec{i}$ representa el vector de variables de inducción $(i_1, i_2, ..., i_n)$ dónde i_j hace referencia al valor de iteración del j-ésimo bucle anidado, contando desde el más externo al más interno. El vector $\vec{h}$ hace referencia a la combinación lineal aplicada sobre las variables de inducción y por último la constante c representa la dirección base de la estructura en memoria. Aplicando esta función se puede describir la captura de reuso para patrones de acceso lineales, por ejemplo para capturar reuso de grupo sería suficiente con encontrar dos referencias a memoria distintas dónde $\vec{h_1} = \vec{h_2}$ y $c_1 = c_2$, o para encontrar auto reuso temporal en una misma instrucción bastaría con conseguir una expresión tal que $\vec{h} \cdot \vec{i_1} + c_1 = \vec{h} \cdot \vec{i_2} + c_2$.

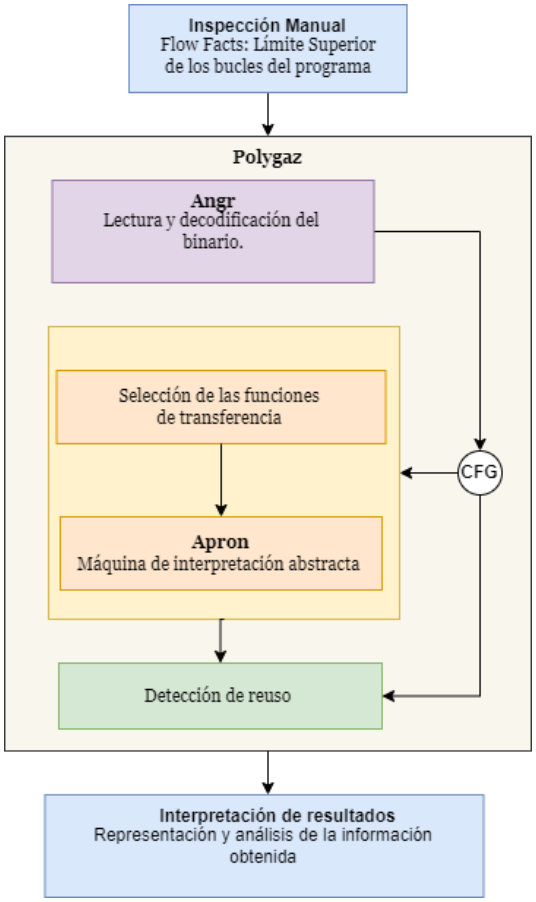
En patrones donde la dirección base se presenta desconocida, es decir $f(\vec{i}) = \vec{h} \cdot \vec{i} + c$

con c no conocida, el reuso espacial es perfectamente detectable puesto que la dirección base no es necesaria para ello. En cambio el reuso temporal de grupo, ya no sería detectable atendiendo a la teoría del reuso de datos por ser necesaria c para ello. No obstante, puede conseguirse la captura de este tipo de reuso a partir de la fase de interpretación abstracta, estableciendo que es posible detectar reuso de grupo si al comparar dos variables que contienen la dirección objetivo de un acceso a memoria estas son idénticas.

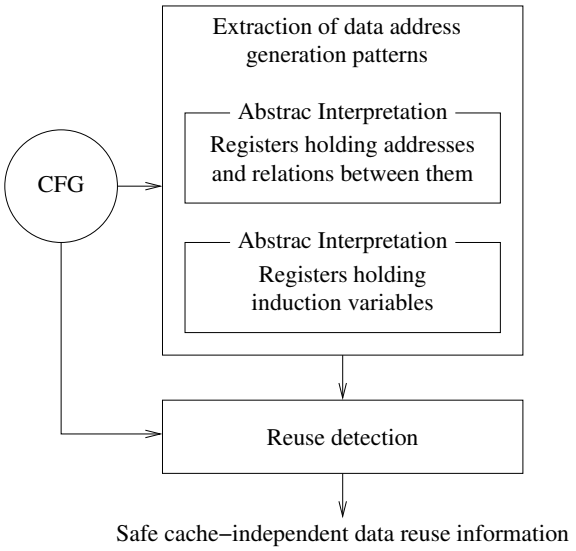
Para tratar accesos no lineales se pueden aplicar los mismos principios enunciados anteriormente, es decir, capturar el reuso de grupo utilizando la información de la interpretación abstracta, puesto que a partir de la teoría de reuso de datos sería imposible [1].

2.5. Herramienta polygaz

La herramienta *polygaz* desarrollada en el gaZ (Grupo de Arquitectura de Zaragoza) [1], se fundamenta sobre los principios teóricos y la metodología expuestos en el propio artículo y las secciones 2.3 y 2.4. De esta forma, consigue implementar una solución capaz de llevar a cabo un análisis de reuso de datos enfocado en aplicaciones de tiempo real, ejecutadas sobre sistemas que cuentan con arquitectura ARMv7 de 32 bits y jerarquía de memorias cache. En la figura 2.3a puede observarse una descomposición modular del comportamiento de la herramienta. Primero sería necesario acotar el dominio de iteración de los bucles presentes en el programa a analizar, ya que el análisis de reuso se esta enfocando sobre aplicaciones en tiempo real para tener la posibilidad de utilizarlo posteriormente en análisis de WCET. Después la herramienta utilizará *angr* [8] [9] [10] para la lectura del binario y su decodificación en un lenguaje intermedio basado en micro-operaciones básicas como las comentadas en la sección 2.3.1. A continuación, la herramienta recorrerá todos los posibles caminos de ejecución presentes en el grafo de flujo de control (CFG) del programa analizado. Definiendo para cada micro-operación básica encontrada, la expresión que representa la función de transferencia que modificará el estado abstracto tras su ejecución. Estas expresiones son utilizadas en la máquina de interpretación abstracta, ilustrada en la figura 2.3b, que emplea la biblioteca *apron* para actualizar las restricciones creadas en el dominio abstracto en cada punto del programa, en base a las funciones de transferencia definidas. De esta forma se puede extraer del estado abstracto la información sobre los registros que contienen direcciones de memoria y variables de inducción cómo se ha visto en la sección 2.3.3, para posteriormente aplicar los principios vistos en la sección 2.4 y obtener así el reuso de datos observado en el programa analizado.



(a) Esquema modular de polygaz.



(b) Máquina de interpretación abstracta [1]

Figura 2.3: Visión conceptual de polygaz

Capítulo 3

Análisis de reuso en x64

Este capítulo trata cada una de las fases en las que el trabajo se ha dividido a lo largo de su línea temporal (ver anexo F.1). En las secciones 3.1 y 3.2 se expone la toma de decisiones realizada para definir el entorno de trabajo. La sección 3.3 detalla el trabajo que es necesario realizar antes de utilizar la herramienta. En la sección 3.4 se tratan las modificaciones realizadas sobre la versión original de polygaz. Por último, en la sección 3.5 se muestran los procesos de análisis y obtención de resultados.

3.1. Elección de compiladores

La elección de *gcc* (versión 9.4.0) se basa en la naturaleza clásica de este compilador, el cual ha sido ampliamente utilizado en el tiempo como compilador de referencia para código escrito en lenguaje C. Además, al ser el mismo compilador empleado en [1] para el análisis, podemos realizar un estudio comparativo de las transformaciones aplicadas por el compilador y ver si se obtiene un reuso de datos similar, a pesar de las diferencias arquitecturales (ARMv7 vs. x64). Por otro lado, la elección de *clang* (versión 10.0.0) se basa en que se trata de otro compilador cuyo código es de desarrollo abierto, el cual además cuenta con la posibilidad de utilizar LLVM [11] como *backend*. Al utilizar dos compiladores diferentes, se han podido analizar las diferencias en el reuso de datos analizado provocadas por sus distintos criterios a la hora de aplicar transformaciones sobre el código de alto nivel.

3.2. Elección de benchmarks

Para la realización del trabajo se ha decidido trabajar con los *benchmarks* de la colección TACLeBench [12], la cual ofrece un banco de pruebas variado para el análisis de WCET en aplicaciones de tiempo real estricto. Los fuentes de esta colección son una buena opción para el análisis de WCET debido a dos motivos principales: (1) Los

fuentes están etiquetados con el máximo número de iteraciones esperadas para cada bucle construido en alto nivel, y (2) Los fuentes no utilizan bibliotecas, si no que todo su código forma parte del propio fuente.

La colección se compone actualmente de un total de 60 programas de prueba divididos en cinco categorías diferenciadas: (1) *kernel*, listada en A.3, que incluye módulos que implementan pequeñas funciones de núcleo, (2) *sequential*, listada en A.4, cuyos módulos se asemejan a las operaciones de codificación y decodificación que ocurren dentro de muchos sistemas empotrados, (3) *test*, listada en A.5, que implementa pruebas de estrés para entornos de análisis de WCET, (4) *app*, listada en A.6, compuesta por aplicaciones de tiempo real ya existentes y (5) *parallel*, listada en A.7, que incluye los *benchmarks* paralelos de la colección.

La selección de TACLeBench se debe a que esta misma colección es la utilizada en [1] para obtener los resultados correspondientes a la arquitectura ARMv7, siendo posible de esta forma comparar los resultados sobre la misma base. Para el desarrollo del proyecto se ha tomado la misma decisión que en [1], donde los *benchmarks* recursivos se mantuvieron fuera del ámbito del análisis, descartando de este modo siete *benchmarks* (*anagram*, *huff_enc*,*bitcount*, *bitonic*, *fac*, *quicksort* y *recursion*) de entre el conjunto inicial de la colección. También, puesto que se persigue un análisis enfocado en aplicaciones de tiempo real, se han descartado los tres *benchmarks* correspondientes a la categoría *parallel*, debido a que los distintos entrelazados que pueden surgir durante el flujo de ejecución de un programa paralelo impiden que el análisis de reuso de datos pueda ser planteado de forma segura, es decir, no se puede garantizar con el análisis que los resultados se mantuviesen iguales en todos los casos de ejecución posibles.

3.3. Inspección Manual

Para conseguir que la herramienta sea capaz de ejecutar el análisis de los *benchmarks* seleccionados, es necesaria una fase previa donde se deben definir para cada binario generado los límites superiores de los bucles presentes en ellos. Esta información es útil para la herramienta, ya que de esta manera puede descartar los bucles que sean imposibles (límite superior 0) en el camino de ejecución. Además, también podría ser utilizada en el hipotético caso de querer ampliar el proyecto en un futuro con un análisis de WCET similar al realizado en [2].

Con intención de desarrollar esta tarea, primero se ha realizado la compilación de los *benchmarks* empleando los compiladores (*gcc* y *clang*) y niveles de optimización seleccionados (O0 y O2), obteniendo finalmente 200 binarios para el análisis. En segundo lugar, se ha usado la utilidad *mkff.py* integrada dentro de polygaz para generar

plantillas que identifiquen en que posición del binario se encuentra cada bucle. Por último, se ha accedido manualmente al código ensamblador de cada binario utilizando la herramienta de visualización *angr management* [13], para inferir el número máximo de iteraciones correspondiente a cada bucle presente en las plantillas, mediante la exploración manual del código ensamblador de cada binario.

La fase de inspección manual es especialmente importante por dos motivos: (1) aunque TACLeBench proporciona en el código fuente de alto nivel etiquetas con el máximo número de iteraciones de cada bucle, estas etiquetas pueden contener errores. Y (2) las transformaciones aplicadas sobre los bucles por los compiladores cuando se utiliza el nivel de optimización O2 (desenrollado de bucles y vectorización), pueden cambiar el número de bucles encontrados en el fuente, así como, el límite superior esperado en ellos.

Como ejemplo del primer caso, durante el trabajo realizado se detectó que en el binario *h264_dec* compilado con nivel de optimización O0 (no se aplican transformaciones de bucles), el límite superior que marcaban las etiquetas en los bucles de la función h264_dec_init() no se correspondía con el límite observado en el código de bajo nivel. Al observar esta diferencia, se comprobó experimentalmente mostrando por pantalla el tamaño de los vectores sobre los que iteraban los bucles conflictivos, que las conclusiones extraídas en la inspección eran correctas y por tanto existía un error en los fuentes de TACLeBench. Finalmente se notificó el error a los responsables de TACLeBench, quienes verificaron el desacierto en las etiquetas y corrigieron los fuentes (ver anexo G.3).

Por otra parte, para ilustrar el segundo caso se han obtenido informes (opciones de compilación -Rpass=unroll|vectorize en *clang* y -fopt-info-loop en *gcc*) que indican las veces que ambos compiladores aplican transformaciones de bucles en cada binario, para el nivel de optimización O2 (ver anexo D.1). A continuación se muestran varios ejemplos que ilustran los distintos tipos de transformaciones obtenidas:

I. Desenrollado

   En el caso de desenrollado, se ha decidido destacar el ejemplo encontrado en el binario *huff_dec*, en el bucle perteneciente a la función huff_dec_return(). Este bucle en alto nivel compara secuencialmente dos cadenas de caracteres, devolviendo el valor del iterador más uno en caso de encontrar una posición donde los caracteres no sean iguales. Como se puede observar en el código 3.1 el límite superior marcado para el bucle es de 600 iteraciones. De igual modo ocurre en nivel de optimización O0 (ver anexo G.1), pero si se inspecciona este bucle en el binario correspondiente al compilador *clang* con nivel de optimización en O2,

Figura 3.1: Código huff_dec_return().

```
int huff_dec_return( void )
{
    int i;
    _Pragma( "loopbound min 600 max 600" )
    for ( i = 0; i < huff_dec_plaintext_len; i++ ) {
      if ( huff_dec_plaintext[ i ] != huff_dec_output[ i ] ) return i + 1;
    }
    return 0;
}
```

lo que se encuentra es diferente (ver anexo G.2). Ahora, puesto que el compilador ha realizado un desenrollado de grado 5, en el cuerpo del bucle ha replicado 5 veces la sentencia que comprueba si se han encontrado dos caracteres diferentes (provocando el fin del bucle). Así mismo se ha dividido por 5 el número máximo de iteraciones, lo cual debe ser corregido a 120 en lugar de 600.

II. Vectorización

En el caso de vectorización, se ha encontrado muy interesante mostrar las transformaciones sufridas por los bucles pertenecientes a las funciones memcpy() y memset() que se encuentran redefinidas en varios ejemplos de TACLeBench. Estas funciones implementan bucles que realizan copias sobre estructuras de datos de diferentes tamaños, por ejemplo se puede mirar con detalle la función memcpy() presente en el programa de prueba *pm*, tal y como se muestra en el código 3.2. Este código implementa un bucle que copia byte a byte (tamaño de una variable *unsigned char*) desde un vector origen hacia un vector destino, con un máximo de 256 iteraciones. Si se observa el bucle dentro del binario para un nivel de optimización O0 se encuentra el mismo comportamiento (ver anexo G.4). En cambio, si se observa el binario compilado por *clang* en un nivel de optimización

Figura 3.2: Código pm_memcpy().

```
void pm_memcpy( void *dest, void *src, int size )
{
    int i;
    _Pragma( "loopbound min 44 max 256" )
    for ( i = 0; i < size; i++ )
        ( ( unsigned char * )dest )[ i ] = ( ( unsigned char * )src )[ i ];
    return;
}
```

O2 lo que se encuentra es que el bucle ha sido dividido en cuatro. Estos nuevos bucles utilizan instrucciones vectoriales para ejecutar la copia de la estructura en distintos tramos, el primero de ellos realiza una copia en bloques de 128 B (ver anexo G.5), el segundo en bloques de 32 B (ver anexo G.6), el tercero en bloques de 1 B para cuando se traten tamaños impares (ver anexo G.7) y el último en bloques de 4 B (ver anexo G.8). De esta forma, de un solo bucle surgen cuatro bucles distintos que necesitan ser estudiados para definir su límite superior.

Además durante la fase de inspección manual se han descartado tres *benchmarks* (*ammunition*, *gsm_enc* y *test3*) por errores provenientes de la utilidad *angr* [8] utilizada para analizar el binario, que ocasionaban la generación de plantillas erróneas. La razón es que *angr* es una herramienta en desarrollo y por tanto puede presentar errores en la decodificación de ciertas instrucciones que resulten en un CFG inválido que impida la correcta elaboración de las plantillas. Además se ha descartado el *benchmark lms* por la presencia de bucles que no habían sido etiquetados en TACLeBench.

3.4. Modificaciones en la herramienta

En esta sección se exponen los cambios realizados sobre la versión original de la herramienta polygaz para conseguir el análisis sobre x64. Todos los cambios realizados pueden ser consultados en el repositorio público de la herramienta x64-polygaz [14] (*analysis1.py*).

En la sección 3.4.1 se explica cómo se inicializaron los registros de propósito general de x64 en el dominio abstracto. En la sección 3.4.2 se expone como se definieron las funciones de transferencia correspondientes a las nuevas instrucciones encontradas en la arquitectura. En la sección 3.4.3 aparecen los nuevos tipos de datos definidos en la herramienta. Y por último en la sección 3.4.4 se desarrollan las correcciones de errores realizadas respecto de la versión original.

3.4.1. Inicialización de registros x64

Para facilitar a la herramienta el análisis, ha sido necesario explorar como define *angr* los 16 registros de propósito general de la arquitectura x64 e implementar una función que los inicialice en la herramienta siguiendo ese mismo estándar. Esto es importante porque estos registros van a formar los estados abstractos (ver sección 2.3) que deben ser actualizados por las funciones de transferencia (ver sección 2.3.1) y son a su vez utilizados para extraer el reuso.

Puesto que no existe una documentación detallada acerca de los valores que asigna *angr* a los registros de cada arquitectura con la que trabaja, los valores se han obtenido

trabajando experimentalmente con la colección de clases *archinfo* [15] que utiliza angr para definir información específica sobre cada arquitectura. Concretamente *archinfo* ofrece los métodos arch_from_id() y get_register_offset(). El primer método crea un objeto con la información de la arquitectura deseada, y sobre este objeto es posible invocar el segundo método para obtener el valor con el que *angr* identifica cualquier registro perteneciente a la arquitectura.

Finalmente, a través de esta labor experimental se ha hallado que *angr* establece un valor inicial para el registro que identifica como registro 0 de la arquitectura, e incrementa este valor en base al tamaño en bytes de cada registro para obtener los valores posteriores. En el caso concreto de x64, la clasificación de los 16 registros de propósito general obedece la expresión $reg = (i + 2) \cdot 8, \quad i \in \{0, 1, ..., 15\}$. Para más detalle en este aspecto de la herramienta es posible consultar la función init_regs_x86_64() del script *analysis1.py* en la herramienta x64-polygaz [14] y la tabla B.1 en el anexo.

3.4.2. Nuevas instrucciones

Conseguir que la herramienta sea capaz de analizar una nueva arquitectura implica que ésta debe ser capaz de interpretar nuevas instrucciones. Estas instrucciones, como ha sido comentado en la sección 2.3.1, son decodificadas por *angr* en micro-operaciones cuyo comportamiento debe ser definido para construir las restricciones en el estado abstracto. Por tanto, para cada nueva micro-operación encontrada durante el análisis cuyo comportamiento no había sido definido previamente, primero ha sido necesario estudiar cómo definirla como función de transferencia (ver sección 2.3.1) y posteriormente implementarla en la herramienta.

Las operaciones añadidas se han clasificado en varios grupos según su funcionalidad, habiéndose añadido finalmente conjuntos de operaciones aritméticas, operaciones lógicas, operaciones vectoriales, operaciones en coma flotante, operaciones de conversión, operaciones geométricas y operaciones de comparación. Esta clasificación y como se ha tratado cada operación puede verse detalladamente en el anexo B.1.

3.4.3. Accesos a memoria: nuevos tipos de dato

Al ampliar el campo de análisis de la herramienta desde una arquitectura de 32 bits a una arquitectura de 64, surgen nuevos posibles accesos a memoria que presentan un tamaño mayor a los que ya existían implementados. En total se ha ampliado la funcionalidad de la interpretación de los accesos a memoria con tres tipos de acceso adicionales a los ya existentes previamente: (1) Ity_64 define en *angr* accesos a datos enteros de 64 bits,(2) Ity_F64 define en *angr* accesos a datos en coma flotante de 64 bits,

y (3) Ity_V128 define en *angr* accesos vectoriales a memoria de 128 bits. Para tratar estos accesos se han decodificado en la herramienta estos tipos de datos especificados por *angr* indicando el número de bytes que implica cada acceso, al igual que se hacía con los otros tipos de dato en la versión previa de la herramienta.

3.4.4. Correcciones en la herramienta

Durante la realización del análisis además de los cambios necesarios para conseguir que la herramienta fuese capaz de trabajar con binarios x64, también se han corregido los comportamientos erróneos heredados de la versión inicial del analizador que se han podido observar. Se han encontrado dos errores principales a los que proponer solución durante la ampliación de la herramienta: (1) gestión de desbordamiento de enteros durante la interpretación de operaciones aritméticas y de comparación, y (2) gestión de las restricciones provocadas por una operación de comparación en un bloque básico con dos o más comparaciones. La descripción detallada de estos errores puede verse comentada en la sección del anexo B.2.

La solución ofrecida al primer problema ha sido la de incluir en las operaciones de resta una comprobación, antes de construir la expresión correspondiente, que verificase si un operando interpretado como valor natural debería ser tratado en realidad como un entero negativo. En caso afirmativo se ha realizado una transformación Ca2 del valor, en lugar de multiplicar el operando por -1 causando desbordamiento como en la versión original de la herramienta.

El segundo error, el cual provocaba un camino de ejecución fallido en la versión original, se ha corregido añadiendo un contador adicional a la herramienta que identificase el número de comparaciones útiles en un bloque básico y reiniciase su valor con 0 si se encuentra más de una. Además se ha mantenido el contador de comparaciones totales original, haciendo que ahora al comprobar si se debe añadir una nueva restricción por operaciones de comparación al finalizar un bloque básico, no solo se comprueba que haya ocurrido alguna (número total de comparaciones mayor que 0), si no que también se comprueba si estas comparaciones son útiles (número de comparaciones útiles mayor que 0). De otro modo, la restricción no es tratada para evitar el problema que existía en la versión original por añadir una restricción inexistente.

3.5. Desarrollo del análisis

El proceso de análisis ha sido desarrollado en el entorno planteado dentro del anexo A, utilizando el *script* reuse_detection.sh proporcionado inicialmente con la herramienta polygaz, al que se han añadido ligeras modificaciones para adaptarlo al

entorno de trabajo desplegado. Este *script* permite ejecutar el analizador de reuso modificado analysis1.py sobre los binarios presentes en la ruta seleccionada, generando un archivo .reuse con la traza de ejecución del análisis realizado y un archivo .xml con los resultados obtenidos para cada acceso a memoria encontrado. El formato detallado de estos archivos .xml puede verse en el anexo C.

Durante el proceso de análisis se han descartado el *benchmark mpeg2* por ser demasiado costosos en recursos de memoria, la ejecución del análisis era abortada por llegar a los límites de la máquina donde se estaba ejecutando (256 GB), y los programas de prueba *epic*, *md5* y *susan O2* por ser demasiado costosos en recursos temporales, por ejemplo uno de los análisis de *epic* fue abortado tras dos semanas sin finalizar. Finalmente el banco de pruebas para obtener resultados quedó cerrado en un total de 42 fuentes (168 binarios).

3.5.1. Obtención de resultados

Para obtener resultados útiles para su valoración se ha planteado un enfoque alternativo a [1]. Esta vez se han obtenido resultados en grueso sobre los datos de reuso temporal y espacial (ver sección 2.1.1) presentes en los archivos .xml.

Para representar el reuso temporal se ha seguido un modelo donde éste queda definido, para cada binario analizado, por un conjunto de tuplas de tamaño dos tal que $\langle l, r \rangle$, $l, r \in \mathbb{N}$. La primera componente dentro de la tupla hace referencia a la longitud de una cadena de reuso temporal observada, es decir, cuantos accesos a memoria a lo largo del flujo de programa han presentado reuso en cadena desde un primer acceso a memoria dominante, o en otras palabras, el acceso que utilizó el dato reusado por primera vez. La segunda componente denota, en cambio, el número de veces que se ha repetido la presencia de una cadena de esa longitud dentro del binario. Por ejemplo en

Figura 3.3: Ejemplo de obtención de resultados en reuso temporal.

```
...
mov eax, [0x601040]; acceso a memoria dominante: <id=1/> <firstreused=1/>
mov [0x601040], 0x0; <id=2/> <firstreused=1/> <lastreused=2/>
mov edx, [0x601040]; <id=3/> <firstreused=1/> <lastreused=3/>
mov ebx, [0x602040]; acceso a memoria dominante: <id=4/> <firstreused=4/>
mov [0x602040],0x0; <id=5/> <firstreused=4/> <lastreused=5/>
mov ecx, [0x602040]; <id=6/> <firstreused=4/> <lastreused=6/>
...
```

la secuencia de accesos a memoria presentada en la figura 3.3, se observa que hay tres accesos que reutilizan la dirección de memoria 0x601040 y tres accesos que reutilizan la dirección de memoria 0x602040, ya que ambas direcciones no pertenecerán a la misma

línea de cache por estar a más de 64 Bytes (tamaño de línea en Intel) de distancia. En esta secuencia se cuenta con dos cadenas de reuso (r) formadas cada una por tres accesos a memoria cada una (l), lo cual en el modelo propuesto quedaría representada con la tupla $\langle 3, 2 \rangle$.

En el caso del reuso espacial se ha seguido el mismo modelo de representación por tuplas, pero en este caso la primera componente hace referencia al número de accesos que presentan reuso espacial dentro de un mismo bucle. Por ejemplo, se pueden plantear los bucles vistos en la figura 3.4 donde se establecería que en cada bucle hay una cadena de reuso espacial de longitud dos, ya que cada uno cuenta con dos accesos que recorren un vector. Quedando finalmente expresado en el modelo de tuplas de reuso espacial como dos cadenas (r) de longitud dos (l), es decir, $\langle 2, 2 \rangle$.

Para obtener el conjunto de tuplas correspondiente a cada binario se ha partido del script *xmlparse.py*, presente originalmente en polygaz, para crear *reuseparse.py*. Una herramienta capaz de extraer las tuplas de reuso temporal y espacial correspondientes a las secuencias de accesos a memoria representadas en cada archivo .xml, y generar para cada tipo de reuso un archivo .csv con la información sobre las tuplas de reuso correspondientes a cada binario. Estos archivos .csv han sido finalmente utilizados junto a una utilidad desarrollada en lenguaje R [16] (reuse_boxplot.r), para representar gráficamente el reuso temporal y espacial hallado en cada caso de comparación.

Figura 3.4: Ejemplo de obtención de resultados en reuso espacial.

```
...
    mov rdx,-0x20; Inicializar iterador
bucle1:
    mov rax, [0x601040 + rdx]; <term i=bucle1 h=8/>
    mov [0x601040 + rdx], 0x0; <term i=bucle 1 h=8/>
    add rdx, 0x08; Actualizar iterador
    jne bucle1; Si el iterador es 0, salir de bucle1
    mov rdx,-0x20
bucle2:
    mov rax,[0x602040 + rdx]; <term i=bucle2 h=8/>
    mov [0x602040 + rdx], 0x0; <term i=bucle2 h=8/>
    add rdx, 0x08; Actualizar iterador
    jne bucle2; Si el iterador es 0, salir de bucle2
...
```

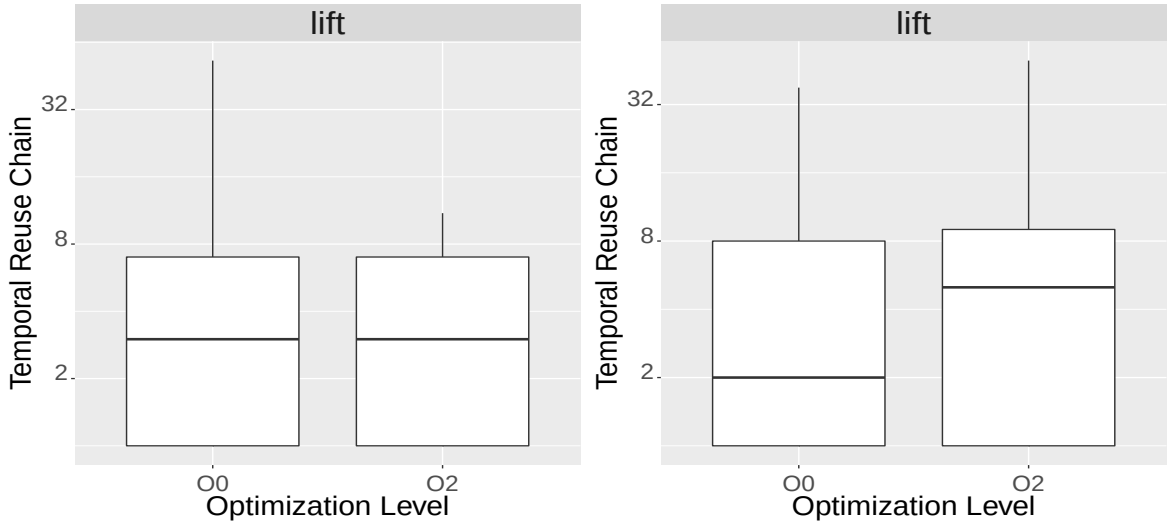
Capítulo 4

Resultados Experimentales

En este capitulo se evalúan los resultados obtenidos tanto en términos de reuso temporal, sección 4.1, como de reuso espacial, sección 4.2. Siguiendo el modelo de representación explicado en la sección 3.5.1 se plantean gráficos dónde se comparan los resultados conseguidos en este trabajo y el artículo [1], para observar las diferencias presentes entre las arquitecturas x64 y ARMv7. Así como también se comparan los resultados entre ambos compiladores: *clang* y *gcc*, para observar como afectan al reuso las transformaciones aplicadas por estos en cada nivel de optimización: O0 y O2. El entorno experimental desplegado para obtener estos resultados puede ser encontrado en el Anexo A.

4.1. Resultados en reuso temporal

Para valorar como afectan al reuso de datos las optimizaciones aplicadas por el compilador se puede atender a la figura 4.1. En esta figura se muestran las gráficas comparativas del reuso temporal encontrado en el binario *lift* entre los niveles de optimización O0 y O2 de cada uno de los compiladores. Las gráficas planteadas han sido diagramas de caja que representan la frecuencia de aparición (r) de cadenas cuya longitud (l) esta ilustrada en el eje Y, por otra parte en el eje X aparece el caso de prueba al que corresponde el diagrama de caja, siendo en este caso los niveles de optimización con los que se ha probado el compilador, toda la información sobre el formato de los resultados esta explicada en detalle en el anexo C. En la figura se puede apreciar que en el caso de *gcc* en el nivel de optimización O0 se consiguen cadenas de reuso temporal más largas, esto es debido a que cuando se aplica un nivel de optimización bajo (O0), el uso de la pila para almacenar variables temporales y los accesos a memoria replicados son más frecuentes que en niveles de optimización altos (O2). En el nivel de optimización O2 en cambio, se realizan transformaciones que permiten utilizar los registros de forma útil para evitar accesos a memoria innecesarios. Esta disminución en los accesos a



(a) Comparativa entre O0 y O2 en gcc (b) Comparativa entre O0 y O2 en clang

Figura 4.1: Comparativas entre niveles de optimización en el binario *lift*

memoria provoca que la frecuencia de aparición y longitud de las cadenas de reuso temporal se vea reducida. Esta comparativa puede verse en detalle para todo el banco de pruebas en el anexo E. Por otra parte, en la propia figura 4.1 se puede observar como en el caso de *clang* las cadenas de reuso temporal de mayor longitud se encuentran en el nivel de optimización O2. Esta diferencia es debida a que el nivel de optimización O2 de *clang* es más agresivo a la hora de aplicar transformaciones de bucles que el de *gcc*. Estas transformaciones provocan que accesos a vector en bucles (reuso espacial) se vean convertidas en accesos a valores escalares (reuso temporal) en clang, provocando que aparezcan más cadenas de reuso temporal y de mayor longitud. Por ejemplo, en el caso concreto de este binario en el nivel O2 *gcc* no ha aplicado ninguna transformación de bucles, en cambio *clang* ha aplicado desenrollado de bucles en nueve ocasiones y vectorización en cinco como se puede ver más en detalle en la tabla D.1. Además como se puede detalladamente en la tabla D.3, en *gcc* se encuentran 273 accesos a memoria que presentan reuso temporal por los 589 observados en *clang*. Si se estudia el banco de pruebas de forma general se puede observar que estas conclusiones se mantienen. Generalmente el nivel de optimización O0 presenta cadenas de reuso temporal más largas que el nivel de optimización O2, pero en el caso de *clang* se encuentran ejemplos dónde esta tendencia se invierte con más facilidad que en *gcc*.

En las figuras 4.2 y 4.3 se puede observar que para los binarios *binarysearch* y *g723_enc* en el nivel O0, la distribución de las cadenas de reuso temporal en ARMv7 ilustra que se encuentran cadenas más cortas que en las soluciones de x64. Se ha podido comprobar que esta diferencia ocurre porque en ciertos binarios el analizador es capaz de capturar las cadenas de reuso temporal provocadas por los accesos a pila

con más facilidad para x64. En concreto se ha podido calcular que aproximadamente esta diferencia ocurre en un 70 % de los binarios en los que había muestras de ARMv7 (22/30). En nivel de optimización O2 esta diferencia no se hace tan evidente, y las diferencias en la distribución del reuso temporal dependen más del equilibrio entre la similitud en las transformaciones aplicadas por el compilador y las ventajas que ofrece cada arquitectura a la hora de optimizar. Por este motivo, pueden encontrarse binarios donde el reuso temporal encontrado para ARMv7 se asemeja al encontrado por los dos compiladores en x64, como en *rijndael_dec* y *rijndael_enc*, binarios dónde la similitud más evidente sea con la solución de gcc, como en *fir2dim* o *complex_updates*, binarios donde la solución se aproxime más a la planteada por clang, como *fft* o *st*, y por último binarios donde la solución sea completamente diferente como en *statemate* y *jfdctint*.

4.2. Resultados en reuso espacial

Al igual que en [1], lo más destacable sobre el reuso espacial es la prácticamente nula detección que existe de este tipo de reuso cuando se aplica un nivel de optimización bajo (O0). Esto es debido a que en la mayoría de las ocasiones en el nivel de optimización O0 se mantienen las variables temporales entre iteraciones de un bucle utilizando la pila. Esto provoca que si se da el caso comentado en la sección 2.3.2, donde un store en una dirección de memoria desconocida durante el proceso de interpretación abstracta hace $\top$ todas las direcciones de memoria, se pierda la información sobre las variables temporales útiles para la indexación de vectores que han sido almacenadas en la pila, impidiendo así encontrar el patrón real de acceso al vector. Esta diferencia se ha podido apreciar investigando la traza de ejecución del binario *complex_updates*, donde en el bucle en que se encuentra reuso espacial para *gcc* se mantiene el iterador en un registro, pero en *clang* al mantenerse en la pila termina convirtiéndose en $\top$ en cierto punto de la ejecución. En general, se ha observado que *clang* en el nivel de optimización O0 opta más por el uso de variables temporales en la pila que *gcc*, quien sí es capaz de aprovechar los registros para esta labor en ciertas ocasiones, por este motivo se puede observar que se ha capturado reuso espacial un número mayor de veces cuando se ha utilizado *gcc*, en cinco binarios en el caso de *gcc* y uno en el de *clang*.

En el nivel de optimización O2, en cambio, sí que se consigue detectar un mayor número de cadenas de reuso espacial como se puede ver en las figuras 4.5 y 4.4, las gráficas en estas figuras tienen el mismo planteamiento que en el reuso temporal, pero se añade un degradado de color que ilustra el número de puntos con los que se ha creado cada diagrama de caja debido a que el reuso espacial presenta ejemplos donde se ha encontrado un número reducido de cadenas.

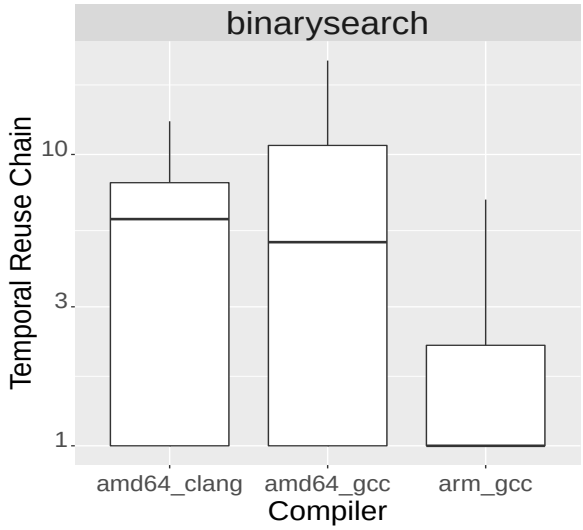


Figura 4.2: *binarysearch* O0

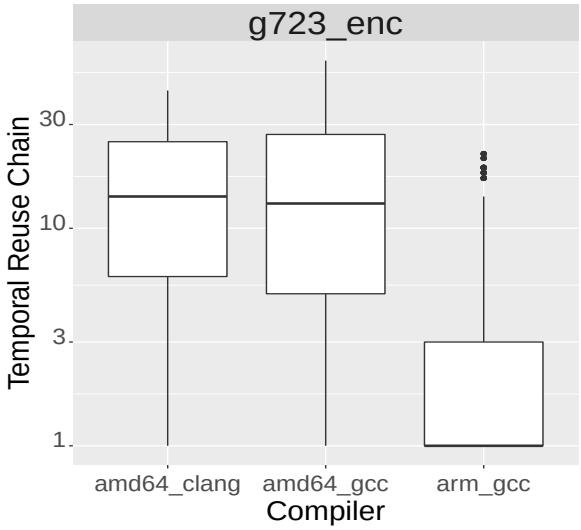


Figura 4.3: *g723_enc* O0

La mayor detección de reuso espacial se debe a que las variables útiles para la indexación de vectores se mantienen en los registros por las técnicas de optimización aplicadas. Si se observa la primera de las figuras se puede intuir la mayor agresividad de *clang* a la hora de aplicar transformaciones sobre los bucles atendiendo a los binarios *adpcm_enc* y *adpcm_dec*. En estos binarios las técnicas de optimización aplicadas por *clang* han eliminado la práctica totalidad de los bucles presentes en ellos ocasionando que con este compilador no se encuentre reuso espacial pero con *gcc* sí.

En esta figura también se muestra el caso del binario *pm*, en el cual, como ya se había visto en la sección 3.3, *clang* optimizaba el bucle de sus funciones pm_memcpy() y pm_memset() dividiéndolo en varios sub-bucles que recreaban la misma funcionalidad, esto como se observa en la figura hace que el reuso espacial encontrado en *clang* varíe por completo. Esto se debe a que se crean nuevos bucles con sus propias cadenas de reuso, las cuales suelen tener mayor longitud ya que esta optimización permite que en los sub-bucles se trabaje con tamaños de dato mayores que en el bucle original.

La segunda de las figuras permite observar que en un número no despreciable de casos los binarios generados por el compilador *gcc* presentan un reuso espacial muy similar, a pesar de las diferencias arquitecturales planteadas por x64, arquitectura con un conjunto de instrucciones complejo (CISC), y ARMv7, arquitectura con un conjunto de instrucciones sencillo (RISC). Este sería el caso de binarios como *matrix1*, *insertsort*, *isqrt*, *jfdctint* o *bsort* en un nivel de optimización O2. Estos ejemplos permiten observar como las transformaciones del compilador pueden salvar las diferencias arquitecturales sobre ciertos algoritmos para implementar soluciones con accesos a vectores equivalentes.

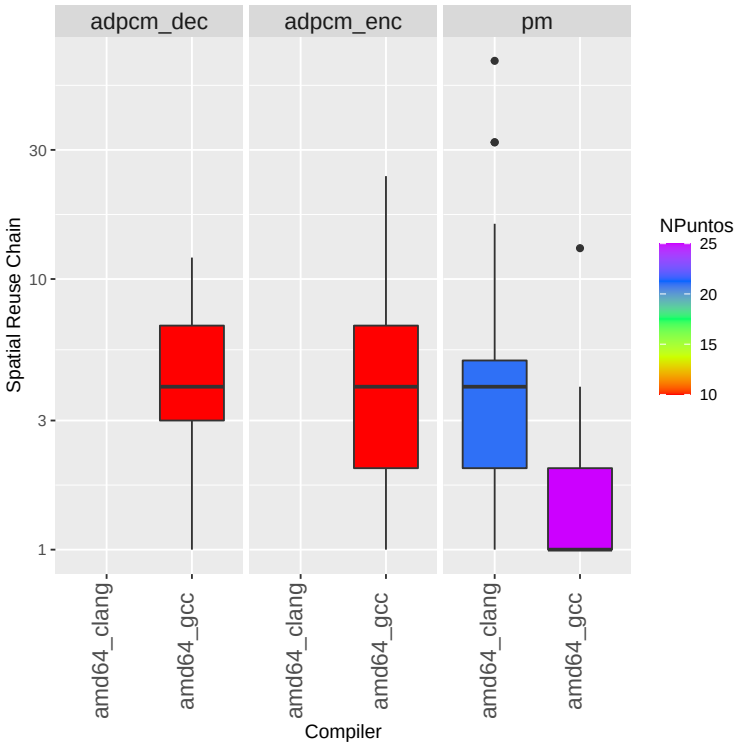


Figura 4.4: Reuso espacial: compiladores O2

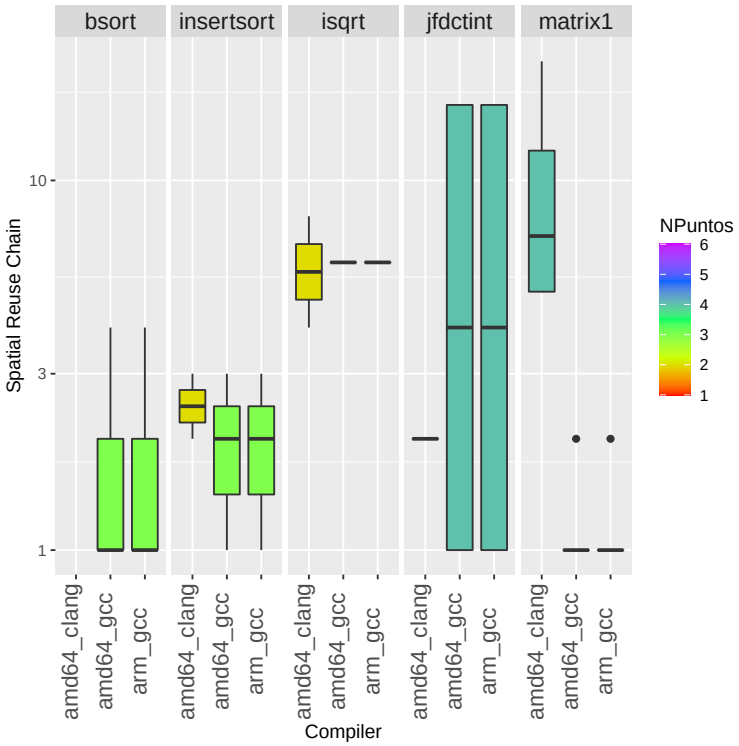


Figura 4.5: Reuso espacial: arquitecturas O2

Capítulo 5

Conclusiones y trabajo futuro

Durante el trabajo realizado se ha podido comprender la importancia de plantear un análisis seguro y preciso del reuso de datos en el contexto de aplicaciones de tiempo real. De esta forma se ha podido comprobar que en este ámbito, trabajar sobre el nivel más próximo a la ejecución real de las aplicaciones es fundamental para poder analizar su comportamiento de la forma más fiable, ya que como se ha podido ver en los resultados, la arquitectura sobre la que se ejecuta la aplicación y las características del compilador utilizado son elementos que pueden cambiar por completo el reuso de datos esperado si simplemente se plantea un análisis en alto nivel. Además la realización de este trabajo ha permitido verificar que la herramienta *polygaz* mantiene su capacidad de análisis del reuso de datos al aplicarla sobre otras arquitecturas, siempre que se le apliquen las extensiones correspondientes para ello.

Por último, tras la realización de este trabajo quedan dos vías posibles con las que ampliar la experimentación realizada durante el transcurso del mismo. En primer lugar, se podrían aprovechar el entorno desplegado, la información de la fase de inspección manual y los análisis desarrollados para proseguir en el estudio con un análisis de WCET, similar al realizado en [2]. En segundo lugar, se podría plantear ampliar la funcionalidad de la herramienta haciéndola capaz de analizar el reuso de datos presente en binarios compilados por otras arquitecturas. Por ejemplo, sería muy interesante plantear el análisis de la arquitectura RISC-V [17], la cual se postula como un agente importante en el futuro de la arquitectura de computadores por contar con un conjunto de instrucciones de desarrollo libre.

Capítulo 6

Bibliografía

[1] Juan Segarra, Jordi Cortadella, Rubén Gran Tejero, and Víctor Viñals-Yúfera. Automatic safe data reuse detection for the wcet analysis of systems with data caches. IEEE Access, 8:192379–192392, 2020.

[2] Juan Segarra, Rubén Gran Tejero, and Víctor Viñals. A generic framework to integrate data caches in the wcet analysis of real-time systems. Journal of Systems Architecture, 120:102304, 2021.

[3] The LLVM Foundation. Clang: a c language family frontend for llvm. `https://clang.llvm.org/`, 2003. Access Date: 10-2021.

[4] The GNU Foundation. Gcc, the gnu compiler collection. `https://gcc.gnu.org/`, 1986. Access Date: 10-2021.

[5] Marcel Erné, Jürgen Koslowski, Austin Melton, and George E Strecker. A primer on galois connections. Annals of the New York Academy of Sciences, 704(1):103–125, 1993.

[6] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In International Conference on Computer Aided Verification, pages 661–667. Springer, 2009.

[7] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, pages 30–44, 1991.

[8] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[9] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016.

[10] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[11] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.

[12] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, 16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016), volume 55 of OpenAccess Series in Informatics (OASIcs), pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[13] Angr Team. Angr/angr-management: The official angr gui. `https://github.com/angr/angr-management`, 2021. Access Date: 11-2021.

[14] Álvaro Moreno. x64-polygaz: A polygaz adaptation for data reuse anlysis over x64 architecture. `https://gitlab.com/m4515/x64-polygaz`, 2022. Access Date: 06-2022.

[15] Angr Team. Angr/archinfo: a collection of classes that contains architecture-specific information . `https://github.com/angr/archinfo`, 2021. Access Date: 12-2021.

[16] The R Foundation. The r project for statistical computing. `https://www.r-project.org/`, 2013. Access Date: 05-2022.

[17] David Patterson and Andrew Waterman. The RISC-V Reader: an open architecture Atlas. Strawberry Canyon, 2017.

[18] Podman community. Podman: Pod manager tool. `https://podman.io/getting-started/`, 2018. Access Date: 04-2022.

[19] OpenBSD. Tmux: terminal multiplexor. https://github.com/tmux/tmux/wiki, 2015. Access Date: 06-2022.

[20] ARM. ARM Architecture Reference Manual, pages 78–79. 2000. https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/third-party/ddi0100e-arm-arm.pdf.

[21] ARM. ARMv7-M Architecture Reference Manual, page 95. 2010. https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARMv7-M_ARM.pdf.

[22] Felix Cloutier. x86 and amd64 instruction reference:mul-unsigned multiply. https://www.felixcloutier.com/x86/mul, 2019. Access Date: 12-2021.

[23] Felix Cloutier. x86 and amd64 instruction reference:imul-signed multiply. https://www.felixcloutier.com/x86/imul, 2019. Access Date: 12-2021.

[24] Felix Cloutier. x86 and amd64 instruction reference:lea-load effective address. https://www.felixcloutier.com/x86/lea, 2019. Access Date: 12-2021.

[25] Felix Cloutier. x86 and amd64 instruction reference:div-unsigned divide. https://www.felixcloutier.com/x86/div.html, 2019. Access Date: 12-2021.

[26] Felix Cloutier. x86 and amd64 instruction reference:jump if condition is met. https://www.felixcloutier.com/x86/jcc.html, 2019. Access Date: 12-2021.

[27] Warren Cheung, William Evans, and Jeremy Moses. Predicated instructions for code compaction. In International Workshop on Software and Compilers for Embedded Systems, pages 17–32. Springer, 2003.

Siglas

Ca2 Complemento a 2. 17, 40

CFG Control Flow Graph. 9

gaZ grupo de arquitectura de computadores de Zaragoza. II, 1, 2

STRe Sistemas de tiempo real estricto. 1

WCET Worst Case Execution Time. II, 1, 3, 5, 9, 11, 12, 25

Glosario

ARMv7 Arquitectura de computadores RISC (Conjunto de instrucciones reducido) desarrollada por la empresa ARM Holdings. II, 1, 9, 11, 12, 20–23, 37, 40, 43, 44

LLVM Conjunto de herramientas que utilizan técnicas de representación intermedia (IR) independientes del lenguaje para el desarrollo de compiladores. . 11

x64 Hace referencia al repertorio básico de instrucciones de las arquitecturas de computadores CISC (Conjunto de instrucciones complejas) de 64 bits desarrolladas por Intel y AMD.. II, 1, 11, 15–17, 20–23, 37, 40, 43, 44

Anexos

Anexos A

Entorno Experimental

A.1. Entorno físico

Todo el trabajo de análisis ha sido desarrollado en las máquinas remotas nevado y atps-20, prestadas por la Universidad de Zaragoza y el grupo de arquitectura de Zaragoza. La primera fase del proyecto se desarrolló en nevado, pero puesto que la máquina era compartida con otros proyectos que requerían un uso amplio de la memoria RAM de la máquina, finalmente se decidió migrar el proyecto a atps-20. A continuación se probe una tabla de características de ambas máquinas:

Máquina	Procesador	Memoria RAM	Cores
Nevado	AMD Ryzen Threadripper 1920X	128 GB	12
Atps-20	AMD EPYC 7702P	256 GB	64

Tabla A.1: Tabla de características de las máquinas físicas utilizadas

A.2. Entorno virtual

A.2.1. Podman

Dado que en la máquina atps-20 existía el problema de que no se podían instalar paquetes adicionales a los ya existentes porque se estaba llevando a cabo un proyecto de investigación delicado en ella. Para recrear el entorno desplegado en nevado fue necesario utilizar la herramienta podman [18](versión 3.0.2) ya instalada previamente en la máquina. Con ella se creó un contenedor donde trabajar con normalidad en el entorno diseñado previamente.

A.2.2. Tmux

Para trabajar en remoto con mayor comodidad manteniendo la sesión remota entre puntos de trabajo, se ha utilizado la herramienta de multiplexación de terminales

tmux [19](versión 3.2) que ya estaba instalada en ambas máquinas.

A.3. Compiladores

A continuación se muestra una tabla con las versiones de los compiladores instalados, junto a las opciones empleadas durante la compilación para elegir el nivel de optimización y generar la información sobre transformaciones de bucles en O2.

Compilador	Versión	Optimización	Generación de información
Clang	10.0.0	-O0/-O2	-Rpass=unroll|vectorize
Gcc	9.4.0	-O0/-O2	-fopt-info-loop

Tabla A.2: Tabla de características de compiladores

A.4. Angr

Se ha instalado la versión 9.1.1223 a través de un entorno virtual de python siguiendo los siguientes pasos:

I. Clonar el repositorio polygaz con: git clone https://gitlab.com/uz−gaz/polygaz.git

II. Crear un entorno virtual de python en el directorio polygaz: virtualenv −−python=$(which python3) venv

III. desde el directorio poligaz ejecutar: ./venv/bin/pip3 install angr

A.5. Apron

Se ha instalado la versión 0.9.13 de apron a partir de sus fuentes siguiendo los siguientes pasos:

I. Descarga de los fuentes de apron desde su repositorio de github: git clone https://github.com/antoinemine/apron.git

II. Ejecutar en el directorio de apron los siguientes comandos: find . -name lib* -exec cp \{\} ∼/apron/lib \ y find . -name *.h -exec cp \{\} ∼/apron/include/ \.

III. Ejecutar make en el directorio de apron

IV. Cambiar ruta a la del entorno actual en el Makefile de polygaz y ejecutar make en polygaz.

A.6. TACLeBench

A continuación, se muestran tablas que describen cada uno de los programas de prueba que componen TACLeBench en el momento de realización de este trabajo:

Nombre	Descripción
binarysearch	Búsqueda binaria de 15 números enteros
bitcount	Cuenta número de bits en un array de enteros
bitonic	Algoritmo de ordenación bitónico
bsort	Algoritmo de ordenación en burbuja
complex_updates	Multiplicaciones y sumas de vectores complejos
cosf	Cálculo de la función coseno
countnegative	Cuenta valores negativos en una matriz
cubic	Resolución de polinomios cúbicos
deg2rad	Algoritmo de conversión de grados a radianes
fac	Cálculo de factoriales
fft	Cálculo de transformada rápida de Fourier de 1024 puntos
filterbank	Filtrado de señales de múltiples frecuencias
fir2dim	Filtro convolucional de respuesta finita al impulso en dos dimensiones
iir	Filtro iir bicuadrado de cuatro secciones
insertsort	Algoritmo de ordenación por inserción
isqrt	Cálculo de raíces cuadradas enteras
jfdcint	Transformación de coseno discreta sobre un bloque de píxels 8x8
lms	Algoritmo LMS de mejora adaptativa de señales
ludcmp	Aplica descomposición LU para resolución de sistemas de ecuaciones lineales
matrix1	Algoritmo genérico de multiplicación de matrices
md5	Algoritmo de generación de funciones hash md5
minver	Inversión de matrices con números en coma flotante
pm	Algoritmo de reconocimiento de patrones
prime	Cálculo de números primos
quicksort	Algoritmo de ordenación quicksort
rad2deg	Algoritmo de conversión de radianes a grados
recursion	Código recursivo artificial
sha	Algoritmo seguro de hash NIST
st	Cálculos estadísticos sobre dos vectores

Tabla A.3: TACLeBench kernel benchmarks

Nombre	Descripción
adpcm_dec	Simula decodificación en un APCM
adpcm_dec	Simula codificación en un APCM
ammunition	Prueba de estrés con operaciones aritméticas
anagram	Cálculo de anagramas dentro de una frase
audiobeam	Algoritmo para conversión de sonido en micrófonos
cjpeg_transupp	Funciones de translación y rotación usadas en compresión JPEG
cjpeg_wrbmp	Funciones de transformación de imágenes a formato Microsoft BMP
dijkstra	Algoritmo de dijkstra para encontrar el camino más corto en un grafo
epic	Algoritmo de compresión de imágenes (Efficient Image Pyramid Coder)
fmref	Software de radio FM con ecualizador
g723_enc	Algoritmo de compresión de voz en audio digital
gsm_dec	Algoritmo de decodificación de señales GSM
gsm_enc	Algoritmo de codificación de señales GSM
h264_dec	Simulación de decodificación en un códec H.264
huff_dec	Decodificación utilizando el método Huffman
huff_enc	Codificación utilizando el método Huffman
mpeg2	Codificación de audio y vídeo basado en el estándar MPGE2
ndes	Código empotrado complejo
petrinet	Simulación de una red de Petri
rijndael_dec	Descifrado simétrico de una clave cifrada con el algoritmo de Rijndael
rijndael_enc	Cifrado simétrico de una clave con el algoritmo de Rijndael
statemate	Simulación de un sistema de elevación de una ventanilla de coche.
susan	Algoritmo de reconocimiento de imágenes MR

Tabla A.4: TACLeBench sequential benchmarks

Nombre	Descripción
cover	Código artificial con muchos posibles caminos de ejecución
duff	Implementación de un dispositivo Duff
test3	Test de estrés para análisis WCET

Tabla A.5: TACLeBench test benchmarks

Nombre	Descripción
lift	Software real de control de un elevador
powerwindow	Software real de control de elevación de ventanillas de un coche

Tabla A.6: TACLeBench app benchmarks

Nombre	Descripción
Debie	Software real para observación de micro-meteoritos y residuos espaciales
PapaBench	Software de piloto automático y remoto para aviación autónoma
rosace	Implementación del caso de estudio ROSACE

Tabla A.7: TACLeBench parallel benchmarks

Anexos B

Modificaciones en la herramienta

B.1. Instrucciones añadidas

I. Operaciones aritméticas

Se han considerado como operaciones aritméticas aquellas que trabajan con valores enteros y representan una operación de suma o resta. Dentro de este grupo se ha añadido en la herramienta la funcionalidad de análisis de las siguientes operaciones:

i) Iop_Add64 Esta operación de 64 bits define una suma entre dos registros fuente o un registro fuente y un valor, siendo almacenada posteriormente en un registro destino. Para interpretar la operación se ha implementado en la herramienta la expresión $r_d = r_i + r_j$ o $r_d = r_i + k$, $k \in \mathbb{Z}$

ii) Iop_Sub64 Esta operación de 64 bits define una resta entre dos registros fuente o un registro fuente y un valor constante, siendo almacenada posteriormente en un registro destino. Para interpretar la operación se ha implementado en la herramienta la expresión $r_d = r_i - r_j$ o $r_d = r_i - k$, $k \in \mathbb{Z}$

II. Operaciones geométricas

Se han considerado como operaciones geométricas aquellas que trabajan con valores enteros y representan una multiplicación, una división o un desplazamiento de bits. Dentro de este grupo se ha añadido en la herramienta la funcionalidad de análisis de las siguientes operaciones:

i) Iop_Shl64

Esta operación de 64 bits desplaza hacia la izquierda el contenido de un registro fuente un valor constante de bits y lo almacena posteriormente en un registro destino. Para interpretar la operación se ha implementado en la herramienta la expresión $r_d = r_i \cdot 2^k$, $k \in (0, 1, ..., 63)$

II) Iop_Mul32

Esta operación de 32 bits ya existente en la versión original de la herramienta ha sido ampliada debido a las características concretas de la arquitectura x64. Originalmente esta instrucción había sido definida con la expresión $r_d = \top$ debido a que en ARMv7 [20] [21] la ejecución de las instrucciones de multiplicación siempre implican una expresión no lineal, es decir no representable en el dominio de los poliedros, al involucrar dos registros fuente. En x64 en cambio existen dos posibilidades: (1) la operación ha sido decodificada desde una instrucción *mul* [22] o *imul* [23] donde se realiza la multiplicación de dos registros y se resuelve igual que en ARMv7, es decir $r_d = \top$, o (2) la operación ha sido decodificada a partir de una instrucción *imul* [23] donde un registro es multiplicado por una constante, resolviéndose en este caso concreto con la expresión $r_d = r_i \cdot k \ , k \in \mathbb{Z}$.

III) Iop_Mul64

Esta operación de 64 bits se ha tratado de forma equivalente a la solución planteada para su homóloga de 32 bits Iop_Mul32.

IV) Iop_MullS32 e Iop_MullU32

Estas operaciones de 32 bits plantean otra forma de decodificación de *angr* para las instrucciones *imul* [23] y *mul* [22] respectivamente, pero como se ha comprobado experimentalmente solo son utilizadas en el caso de multiplicación entre dos registros. Por tanto se han resulto como $r_d = \top$.

V) Iop_Sar64 e Iop_Shr64

Estas operaciones de 64 bits representan el desplazamiento hacia la derecha de un valor constante de bits con y sin signo respectivamente. Ambas se han tratado siguiendo el mismo planteamiento conservador presente en las versiones de 32 bits existentes previamente en la herramienta. Por tanto se han resuelto con la expresión $r_d = \top$.

VI) Iop_DivModS64to32 e Iop_DivModU64to32

Estas operaciones de 64 bits representan la operación módulo con y sin signo respectivamente. Puesto que no es posible expresar el operador módulo linealmente utilizando únicamente sumas, restas o productos, ambas operaciones se han resuelto como $r_d = \top$.

III. Operaciones lógicas

Se han considerado como operaciones lógicas aquellas que trabajan con valores enteros y pertenecen al álgebra de Boole, es decir operaciones and,or,xor,etc.

Toda nueva operación perteneciente a esta categoría ha sido interpretada en la herramienta con la expresión $r_d = \top$, debido a que no son representables linealmente en el dominio de los poliedros del álgebra euclídea utilizando solo restas, sumas o productos.

IV. Operaciones en coma flotante

Se han considerado como operaciones en coma flotante aquellas que trabajan con valores en coma flotante. Toda nueva operación perteneciente a esta categoría ha sido clasificada en la herramienta con la expresión $r_d = \top$ ya que el valor producido en el registro destino de estas instrucciones estará en coma flotante, invalidando dicho valor para ser una dirección de memoria. Al no ser posible que se este tratando una dirección de memoria, esta información se descarta para ganar rendimiento en el análisis a pesar de que si que podría haber sido tratada.

V. Operaciones de comparación

Se han considerado como operaciones de comparación aquellas que trabajan con valores enteros y representan, como indica su nombre, una comparación entre dos valores que a su vez resulta en una restricción dentro del estado abstracto. Dentro de este grupo se ha añadido en la herramienta la funcionalidad de análisis de las siguientes operaciones:

i) Iop_CmpEQ64 e Iop_CmpEQ8

Estas operaciones de 64 y 8 bits respectivamente representan una comparación de igualdad entre dos registros, un registro y un valor constante o dos valores constantes. Esta comparación se ha modelado con la restricción $r_i - r_j = 0$.

ii) Iop_CmpNE64 e Iop_CmpNE8

Estas operaciones de 64 y 8 bits respectivamente representan una comparación de desigualdad entre dos registros, un registro y un valor constante o dos valores constantes. Esta comparación se ha modelado con la restricción $r_i - r_j \neq 0$.

iii) Iop_LE64U

Esta operación de 64 bits representa una comparación "menor o igual que" sin signo entre dos registros o un registro y un valor constante. Esta comparación se ha modelado con la restricción $-r_i + r_j \geq 0$.

iv) Iop_LT64S e Iop_LT64U

Estas operaciones de 64 bits representan una comparación "menor que" con y sin signo, respectivamente, entre dos registros o un registro y un valor constante. Esta comparación se ha modelado con la restricción $-r_i + r_j > 0$.

VI. Operaciones vectoriales

Se han considerado como operaciones vectoriales todas aquellas resultantes de la decodificación de una instrucción del repertorio vectorial de la arquitectura. Toda nueva operación perteneciente a esta categoría ha sido clasificada en la herramienta con la expresión $r_d = \top$. Debido a que estas operaciones implican la división de un registro en varios componentes para ser realizadas, impidiendo de esta manera que sea posible construir una expresión lineal en el dominio de los poliedros del álgebra euclídea. Descartar este tipo de operaciones no tiene un gran impacto sobre el análisis ya que se persigue la captura de direcciones de memoria (valores enteros), en lugar de el trabajo con cálculo vectorial.

VII. Operaciones de conversión

Se han considerado como operaciones de conversión todas aquellas que representan una transformación en el tamaño o tipo de un dato utilizado dentro de un registro. Dentro de este grupo se han expresado como $r_d = \top$ todas las operaciones que involucran reducción o extensión del número de bits útiles en el registro, por la imposibilidad de poder expresar estas transformaciones de forma lineal.

Excepcionalmente se ha interpretado la operación Iop_F64toI64S, la cual transforma el valor de 64 bits en coma flotante almacenado en un registro como un valor entero con signo también de 64 bits. En este caso se ha utilizado la expresión $r_d = r_i \cdot 1$ para representar el comportamiento correspondiente a la operación.

B.2. Definición de los errores corregidos

B.2.1. Desbordamiento de enteros en operaciones con resta

El error de desbordamiento era provocado por el tratamiento que se daba en la herramienta original a las operaciones en las que el modelado de sus funciones de transferencia implicaba definir una expresión con una resta, así como las peculiaridades de *angr* durante la decodificación. Más concretamente se puede plantear como ejemplo el comportamiento observado experimentalmente cuando se debía interpretar

la instrucción lea rax,[rbx + 1] [24]. Primero *angr* decodificaría la instrucción utilizando micro-operaciones básicas, dando lugar al siguiente lenguaje intermedio:

```
t1 = rbx; Variable temporal t1 = registro rbx
t2 = Sub64(t1,0xffffffffffffffff); Variable temporal t2 = t1 - (-1)
```

Al procesar la operación Iop_Sub64 la herramienta interpretaba originalmente el valor `0xffffffffffffffff` como el número natural $2^{64} - 1$ en lugar de como el número entero -1 en complemento a 2 (Ca2). Dado que en la herramienta, para modelar las funciones de transferencia basadas en una resta se realizaba una multiplicación del valor interpretado como natural por -1, se terminaba ocasionado un desbordamiento del valor al construirse la expresión.

Por ejemplo en el caso experimental primero se realizaba el cálculo `0xffffffffffffffff` $\cdot -1$ provocando que la expresión finalmente construida no fuese la presente originalmente en la mico-operación $t1+1$, si no $t1-(2^{64}-1)$. Es importante notar que este error había sobrevivido en la versión original de la herramienta, debido a que al trabajar con arquitectura ARMv7 de 32 bits el operando pasaría a tener un valor erróneo con tamaño de 33 bits, pero no sobrepasaría el tamaño máximo de un entero dentro del analizador. Lo cual si que ocurre al trabajar con una arquitectura de 64 bits, donde el paso a un valor con tamaño de 65 bits produce el desbordamiento de la variable y por tanto la finalización de la ejecución del analizador.

B.2.2. Error de ejecución en bloques básicos con 2 o más comparaciones

Este error se originaba al procesar un bloque básico con dos o más operaciones de comparación dentro de él. Originalmente, se había previsto en la herramienta que ante esta situación las restricciones aportadas por las operaciones de comparación fuesen descartadas de forma conservadora, con intención de evitar situaciones donde al encontrar múltiples comparaciones no se pudiese inferir cual de ellas marcaba la restricción de salida del bloque básico. Como durante el trabajo de análisis desarrollado con la herramienta original nunca se dio esta situación, había sobrevivido en ella un error de programación que ocasionaba que durante la interpretación abstracta, se intentase actualizar una restricción inexistente provocando el fin de la ejecución del análisis. Para solucionar este problema, debido a que los binarios reales (*fft*, *prime* y *cover*) donde se había observado consumían un tiempo de análisis considerable hasta llegar a ese punto, se creo un código simple en ensamblador x64 que permitiese acotar el error durante la ejecución del análisis. Este código mostrado en la figura B.1 está formado por tres bloques básicos y garantiza el caso de error debido a que *angr* resuelve

Figura B.1: Código ensamblador creado para corrección de errores.

```
section .text
  global main
main:
      prologue; push de los registros y establecimiento del frame pointer
      xor edx,edx
      mov eax,0x80
      mov ecx,0x2
bucle:
      mov rbx,0x1
      add rbx,0x1
      div ecx
      cmp eax,0x0
      jne bucle
      add rbx,0x3
      add rbx,0x4
      neg rbx
      xor rax,rax
      epilogue; pop de los registros y retorno del programa
```

la decodificación de las instrucciones div [25] y jne [26] incluyendo una micro-operación básica de comparación Iop_CmpEQ32 (ver más en detalle en anexos B.1 y B.2). Lo cual provoca que exista un bloque básico que incluye dos operaciones de comparación.

Después se ha descubierto, a través de la traza de ejecución del analizador sobre este código, por qué terminaba la ejecución de forma errónea al encontrar dos o más comparaciones en un bloque básico. Como se comentaba al principio del anterior párrafo, se estaban descartando las restricciones generadas por las operaciones de comparación, intentando añadir posteriormente al estado abstracto una restricción inexistente.

B.3. Decodificación de instrucciones de angr

Código B.1: Código de debug: Decodificación de instrucción div

```
div ecx:
t10 = rcx
t31 = 64to32(t10)
t9 = t31
t13 = rax
t32 = 64to32(t13)
t12 = t32
t15 = rdx
t33 = 64to32(t15)
t14 = t33
t11 = 32HLto64(t14,t12)
t45 = CmpEQ32(t9,0x0)
if (t45) {PUT{rip} = 0x401085; Ijk_SigFPE_IntDiv }
```

```
t5 = DivModU64to32(t11,t9)
t34 = 64to32(t5)
t17 = t34
t35 = 32Uto64(t17)
t16 = t35
PUT(rax) = t16
t36 = 64HIto32(t5)
t19 = t36
t37 = 32Uto64(t19)
t18 = t37
PUT(rdx) = t18
```

Código B.2: Código de debug: Decodificación de instrucción jne

```
jne 0x4017a:
t42 = 64to32(0x0)
t43 = 64to32(t22)
t41 = CmpEQ32(t43,t42)
t40 = 1Uto64(t41)
t29 = t40
t44 = 64to1(t29)
t24 = t44
if (t24) {PUT{rip} = 0x40108c; Ijk_Boring }
```

B.4. Inicialización de los registros

Registro	Id	Offset
rax	0	16
rcx	1	24
rdx	2	32
rbx	3	40
rsp	4	48
rbp	5	56
rsi	6	64
rdi	7	72
r8	8	80
r9	9	88
r10	10	96
r11	11	104
r12	12	112
r13	13	120
r14	14	128
r15	15	136

Tabla B.1: Valores de los registros

Anexos C

Formato de los resultados

C.1. Formato de los archivos xml

Un acceso a memoria puede estar definido dentro del archivo .xml con las etiquetas mostradas en la figura C.1.

La etiqueta *reference id* representa el identificador del acceso a memoria encontrado, *cfg_node* contiene el número que identifica el nodo dentro del CFG del programa en el que se encuentra el acceso a memoria, *pc* hace referencia a la dirección de memoria en la que se encuentra la instrucción que ocasiona el acceso en memoria, *number* es una etiqueta creada en la versión original para las instrucciones PUSH y POP múltiples de ARMv7 que representa el número de ese acceso con respecto al resto de accesos presentes dentro de la instrucción. Puesto que en la arquitectura x64 no existen instrucciones que se traten de forma similar, esta etiqueta tiene un valor fijo de 0 (primer acceso a memoria en la instrucción). La etiqueta *type* distingue si el acceso a memoria se realizó para obtener un dato, siendo una instrucción *load*, o en cambio se realizó para almacenarlo, siendo una instrucción *store*. *Predicated* es otra etiqueta

Figura C.1: Formato de archivo xml.

Código C.1: Código de debug: Decodificación de instrucción jne

```
<reference id=...>
<cfg_node>...</cfg_node>
<pc>...</pc>
<number>...</number>
<type>...</type>
<predicated>...</predicated>
<bytes>...<bytes>
<term i=... h=... />
<c>...</c>
<firstreused>...</firstreused>
<lastreused>...</lastreused>
```

pensada originalmente para aspectos de la arquitectura de ARMv7 e identifica si una instrucción está predicada [27] o no. En el caso de x64 esta etiqueta sencillamente marca "no" ya que en esta arquitectura no existe la predicación de instrucciones de memoria, es decir, no esta definida en el formato de instrucción la posibilidad de que estas instrucciones incluyan una condición de la que dependa su ejecución. *Bytes* simplemente indica el tamaño del acceso a memoria en bytes. *Term* es una etiqueta pensada para representar accesos secuenciales a un vector dentro de un bucle, indicando de esta manera la dirección de memoria correspondiente a la instrucción de inicio del bucle en la componente "i" y el incremento en bytes de la dirección accedida en cada iteración en su componente "h" (ver Sección 2.4). Con c se identifica la dirección del bloque de memoria sobre el que se realiza el acceso (ver Sección 2.4). Por último las etiquetas *firstreused* y *lastreused* aparecen cuando se ha detectado reuso sobre el acceso a memoria, haciendo referencia a la primera instrucción donde se detectó reuso sobre ese mismo acceso y la última respectivamente.

C.2. Formato de los gráficos y tablas

En el anexo E se muestran gráficamente los resultados obtenidos, partiendo del modelo de clasificación en tuplas visto en la sección 3.5.1. Aquí se pueden encontrar dos tipos de gráficas: (1) gráficas que presentan la comparativa en el reuso encontrado entre los pares arquitectura-compilador fijando un nivel de optimización concreto, y (2) gráficas que presentan la comparativa entre los niveles de optimización de un compilador concreto en la arquitectura x64. Cada una de las gráficas presenta un subconjunto del total de binarios utilizados, contando en su eje X con el par arquitectura-compilador o nivel de optimización que se ha analizado, y en su eje Y con la longitud de las cadenas de reuso encontradas, es decir, en el modelo de tuplas aplicado la componente l de cada tupla. Además mediante los diagramas de caja planteados para cada elemento, se pretende mostrar que cadenas de longitud l son las que más frecuentemente aparecen durante el análisis, quedando de esta manera representada la componente r definida en el modelo de tuplas.

En el anexo D se muestran las tablas construidas para aportar un mayor espectro de validación a los gráficos presentados. De esta manera, se incluyen las tablas D.2 y D.3 que miden el número de accesos a memoria que presentan reuso temporal encontrados en cada binario para cada arquitectura, compilador y nivel de optimización probados. Este valor se ha obtenido partiendo del total de tuplas obtenidas para cada solución y calculando, dadas k tuplas del tipo $\langle l, r \rangle$, la expresión $\sum_{i=1}^{k} l_i \cdot r_i$.

En el caso del reuso espacial se ha incluido la tabla D.5 que muestra el número de

puntos con los que ha sido formado cada diagrama de caja, es decir, la suma de todas las componentes r dentro del conjunto total de tuplas correspondiente a cada caso de análisis. Para este tipo de reuso se ha preferido utilizar esta información porque al ser menos frecuente el reuso espacial que el temporal, los diagramas de caja presentes en las gráficas han sido construidos con pocos puntos y se ha considerado relevante resaltar las diferencias de construcción entre cada uno de ellos. Además de en la tabla, también se ha resaltado esta diferencia mediante un degradado de color en las gráficas correspondientes (E.17, E.18, E.19, E.20, E.21, E.22, E.23, E.24).

Anexos D

Estadísticas de validación de resultados

Tabla D.1: Estadísticas de transformación de bucles en O2.

Binarios	Optimización	Desenrrollado	Vectorización
lift_gcc	O2	0	0
lift_clang	O2	9	5
powerwindow_gcc	O2	0	0
powerwindow_clang	O2	89	45
custom_gcc	O2	0	0
custom_clang	O2	0	0
fft_gcc	O2	0	0
fft_clang	O2	0	0
binarysearch_gcc	O2	0	0
binarysearch_clang	O2	0	0
bitcount_gcc	O2	0	0
bitcount_clang	O2	0	0
bitonic_gcc	O2	0	0
bitonic_clang	O2	1	16
bsort_gcc	O2	0	0
bsort_clang	O2	3	3
complex_updates_gcc	O2	0	0
complex_updates_clang	O2	4	2
cosf_gcc	O2	0	0
cosf_clang	O2	1	0
countnegative_gcc	O2	0	0
countnegative_clang	O2	6	3
cubic_gcc	O2	0	0
cubic_clang	O2	1	3
deg2rad_gcc	O2	0	0
deg2rad_clang	O2	0	0
fac_gcc	O2	0	0
fac_clang	O2	3	3
fft_gcc	O2	2	0

Binarios	Optimización	Desenrrollado	Vectorización
fft_clang	O2	10	7
filterbank_gcc	O2	0	0
filterbank_clang	O2	8	5
fir2dim_gcc	O2	2	0
fir2dim_clang	O2	15	12
iir_gcc	O2	2	0
iir_clang	O2	6	0
insertsort_gcc	O2	0	0
insertsort_clang	O2	1	2
isqrt_gcc	O2	0	0
isqrt_clang	O2	3	1
jfdctint_gcc	O2	0	0
jfdctint_clang	O2	6	4
lms_gcc	O2	0	0
lms_clang	O2	10	20
ludcmp_gcc	O2	0	0
ludcmp_clang	O2	6	0
matrix1_gcc	O2	0	0
matrix1_clang	O2	12	5
md5_gcc	O2	0	0
md5_clang	O2	24	31
minver_gcc	O2	8	0
minver_clang	O2	19	11
pm_gcc	O2	0	0
pm_clang	O2	27	26
prime_gcc	O2	0	0
prime_clang	O2	0	0
quicksort_gcc	O2	0	0
quicksort_clang	O2	5	1
rad2deg_gcc	O2	0	0
rad2deg_clang	O2	0	0
recursion_gcc	O2	0	0
recursion_clang	O2	0	0
sha_gcc	O2	0	0
sha_clang	O2	15	31
st_gcc	O2	0	0
st_clang	O2	7	0
adpcm_dec_gcc	O2	6	0
adpcm_dec_clang	O2	12	15
adpcm_enc_gcc	O2	6	0
adpcm_enc_clang	O2	13	23
ammunition_gcc	O2	136	0
ammunition_clang	O2	36	80
anagram_gcc	O2	0	0
anagram_clang	O2	17	22

Binarios	Optimización	Desenrrollado	Vectorización
audiobeam_gcc	O2	0	0
audiobeam_clang	O2	26	13
cjpeg_transupp_gcc	O2	6	0
cjpeg_transupp_clang	O2	38	11
cjpeg_wrbmp_gcc	O2	0	0
cjpeg_wrbmp_clang	O2	7	0
dijkstra_gcc	O2	0	0
dijkstra_clang	O2	3	0
epic_gcc	O2	0	0
epic_clang	O2	18	6
fmref_gcc	O2	0	0
fmref_clang	O2	12	6
g723_enc_gcc	O2	6	0
g723_enc_clang	O2	15	3
gsm_dec_gcc	O2	0	0
gsm_dec_clang	O2	14	10
gsm_enc_gcc	O2	0	0
gsm_enc_clang	O2	47	21
h264_dec_gcc	O2	0	0
h264_dec_clang	O2	8	4
huff_dec_gcc	O2	0	0
huff_dec_clang	O2	1	6
huff_enc_gcc	O2	0	0
huff_enc_clang	O2	24	9
mpeg2_gcc	O2	12	0
mpeg2_clang	O2	36	31
ndes_gcc	O2	2	0
ndes_clang	O2	6	22
petrinet_gcc	O2	4	0
petrinet_clang	O2	3	0
rijndael_dec_gcc	O2	0	0
rijndael_dec_clang	O2	2	4
rijndael_enc_gcc	O2	0	0
rijndael_enc_clang	O2	4	5
statemate_gcc	O2	0	0
statemate_clang	O2	2	0
susan_gcc	O2	4	0
susan_clang	O2	22	12
cover_gcc	O2	0	0
cover_clang	O2	1	0
duff_gcc	O2	0	0
duff_clang	O2	8	5
test3_gcc	O2	0	0
test3_clang	O2	129	0

Tabla D.2: Valores de validación de reuso temporal en O0.

Binario	Optimización	Arquitectura y Compilador	Accesos a memoria
adpcm_dec	0	amd64_gcc	1602
adpcm_dec	0	amd64_clang	1676
adpcm_enc	0	amd64_gcc	2018
adpcm_enc	0	amd64_clang	2216
audiobeam	0	amd64_gcc	4686
audiobeam	0	arm_gcc	2318
audiobeam	0	amd64_clang	5034
binarysearch	0	amd64_gcc	160
binarysearch	0	arm_gcc	132
binarysearch	0	amd64_clang	137
bsort	0	amd64_gcc	154
bsort	0	arm_gcc	148
bsort	0	amd64_clang	152
cjepg_transupp	0	amd64_gcc	2220
cjepg_transupp	0	amd64_clang	2452
cjpeg_transupp	0	arm_gcc	2638
cjpeg_wrbmp	0	amd64_gcc	1081
cjpeg_wrbmp	0	amd64_clang	1131
complex_updates	0	amd64_gcc	162
complex_updates	0	arm_gcc	164
complex_updates	0	amd64_clang	183
cosf	0	amd64_gcc	1191
cosf	0	arm_gcc	1150
cosf	0	amd64_clang	1260
countnegative	0	amd64_gcc	120
countnegative	0	arm_gcc	123
countnegative	0	amd64_clang	178
cover	0	amd64_gcc	191
cover	0	arm_gcc	153
cover	0	amd64_clang	183
cubic	0	amd64_gcc	33366
cubic	0	amd64_clang	35713
deg2rad	0	amd64_gcc	40
deg2rad	0	arm_gcc	51
deg2rad	0	amd64_clang	54
dijkstra	0	amd64_gcc	458
dijkstra	0	arm_gcc	390
dijkstra	0	amd64_clang	464
duff	0	amd64_gcc	145
duff	0	amd64_clang	142
fft	0	amd64_gcc	324
fft	0	arm_gcc	309
fft	0	amd64_clang	523
filterbank	0	amd64_gcc	231

Binario	Optimización	Arquitectura y Compilador	Accesos a memoria
filterbank	0	arm_gcc	289
filterbank	0	amd64_clang	332
fir2dim	0	amd64_gcc	423
fir2dim	0	arm_gcc	358
fir2dim	0	amd64_clang	677
fmref	0	amd64_gcc	3749
fmref	0	amd64_clang	3869
g723_enc	0	amd64_gcc	6116
g723_enc	0	arm_gcc	1861
g723_enc	0	amd64_clang	5760
gsm_dec	0	amd64_gcc	5510
gsm_dec	0	arm_gcc	4118
gsm_dec	0	amd64_clang	5736
h264_dec	0	amd64_gcc	2826
h264_dec	0	amd64_clang	7144
huff_dec	0	amd64_gcc	1165
huff_dec	0	amd64_clang	1216
iir	0	amd64_gcc	145
iir	0	arm_gcc	165
iir	0	amd64_clang	191
insertsort	0	amd64_gcc	149
insertsort	0	arm_gcc	221
insertsort	0	amd64_clang	144
isqrt	0	amd64_gcc	218
isqrt	0	arm_gcc	183
isqrt	0	amd64_clang	222
jfdctint	0	amd64_gcc	345
jfdctint	0	arm_gcc	371
jfdctint	0	amd64_clang	365
lift	0	amd64_gcc	768
lift	0	amd64_clang	773
ludcmp	0	amd64_gcc	440
ludcmp	0	arm_gcc	614
ludcmp	0	amd64_clang	538
matrix1	0	amd64_gcc	119
matrix1	0	arm_gcc	140
matrix1	0	amd64_clang	149
minver	0	amd64_gcc	804
minver	0	arm_gcc	925
minver	0	amd64_clang	736
ndes	0	amd64_clang	1195
ndes	0	amd64_gcc	1367
petrinet	0	amd64_gcc	1116
petrinet	0	arm_gcc	2491
petrinet	0	amd64_clang	1126

Binario	Optimización	Arquitectura y Compilador	Accesos a memoria
pm	0	amd64_gcc	5122
pm	0	amd64_clang	4826
powerwindow	0	amd64_gcc	13061
powerwindow	0	arm_gcc	7038
powerwindow	0	amd64_clang	16213
prime	0	amd64_gcc	229
prime	0	amd64_clang	289
rad2deg	0	amd64_gcc	40
rad2deg	0	arm_gcc	52
rad2deg	0	amd64_clang	54
rijndael_dec	0	amd64_gcc	2590
rijndael_dec	0	arm_gcc	2968
rijndael_dec	0	amd64_clang	2863
rijndael_enc	0	amd64_gcc	2550
rijndael_enc	0	arm_gcc	3002
rijndael_enc	0	amd64_clang	2632
sha	0	amd64_gcc	2448
sha	0	amd64_clang	2687
st	0	amd64_gcc	1064
st	0	arm_gcc	484
st	0	amd64_clang	965
statemate	0	amd64_gcc	1525
statemate	0	arm_gcc	3423
statemate	0	amd64_clang	1656
susan	0	amd64_gcc	14624
susan	0	amd64_clang	19462

Tabla D.3: Valores de validación de reuso temporal en O2.

Binario	Optimización	Arquitectura y Compilador	Accesos a memoria
adpcm_dec	2	amd64_gcc	404
adpcm_dec	2	amd64_clang	695
adpcm_enc	2	amd64_gcc	446
adpcm_enc	2	amd64_clang	1118
audiobeam	2	amd64_gcc	976
audiobeam	2	arm_gcc	725
audiobeam	2	amd64_clang	857
binarysearch	2	amd64_gcc	15
binarysearch	2	arm_gcc	33
binarysearch	2	amd64_clang	13
bsort	2	amd64_gcc	11
bsort	2	arm_gcc	34
bsort	2	amd64_clang	57
cjepg_transupp	2	amd64_gcc	410
cjepg_transupp	2	amd64_clang	1404

Binario	Optimización	Arquitectura y Compilador	Accesos a memoria
cjpeg_transupp	2	arm_gcc	548
cjpeg_wrbmp	2	amd64_gcc	236
cjpeg_wrbmp	2	amd64_clang	260
complex_updates	2	amd64_gcc	42
complex_updates	2	arm_gcc	83
complex_updates	2	amd64_clang	142
cosf	2	amd64_gcc	197
cosf	2	arm_gcc	238
cosf	2	amd64_clang	241
countnegative	2	amd64_gcc	20
countnegative	2	arm_gcc	63
countnegative	2	amd64_clang	19
cover	2	amd64_gcc	27
cover	2	arm_gcc	20
cover	2	amd64_clang	19
cubic	2	amd64_gcc	3877
cubic	2	amd64_clang	4456
deg2rad	2	amd64_gcc	6
deg2rad	2	arm_gcc	12
deg2rad	2	amd64_clang	9
dijkstra	2	amd64_gcc	104
dijkstra	2	arm_gcc	208
dijkstra	2	amd64_clang	130
duff	2	amd64_gcc	25
duff	2	amd64_clang	50
fft	2	amd64_gcc	93
fft	2	arm_gcc	122
fft	2	amd64_clang	93
filterbank	2	amd64_gcc	107
filterbank	2	arm_gcc	81
filterbank	2	amd64_clang	14343
fir2dim	2	amd64_gcc	58
fir2dim	2	arm_gcc	114
fir2dim	2	amd64_clang	192
fmref	2	amd64_gcc	1207
fmref	2	amd64_clang	1188
g723_enc	2	amd64_gcc	1375
g723_enc	2	arm_gcc	1495
g723_enc	2	amd64_clang	1242
gsm_dec	2	amd64_gcc	596
gsm_dec	2	arm_gcc	715
gsm_dec	2	amd64_clang	580
h264_dec	2	amd64_gcc	205
h264_dec	2	amd64_clang	218
huff_dec	2	amd64_gcc	295

Binario	Optimización	Arquitectura y Compilador	Accesos a memoria
huff_dec	2	amd64_clang	169
iir	2	amd64_gcc	36
iir	2	arm_gcc	60
iir	2	amd64_clang	176
insertsort	2	amd64_gcc	53
insertsort	2	arm_gcc	115
insertsort	2	amd64_clang	43
isqrt	2	amd64_gcc	48
isqrt	2	arm_gcc	98
isqrt	2	amd64_clang	95
jfdctint	2	amd64_gcc	48
jfdctint	2	arm_gcc	108
jfdctint	2	amd64_clang	229
lift	2	amd64_gcc	273
lift	2	amd64_clang	589
lms	2	arm_gcc	115
ludcmp	2	amd64_gcc	83
ludcmp	2	arm_gcc	122
ludcmp	2	amd64_clang	166
matrix1	2	amd64_gcc	16
matrix1	2	arm_gcc	46
matrix1	2	amd64_clang	120
minver	2	amd64_gcc	95
minver	2	arm_gcc	216
minver	2	amd64_clang	355
ndes	2	amd64_gcc	283
ndes	2	amd64_clang	643
petrinet	2	amd64_gcc	793
petrinet	2	arm_gcc	634
petrinet	2	amd64_clang	750
pm	2	amd64_gcc	911
pm	2	amd64_clang	2249
powerwindow	2	amd64_gcc	4913
powerwindow	2	arm_gcc	2283
powerwindow	2	amd64_clang	5672
prime	2	amd64_gcc	37
prime	2	amd64_clang	22
rad2deg	2	amd64_gcc	6
rad2deg	2	arm_gcc	12
rad2deg	2	amd64_clang	9
rijndael_dec	2	amd64_gcc	999
rijndael_dec	2	arm_gcc	1348
rijndael_dec	2	amd64_clang	1198
rijndael_enc	2	amd64_gcc	1032
rijndael_enc	2	arm_gcc	1510

Binario	Optimización	Arquitectura y Compilador	Accesos a memoria
rijndael_enc	2	amd64_clang	912
sha	2	amd64_gcc	368
sha	2	amd64_clang	840
st	2	amd64_gcc	102
st	2	arm_gcc	150
st	2	amd64_clang	98
statemate	2	amd64_gcc	1007
statemate	2	arm_gcc	2142
statemate	2	amd64_clang	1197

Tabla D.4: Número de puntos en gráficas de reuso espacial en O0.

Binario	Optimización	Arquitectura y Compilador	Número de puntos
adpcm_dec	0	amd64_gcc	0
adpcm_dec	0	amd64_clang	0
adpcm_enc	0	amd64_gcc	0
adpcm_enc	0	amd64_clang	0
audiobeam	0	amd64_gcc	0
audiobeam	0	arm_gcc	1
audiobeam	0	amd64_clang	0
binarysearch	0	amd64_gcc	0
binarysearch	0	arm_gcc	0
binarysearch	0	amd64_clang	0
bsort	0	amd64_gcc	0
bsort	0	arm_gcc	0
bsort	0	amd64_clang	0
cjpeg_transupp	0	amd64_gcc	0
cjpeg_transupp	0	amd64_clang	0
cjpeg_transupp	0	arm_gcc	0
cjpeg_wrbmp	0	amd64_gcc	0
cjpeg_wrbmp	0	amd64_clang	0
complex_updates	0	amd64_gcc	1
complex_updates	0	arm_gcc	1
complex_updates	0	amd64_clang	0
cosf	0	amd64_gcc	0
cosf	0	arm_gcc	0
cosf	0	amd64_clang	0
countnegative	0	amd64_gcc	0
countnegative	0	arm_gcc	0
countnegative	0	amd64_clang	0
cover	0	amd64_gcc	0
cover	0	arm_gcc	0
cover	0	amd64_clang	0
cubic	0	amd64_gcc	0
cubic	0	amd64_clang	0

Binario	Optimización	Arquitectura y Compilador	Número de puntos
deg2rad	0	amd64_gcc	0
deg2rad	0	arm_gcc	0
deg2rad	0	amd64_clang	0
dijkstra	0	amd64_gcc	0
dijkstra	0	arm_gcc	0
dijkstra	0	amd64_clang	0
duff	0	amd64_gcc	0
duff	0	amd64_clang	0
fft	0	amd64_gcc	1
fft	0	arm_gcc	2
fft	0	amd64_clang	0
filterbank	0	amd64_gcc	0
filterbank	0	arm_gcc	2
filterbank	0	amd64_clang	0
fir2dim	0	amd64_gcc	4
fir2dim	0	arm_gcc	4
fir2dim	0	amd64_clang	0
fmref	0	amd64_gcc	0
fmref	0	amd64_clang	0
g723_enc	0	amd64_gcc	0
g723_enc	0	arm_gcc	1
g723_enc	0	amd64_clang	0
gsm_dec	0	amd64_gcc	0
gsm_dec	0	arm_gcc	0
gsm_dec	0	amd64_clang	0
h264_dec	0	amd64_gcc	0
h264_dec	0	amd64_clang	0
huff_dec	0	amd64_gcc	0
huff_dec	0	amd64_clang	0
iir	0	amd64_gcc	1
iir	0	arm_gcc	0
iir	0	amd64_clang	0
insertsort	0	amd64_gcc	0
insertsort	0	arm_gcc	0
insertsort	0	amd64_clang	0
isqrt	0	amd64_gcc	0
isqrt	0	arm_gcc	1
isqrt	0	amd64_clang	0
jfdctint	0	amd64_gcc	0
jfdctint	0	arm_gcc	0
jfdctint	0	amd64_clang	0
lift	0	amd64_gcc	0
lift	0	amd64_clang	0
ludcmp	0	amd64_gcc	0
ludcmp	0	arm_gcc	0

Binario	Optimización	Arquitectura y Compilador	Número de puntos
ludcmp	0	amd64_clang	0
matrix1	0	amd64_gcc	2
matrix1	0	arm_gcc	2
matrix1	0	amd64_clang	0
minver	0	amd64_gcc	0
minver	0	arm_gcc	0
minver	0	amd64_clang	0
ndes	0	amd64_gcc	0
ndes	0	amd64_clang	0
petrinet	0	amd64_gcc	0
petrinet	0	arm_gcc	0
petrinet	0	amd64_clang	0
pm	0	amd64_gcc	0
pm	0	amd64_clang	0
powerwindow	0	amd64_gcc	0
powerwindow	0	arm_gcc	0
powerwindow	0	amd64_clang	1
prime	0	amd64_gcc	0
prime	0	amd64_clang	0
rad2deg	0	amd64_gcc	0
rad2deg	0	arm_gcc	0
rad2deg	0	amd64_clang	0
rijndael_dec	0	amd64_gcc	0
rijndael_dec	0	arm_gcc	0
rijndael_dec	0	amd64_clang	0
rijndael_enc	0	amd64_gcc	0
rijndael_enc	0	arm_gcc	0
rijndael_enc	0	amd64_clang	0
sha	0	amd64_gcc	0
sha	0	amd64_clang	0
st	0	amd64_gcc	0
st	0	arm_gcc	3
st	0	amd64_clang	0
statemate	0	amd64_gcc	0
statemate	0	arm_gcc	0
statemate	0	amd64_clang	0
susan	0	amd64_gcc	0
susan	0	amd64_clang	0

Tabla D.5: Número de puntos en gráficas de reuso espacial en O2.

Binario	Optimización	Arquitectura y Compilador	Número de puntos
adpcm_dec	2	amd64_gcc	10
adpcm_dec	2	amd64_clang	0
adpcm_enc	2	amd64_gcc	10

Binario	Optimización	Arquitectura y Compilador	Número de puntos
adpcm_enc	2	amd64_clang	0
audiobeam	2	amd64_gcc	15
audiobeam	2	arm_gcc	17
audiobeam	2	amd64_clang	4
binarysearch	2	amd64_gcc	1
binarysearch	2	arm_gcc	1
binarysearch	2	amd64_clang	1
bsort	2	amd64_gcc	3
bsort	2	arm_gcc	3
bsort	2	amd64_clang	0
cjpeg_transupp	2	amd64_gcc	22
cjpeg_transupp	2	amd64_clang	16
cjpeg_transupp	2	arm_gcc	24
cjpeg_wrbmp	2	amd64_gcc	3
cjpeg_wrbmp	2	amd64_clang	6
complex_updates	2	amd64_gcc	4
complex_updates	2	arm_gcc	4
complex_updates	2	amd64_clang	2
cosf	2	amd64_gcc	0
cosf	2	arm_gcc	0
cosf	2	amd64_clang	0
countnegative	2	amd64_gcc	2
countnegative	2	arm_gcc	2
countnegative	2	amd64_clang	1
cover	2	amd64_gcc	0
cover	2	arm_gcc	0
cover	2	amd64_clang	0
cubic	2	amd64_gcc	0
cubic	2	amd64_clang	5
deg2rad	2	amd64_gcc	0
deg2rad	2	arm_gcc	0
deg2rad	2	amd64_clang	0
dijkstra	2	amd64_gcc	2
dijkstra	2	arm_gcc	3
dijkstra	2	amd64_clang	2
duff	2	amd64_gcc	3
duff	2	amd64_clang	1
fft	2	amd64_gcc	6
fft	2	arm_gcc	5
fft	2	amd64_clang	4
filterbank	2	amd64_gcc	8
filterbank	2	arm_gcc	10
filterbank	2	amd64_clang	1
fir2dim	2	amd64_gcc	15
fir2dim	2	arm_gcc	15

Binario	Optimización	Arquitectura y Compilador	Número de puntos
fir2dim	2	amd64_clang	3
fmref	2	amd64_gcc	8
fmref	2	amd64_clang	13
g723_enc	2	amd64_gcc	5
g723_enc	2	arm_gcc	10
g723_enc	2	amd64_clang	2
gsm_dec	2	amd64_gcc	12
gsm_dec	2	arm_gcc	16
gsm_dec	2	amd64_clang	11
h264_dec	2	amd64_gcc	3
h264_dec	2	amd64_clang	3
huff_dec	2	amd64_gcc	6
huff_dec	2	amd64_clang	9
iir	2	amd64_gcc	3
iir	2	arm_gcc	3
iir	2	amd64_clang	1
insertsort	2	amd64_gcc	3
insertsort	2	arm_gcc	3
insertsort	2	amd64_clang	2
isqrt	2	amd64_gcc	1
isqrt	2	arm_gcc	1
isqrt	2	amd64_clang	2
jfdctint	2	amd64_gcc	4
jfdctint	2	arm_gcc	4
jfdctint	2	amd64_clang	1
lift	2	amd64_gcc	7
lift	2	amd64_clang	2
lms	2	arm_gcc	6
ludcmp	2	amd64_gcc	8
ludcmp	2	arm_gcc	12
ludcmp	2	amd64_clang	6
matrix1	2	amd64_gcc	6
matrix1	2	arm_gcc	6
matrix1	2	amd64_clang	4
minver	2	amd64_gcc	13
minver	2	arm_gcc	14
minver	2	amd64_clang	2
ndes	2	amd64_gcc	7
ndes	2	amd64_clang	24
petrinet	2	amd64_gcc	0
petrinet	2	arm_gcc	1
petrinet	2	amd64_clang	0
pm	2	amd64_gcc	25
pm	2	amd64_clang	21
powerwindow	2	amd64_gcc	21

Binario	Optimización	Arquitectura y Compilador	Número de puntos
powerwindow	2	arm_gcc	22
powerwindow	2	amd64_clang	24
prime	2	amd64_gcc	0
prime	2	amd64_clang	0
rad2deg	2	amd64_gcc	0
rad2deg	2	arm_gcc	0
rad2deg	2	amd64_clang	0
rijndael_dec	2	amd64_gcc	7
rijndael_dec	2	arm_gcc	8
rijndael_dec	2	amd64_clang	7
rijndael_enc	2	amd64_gcc	9
rijndael_enc	2	arm_gcc	10
rijndael_enc	2	amd64_clang	9
sha	2	amd64_gcc	14
sha	2	amd64_clang	6
st	2	amd64_gcc	4
st	2	arm_gcc	6
st	2	amd64_clang	4
statemate	2	amd64_gcc	1
statemate	2	arm_gcc	0
statemate	2	amd64_clang	1

Anexos E

Gráficas de resultados

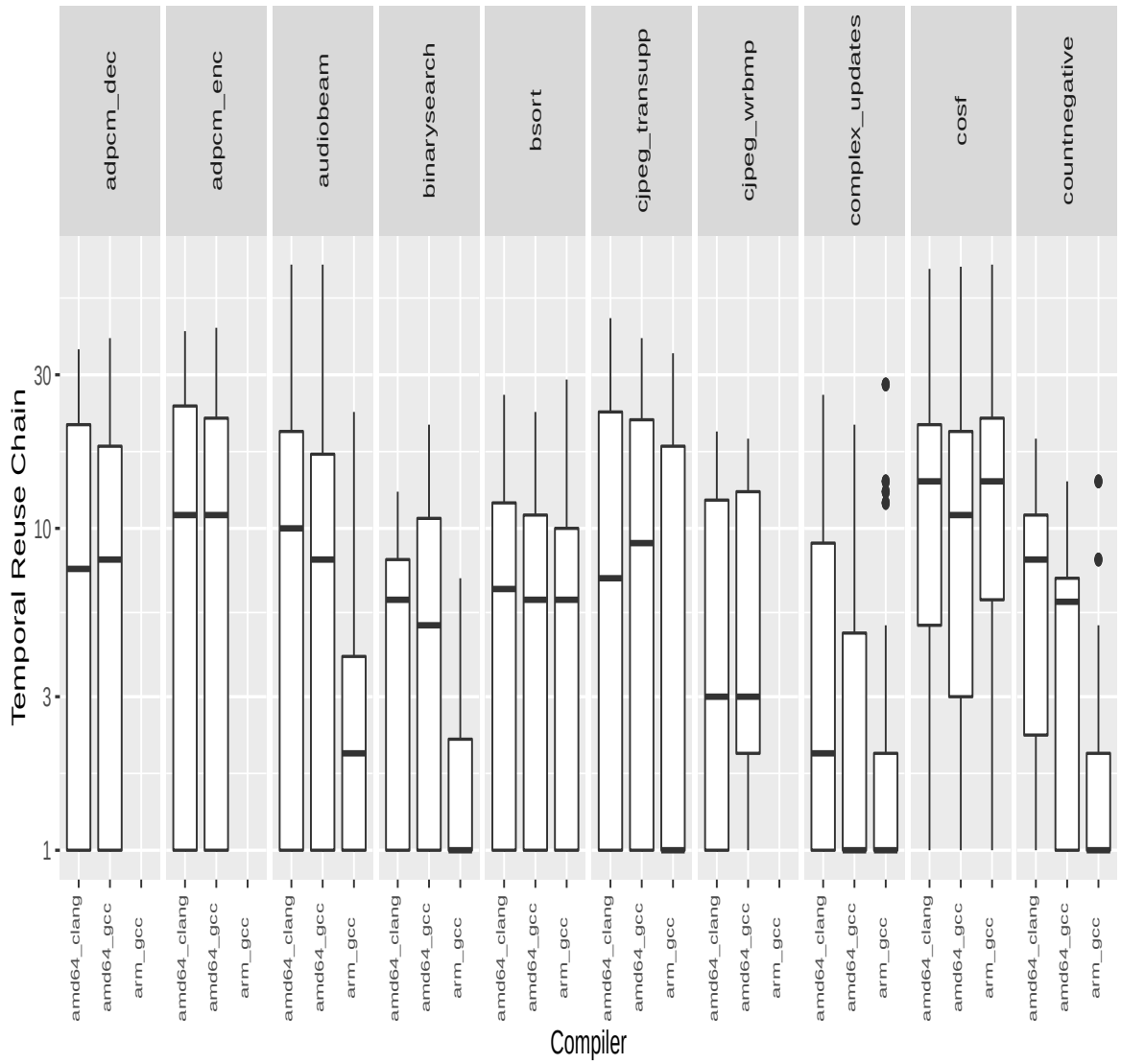


Figura E.1: Parte 1-Gráficas de reuso temporal en 00.

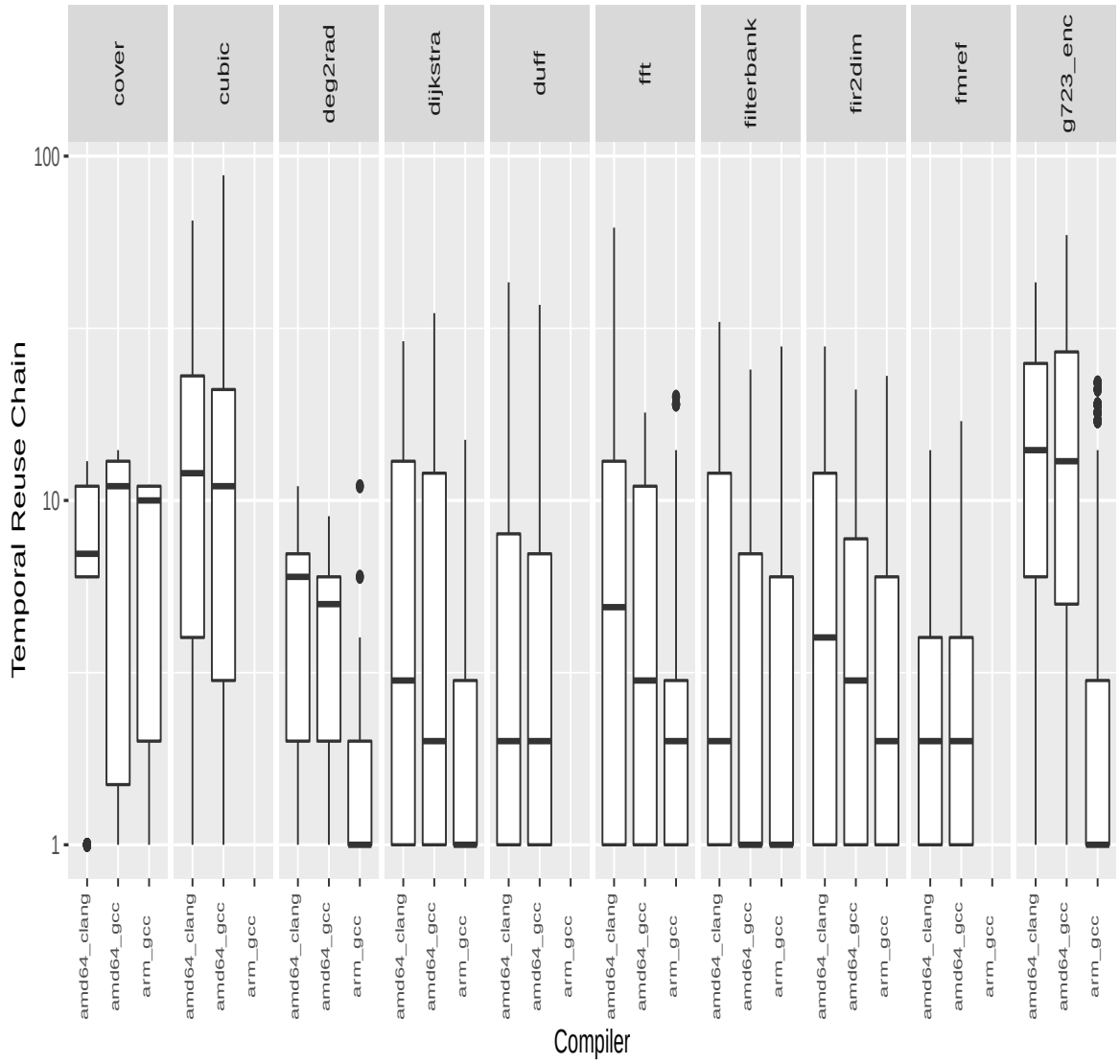


Figura E.2: Parte 2-Gráficas de reuso temporal en 00.

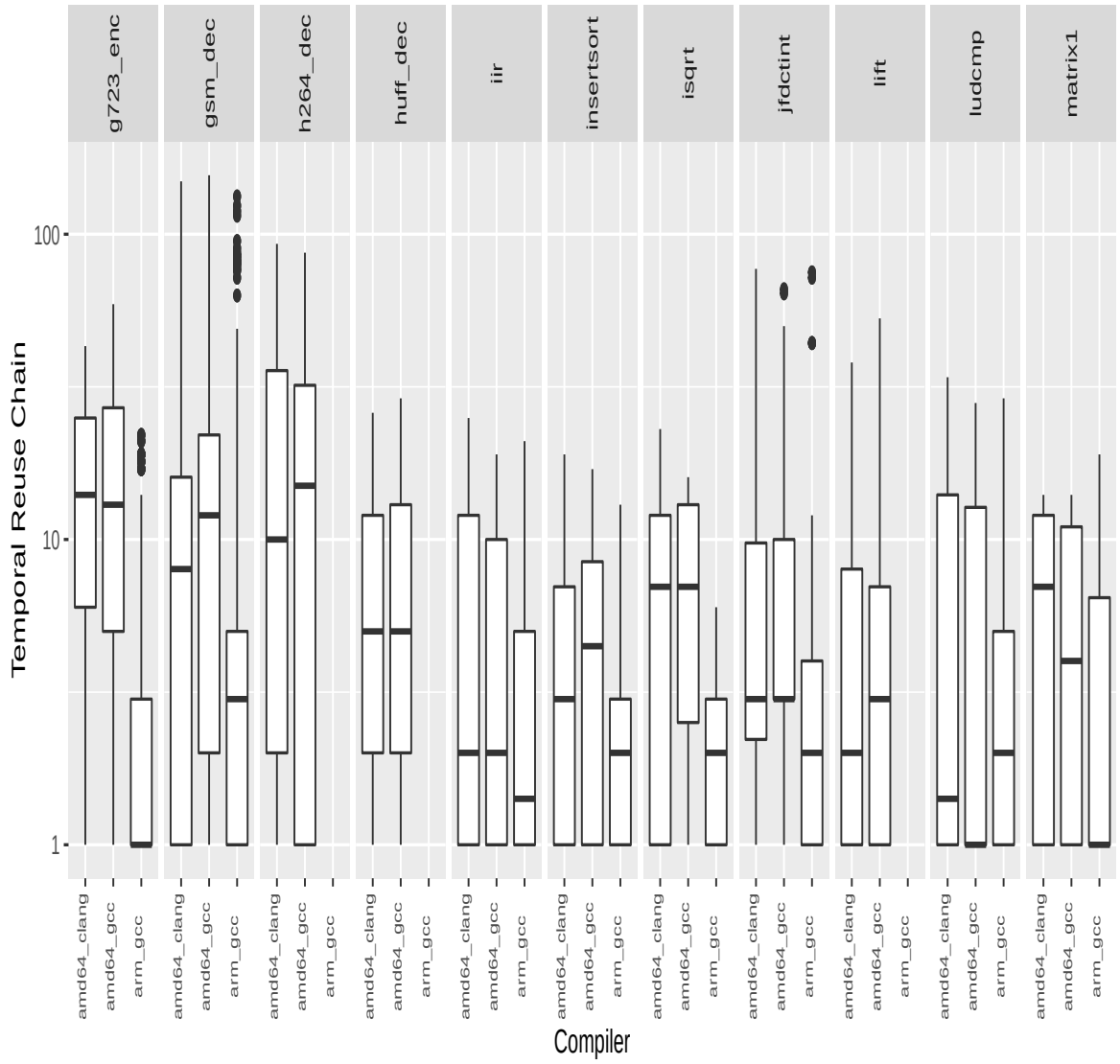


Figura E.3: Parte 3-Gráficas de reuso temporal en 00.

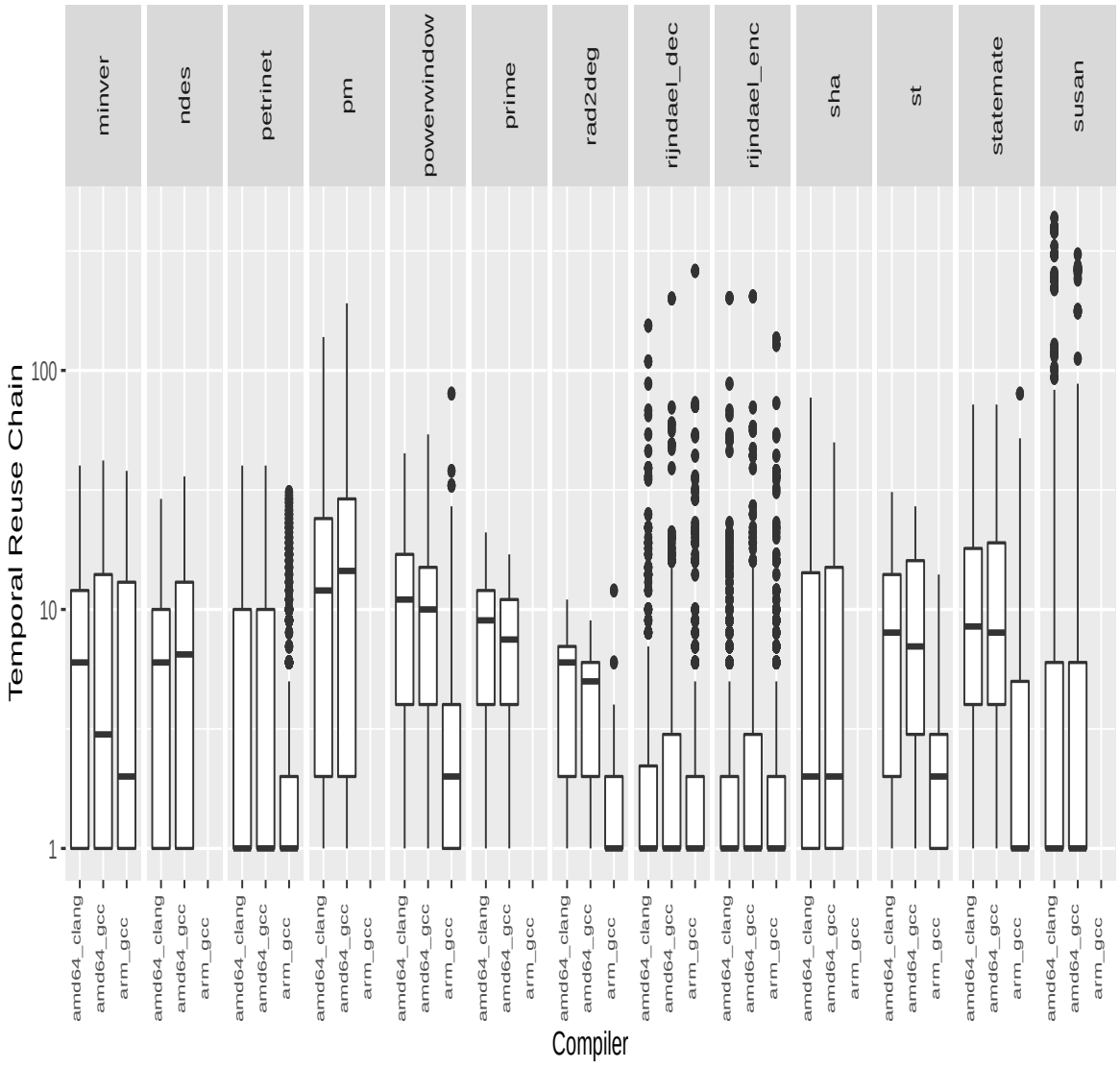


Figura E.4: Parte 4-Gráficas de reuso temporal en 00.

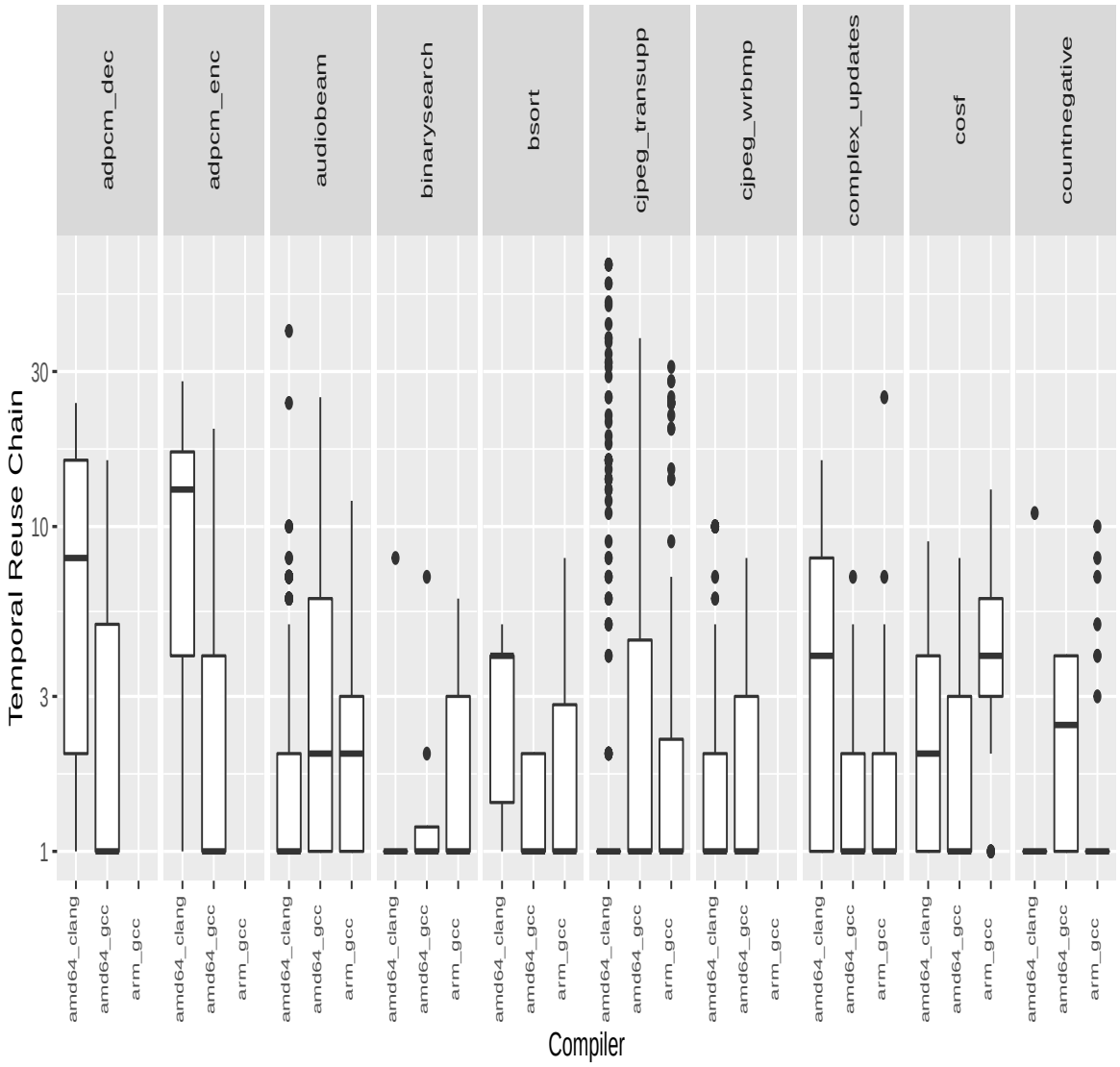


Figura E.5: Parte 1-Gráficas de reuso temporal en 02.

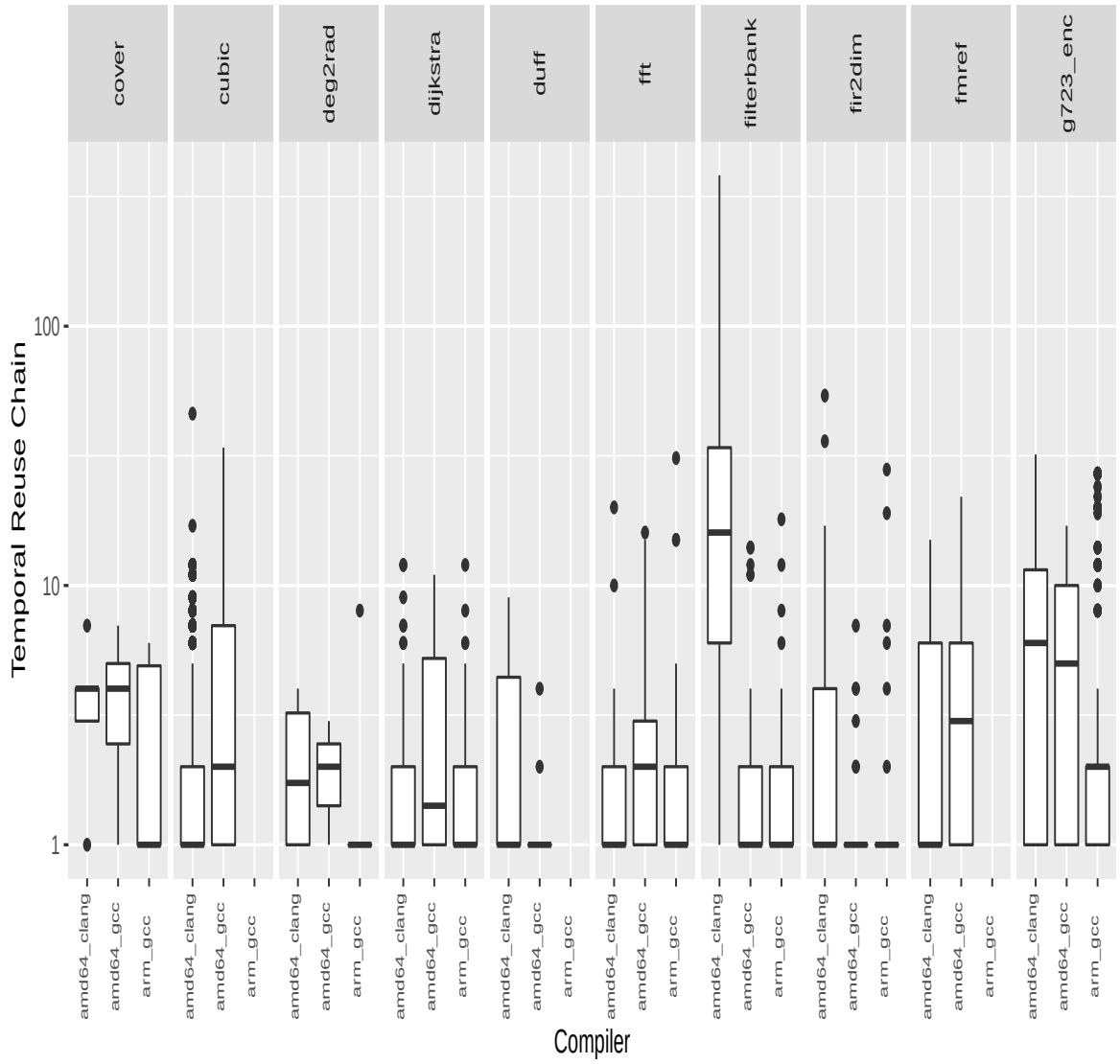


Figura E.6: Parte 2-Gráficas de reuso temporal en 02.

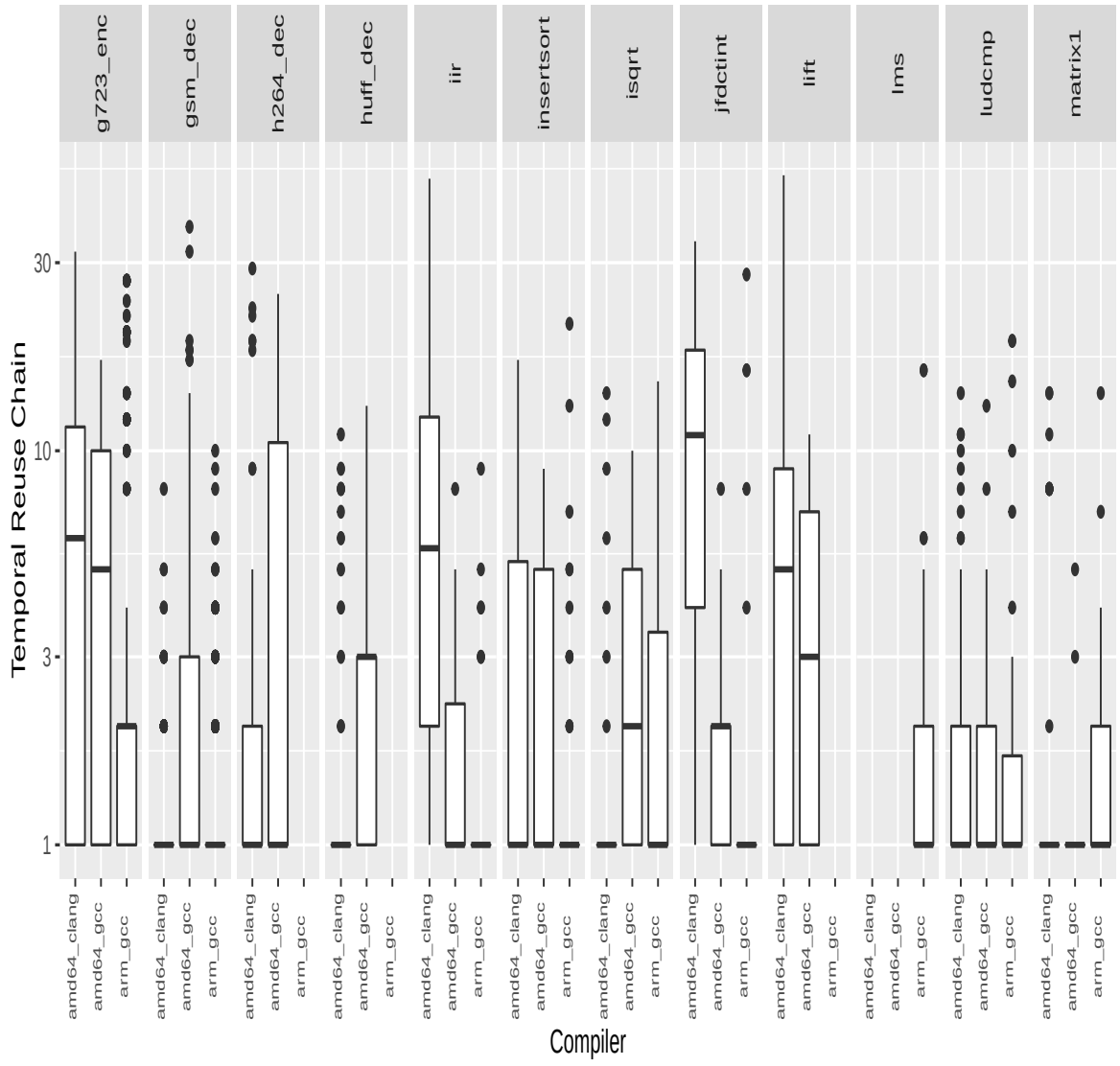


Figura E.7: Parte 3-Gráficas de reuso temporal en O2.

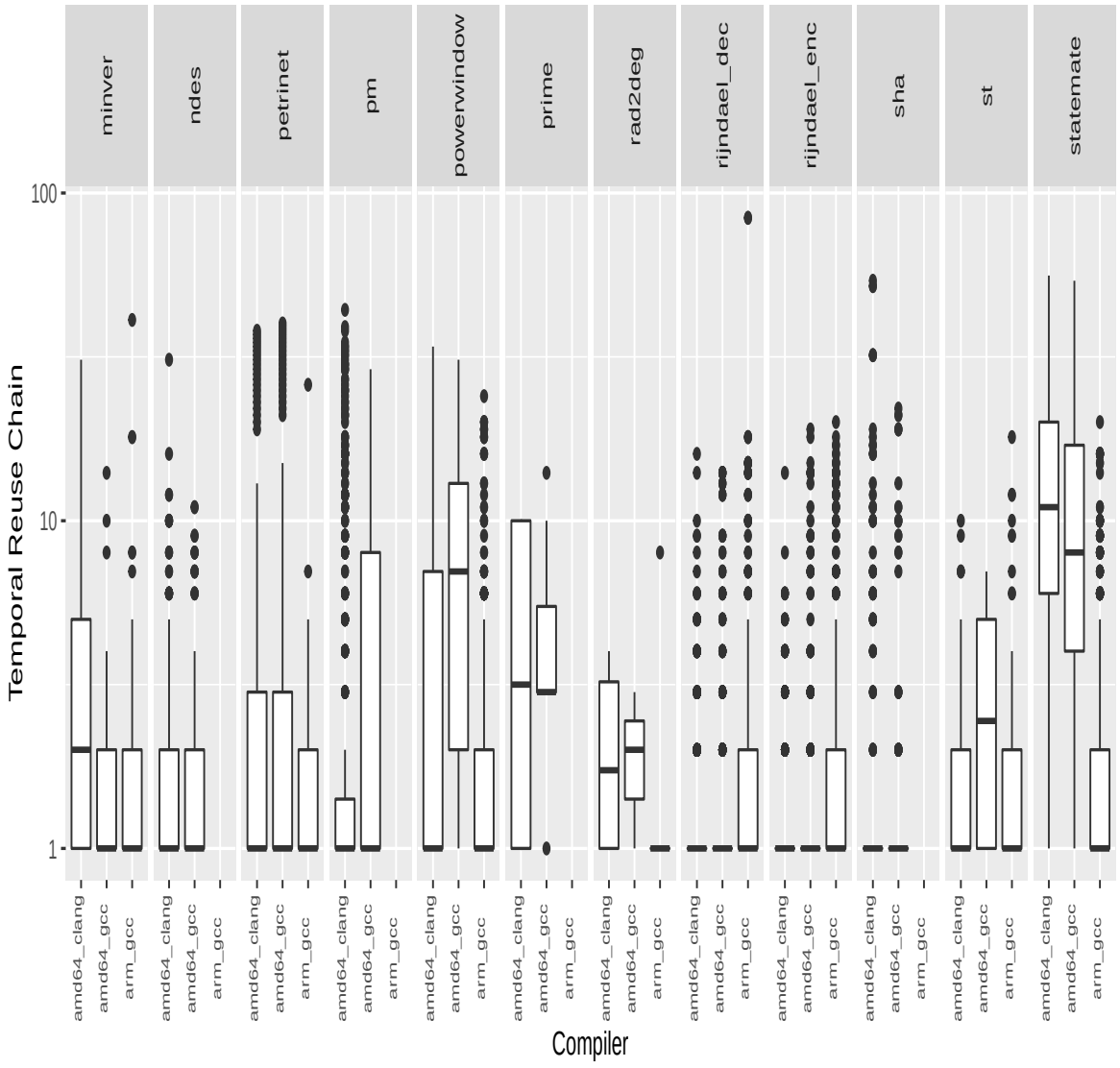


Figura E.8: Parte 4-Gráficas de reuso temporal en 02.

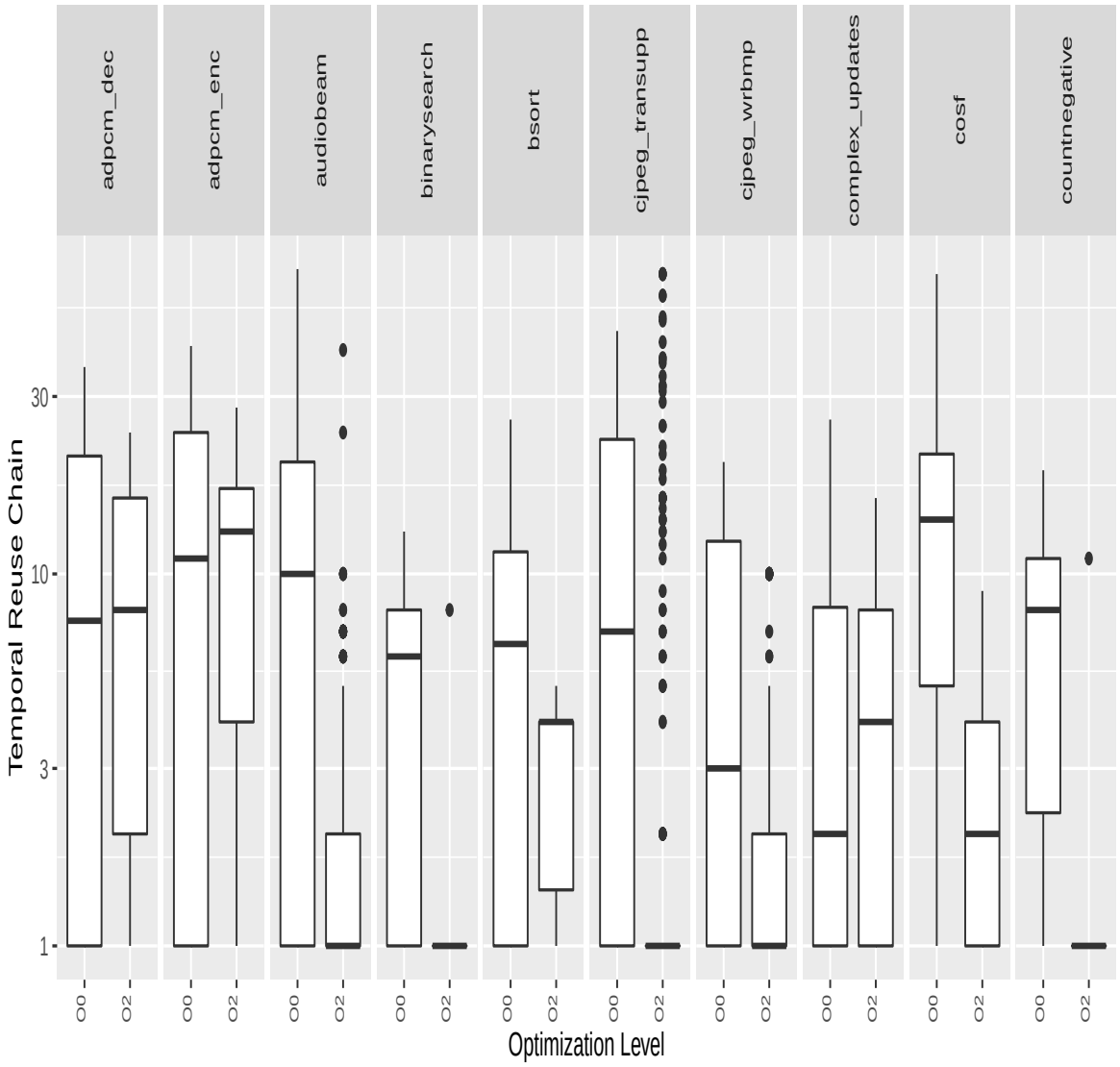


Figura E.9: Parte 1-Gráficas de reuso temporal comparativas entre Clang O0 y O2.

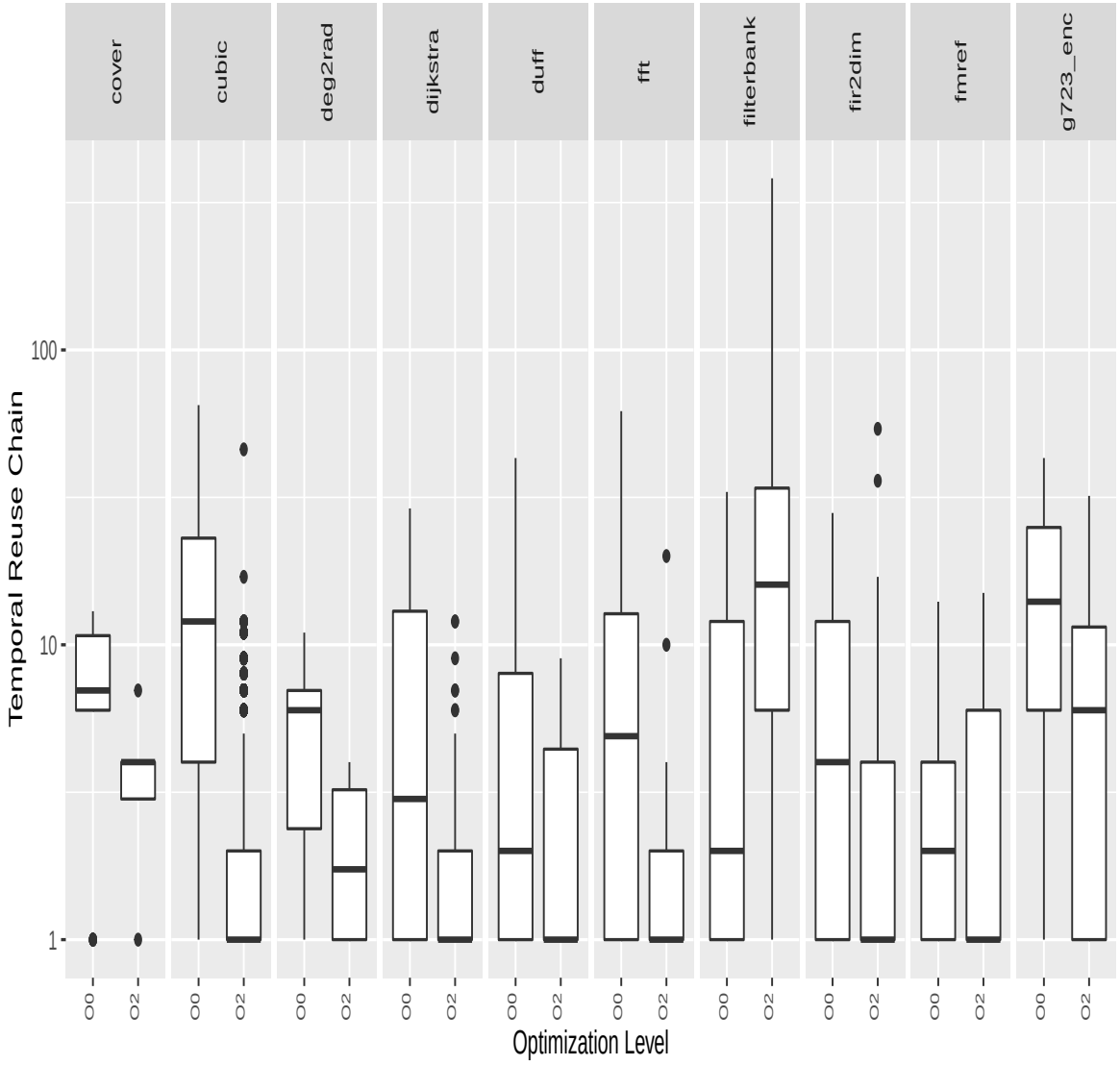


Figura E.10: Parte 2-Gráficas de reuso temporal comparativas entre Clang O0 y 02.

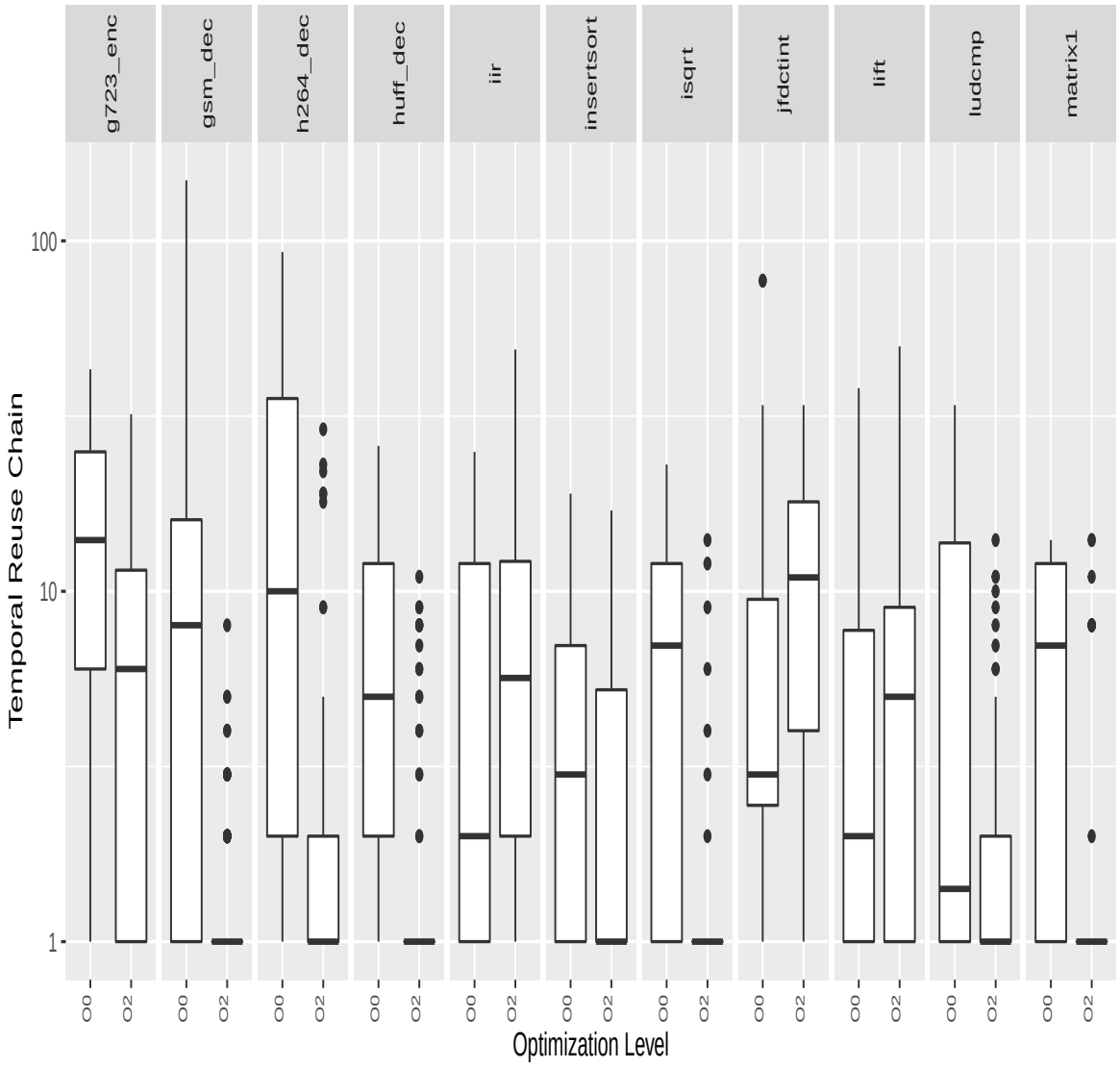


Figura E.11: Parte 3-Gráficas de reuso temporal comparativas entre Clang O0 y O2.

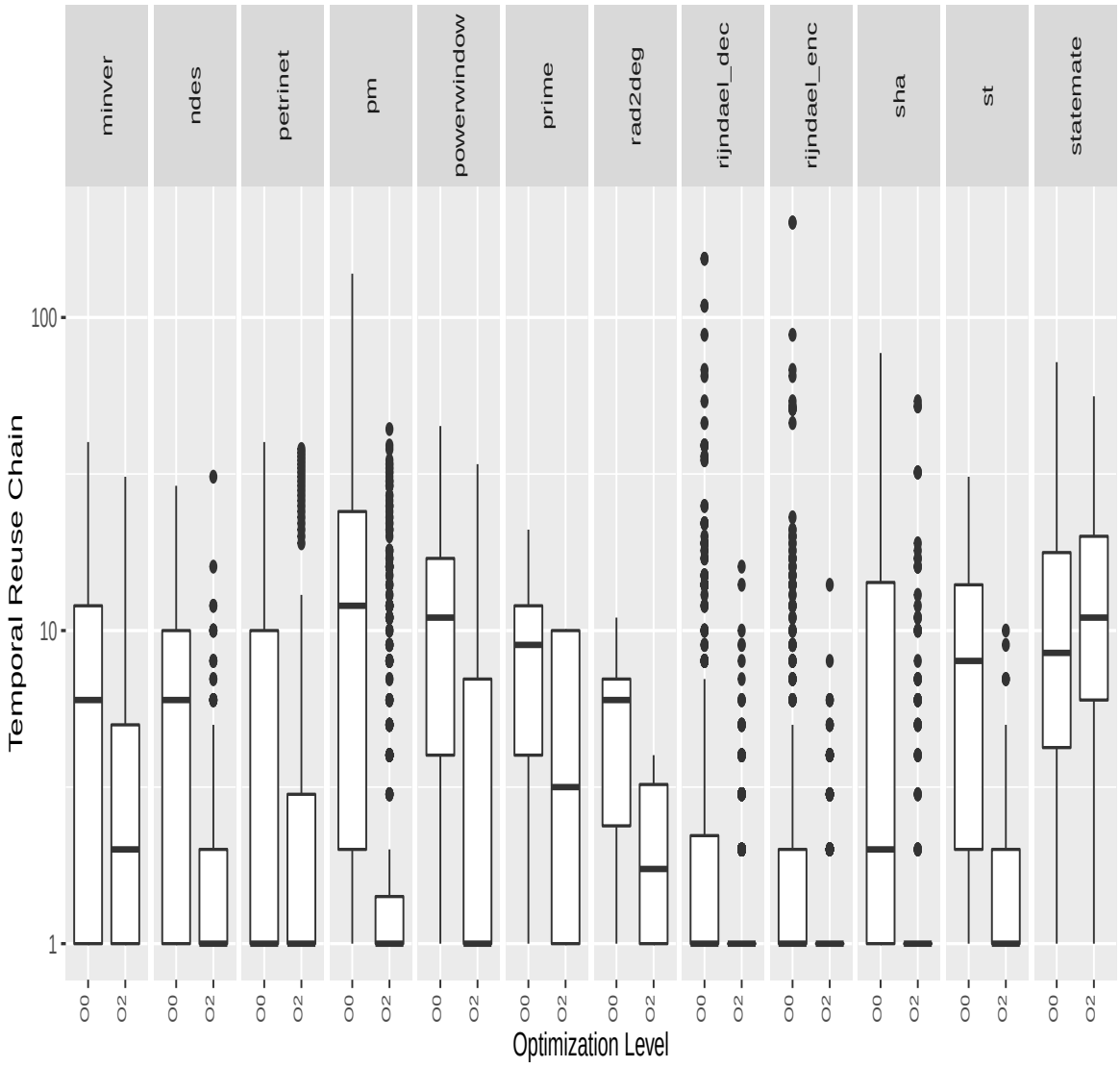


Figura E.12: Parte 4-Gráficas de reuso temporal comparativas entre Clang O0 y 02.

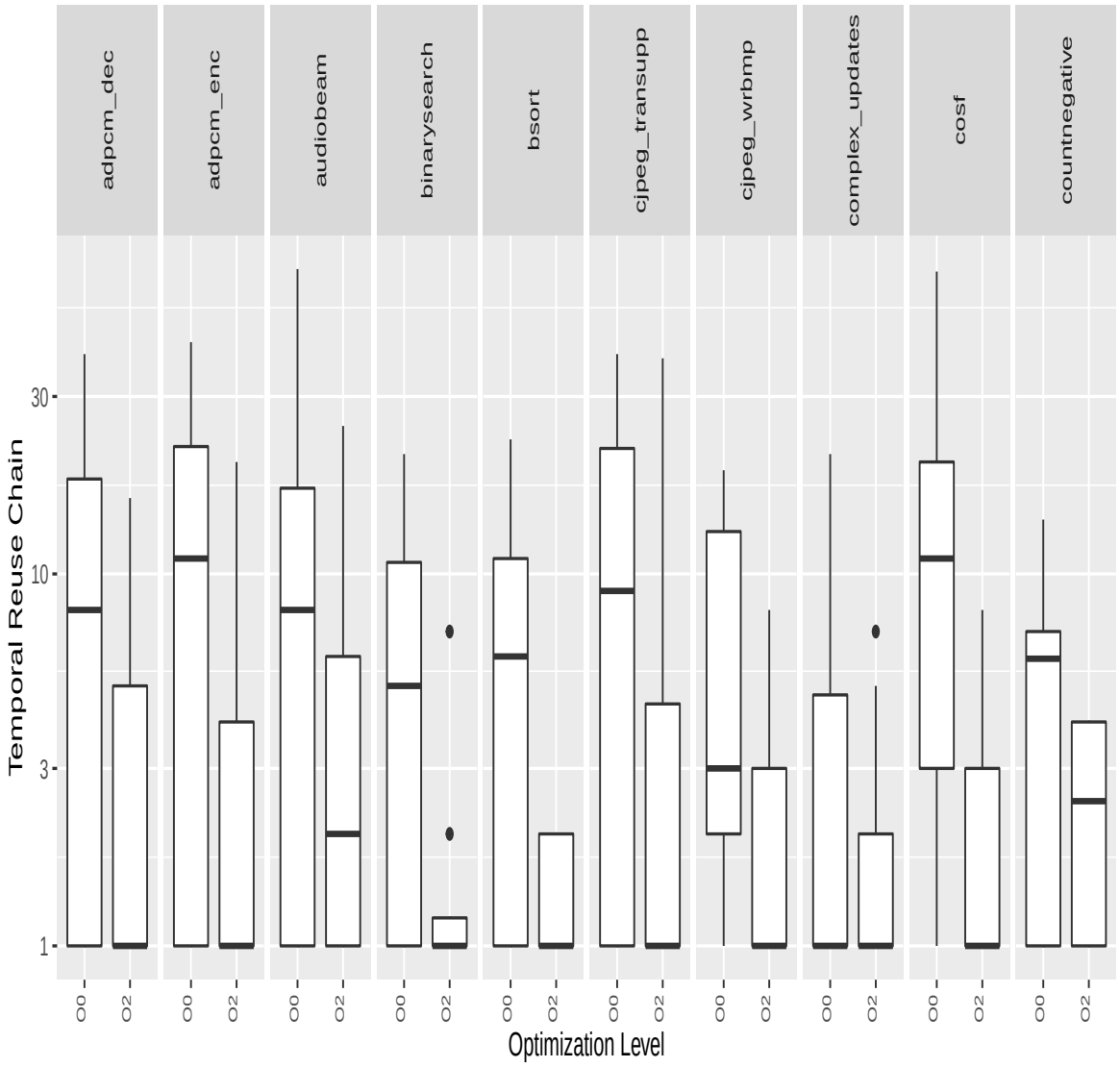


Figura E.13: Parte 1-Gráficas de reuso temporal comparativas entre GCC O0 y O2.

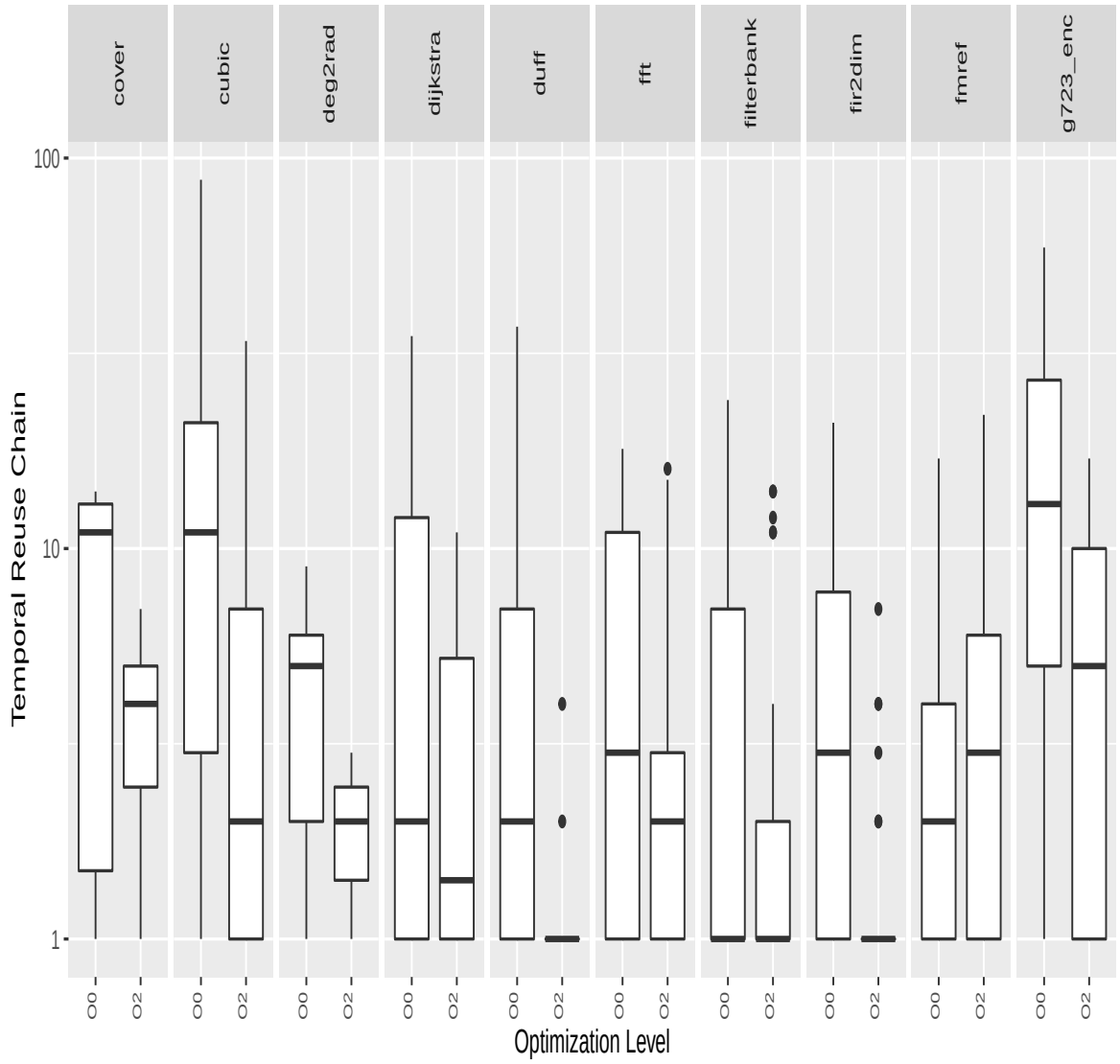


Figura E.14: Parte 2-Gráficas de reuso temporal comparativas entre GCC O0 y 02.

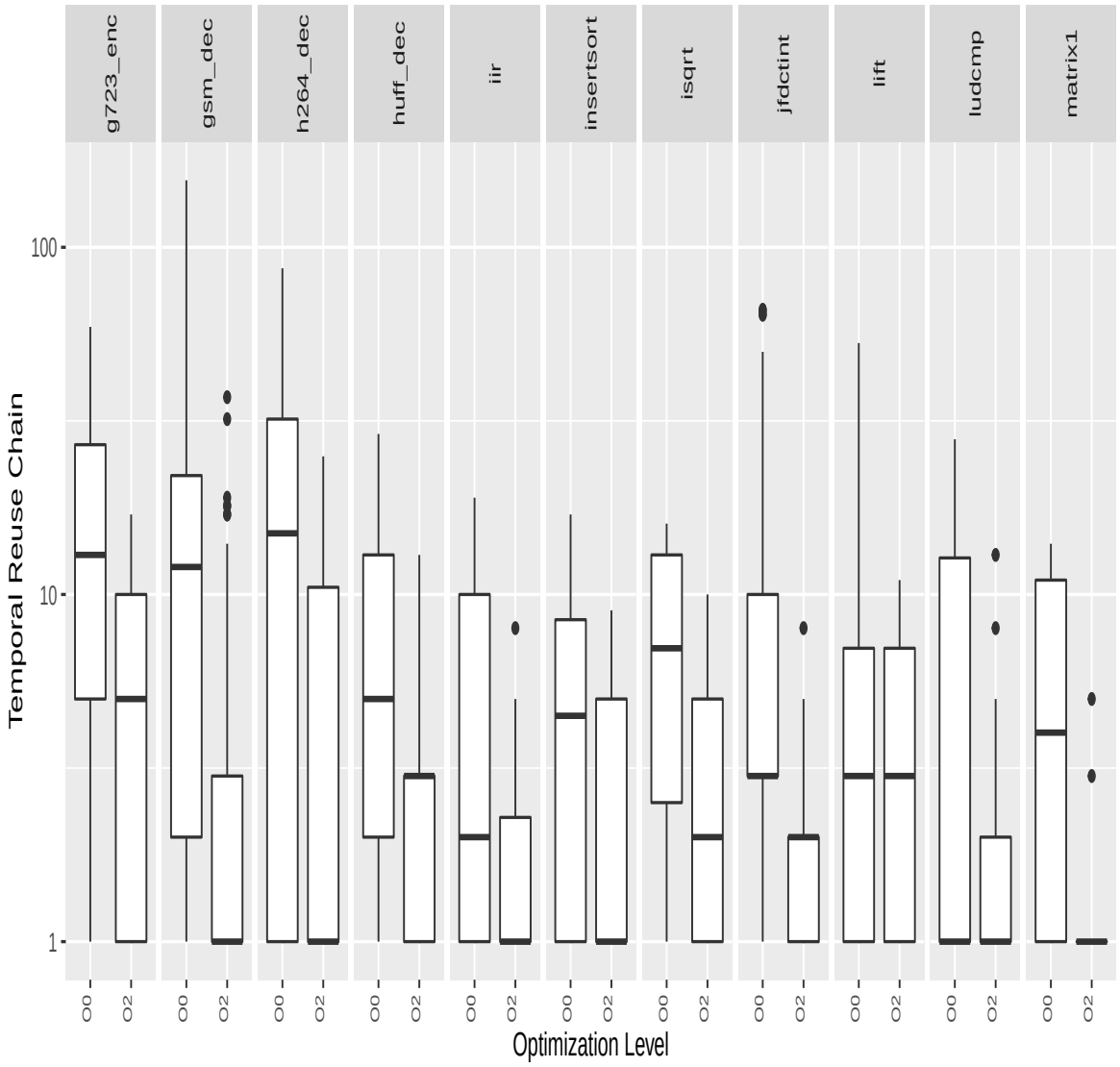


Figura E.15: Parte 3-Gráficas de reuso temporal comparativas entre GCC O0 y O2.

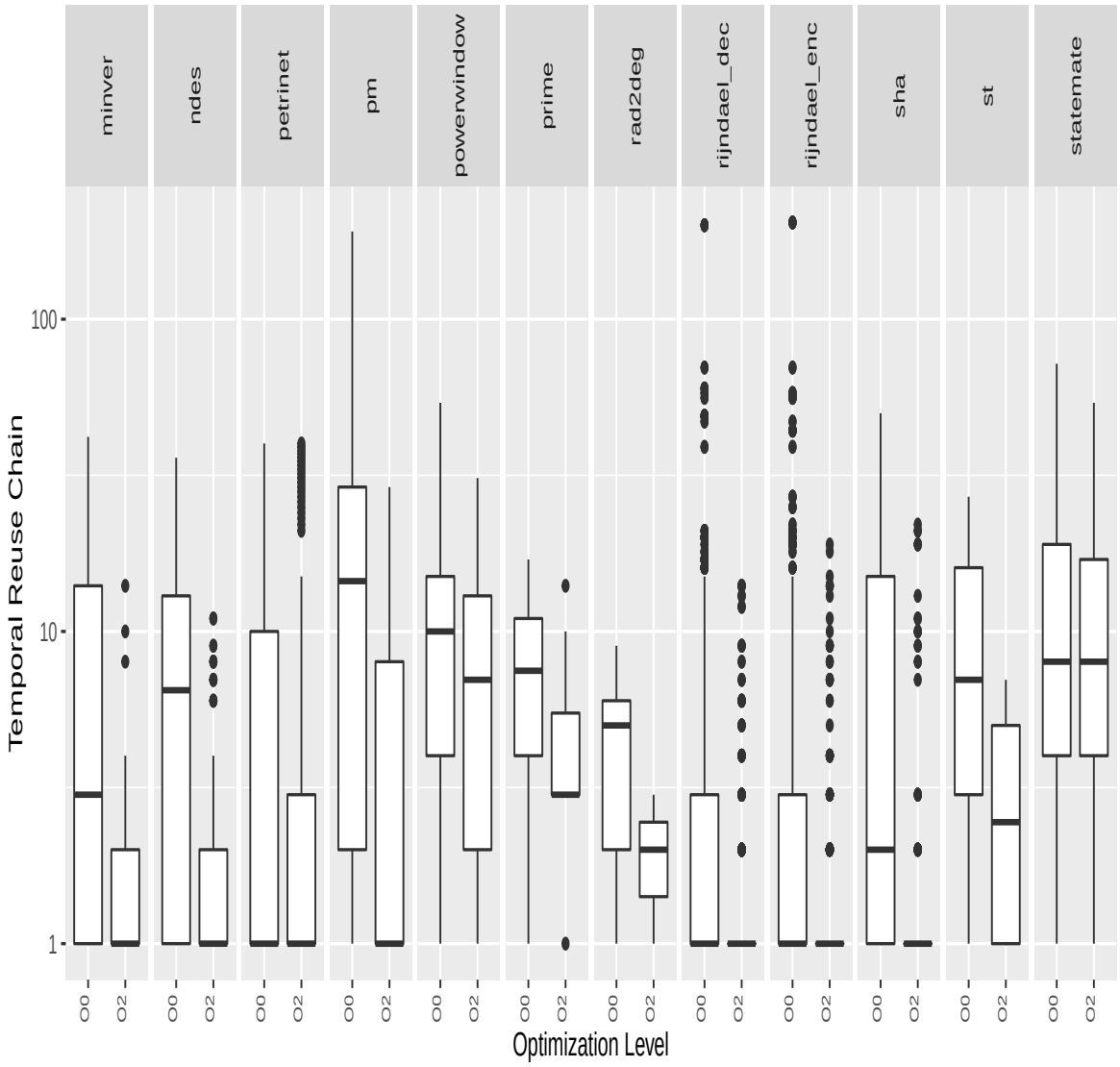


Figura E.16: Parte 4-Gráficas de reuso temporal comparativas entre GCC O0 y 02.

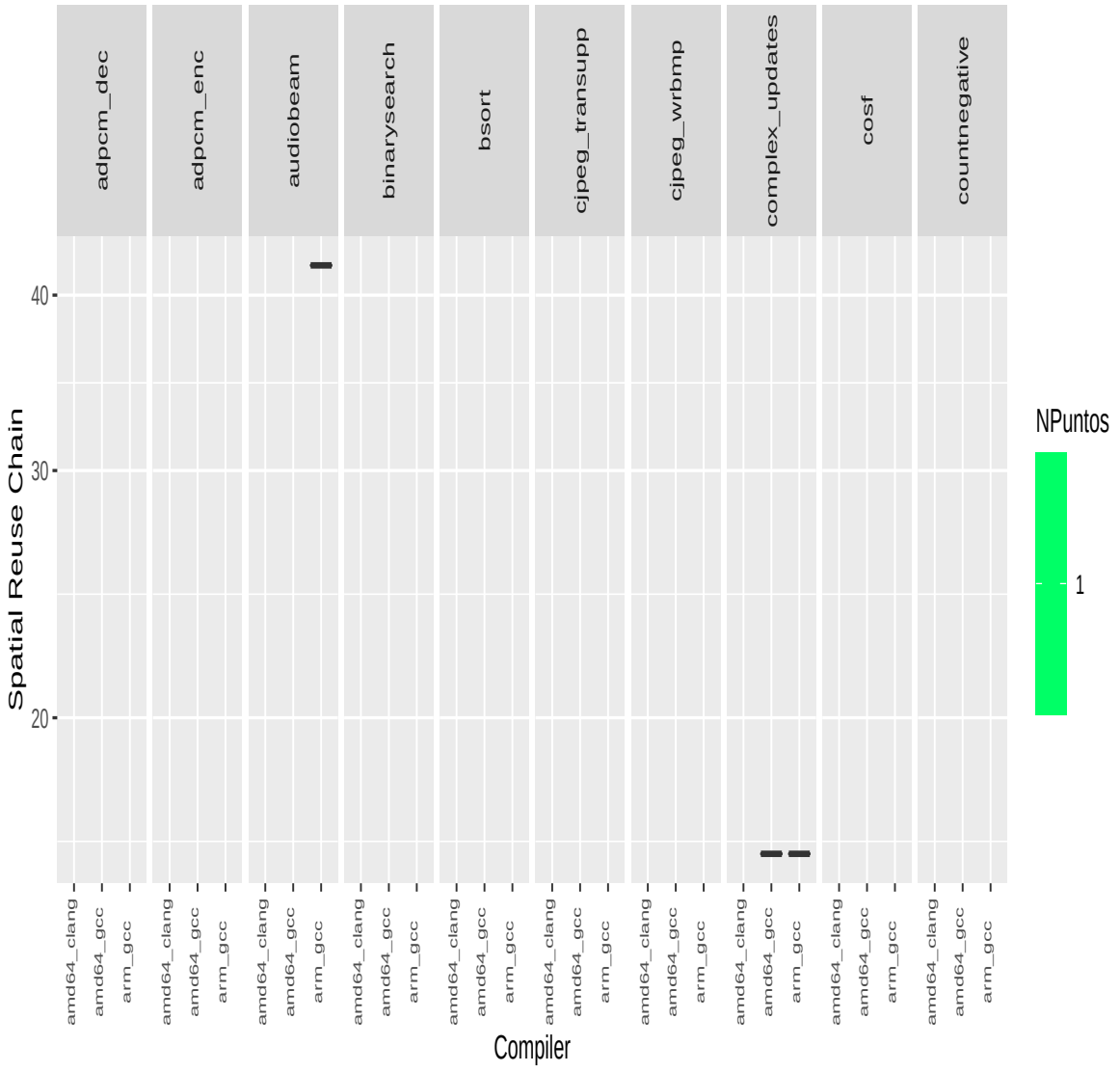


Figura E.17: Parte 1-Gráficas de reuso espacial en 00.

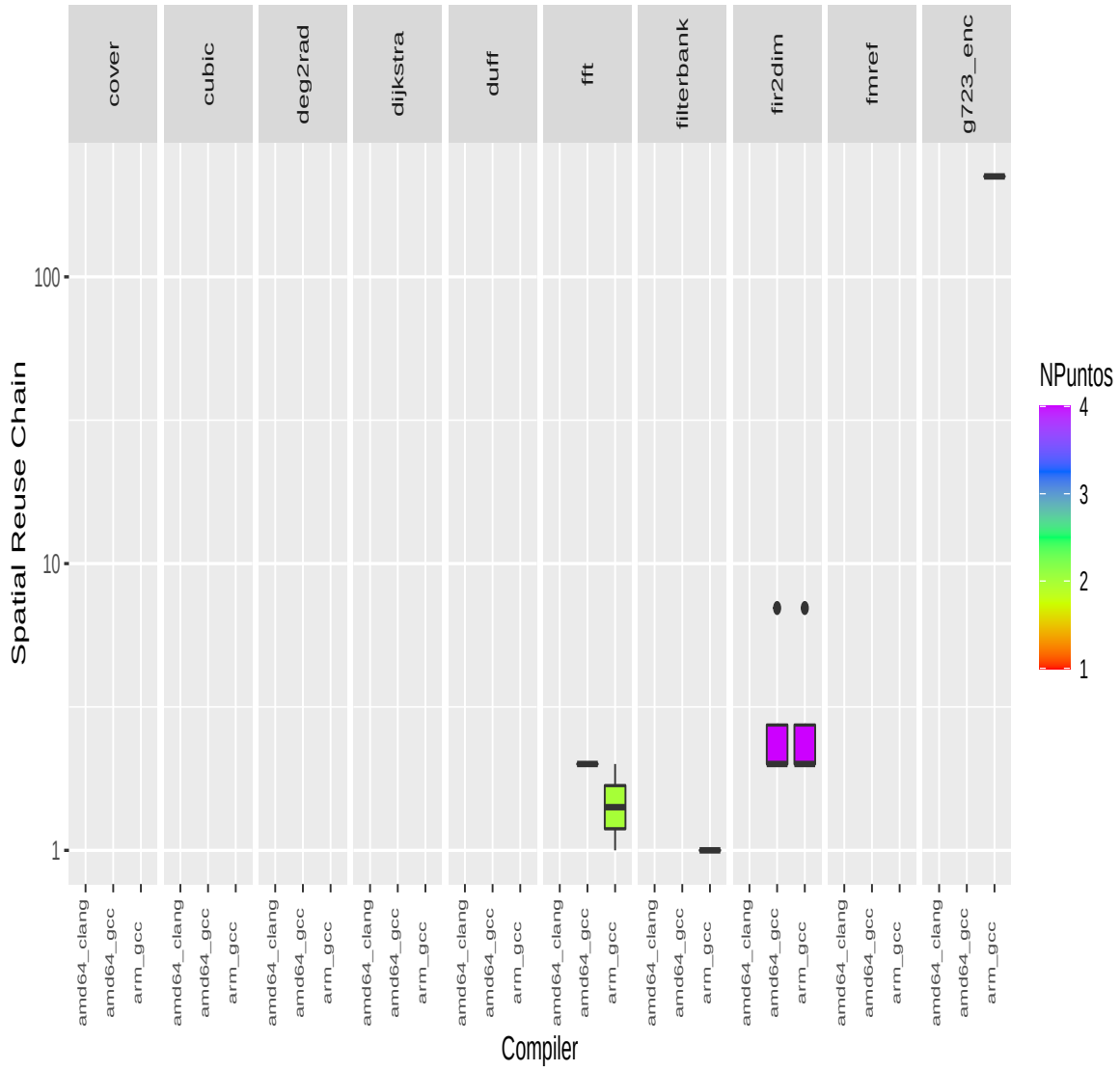


Figura E.18: Parte 2-Gráficas de reuso espacial en 00.

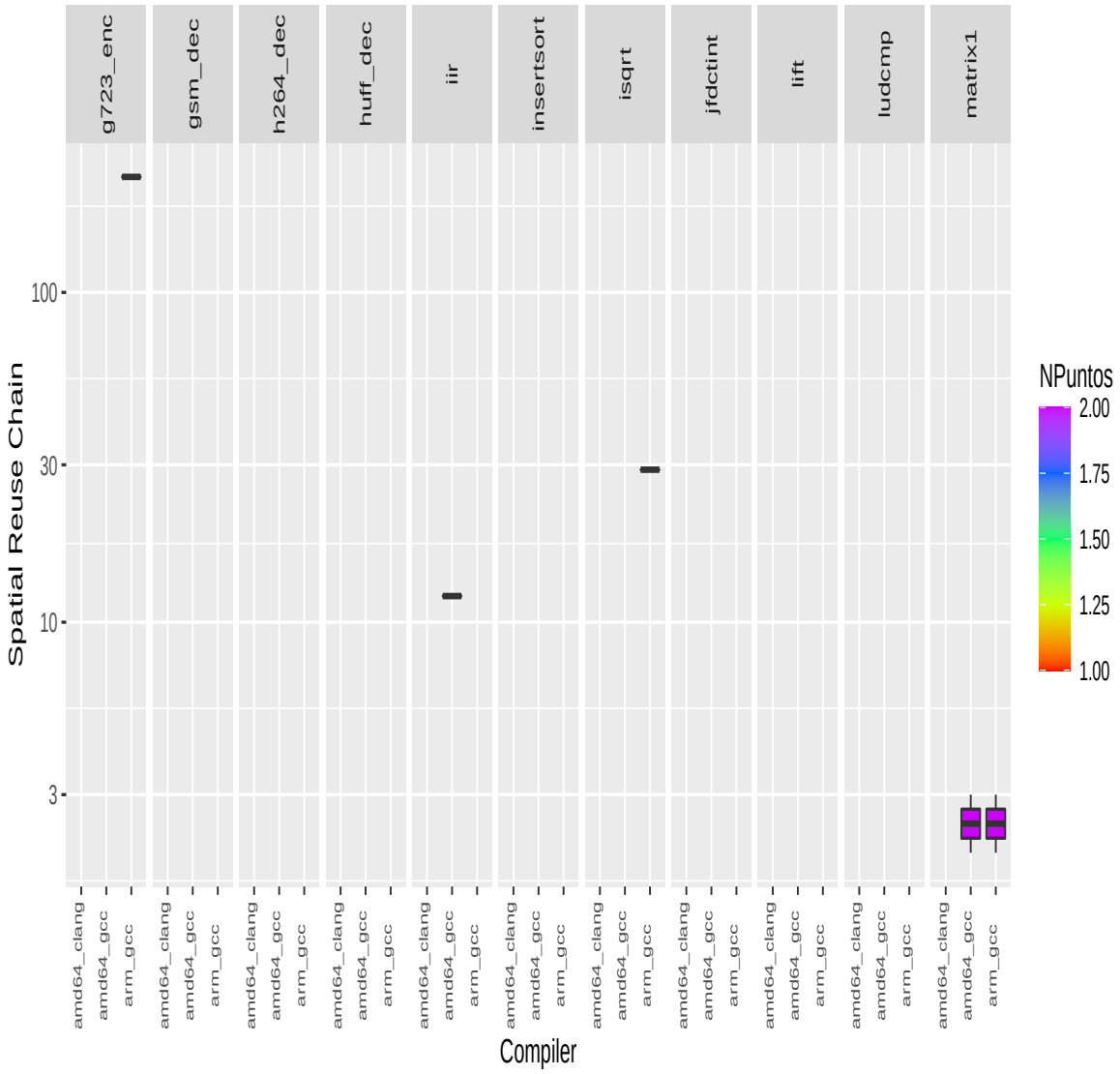


Figura E.19: Parte 3-Gráficas de reuso espacial en 00.

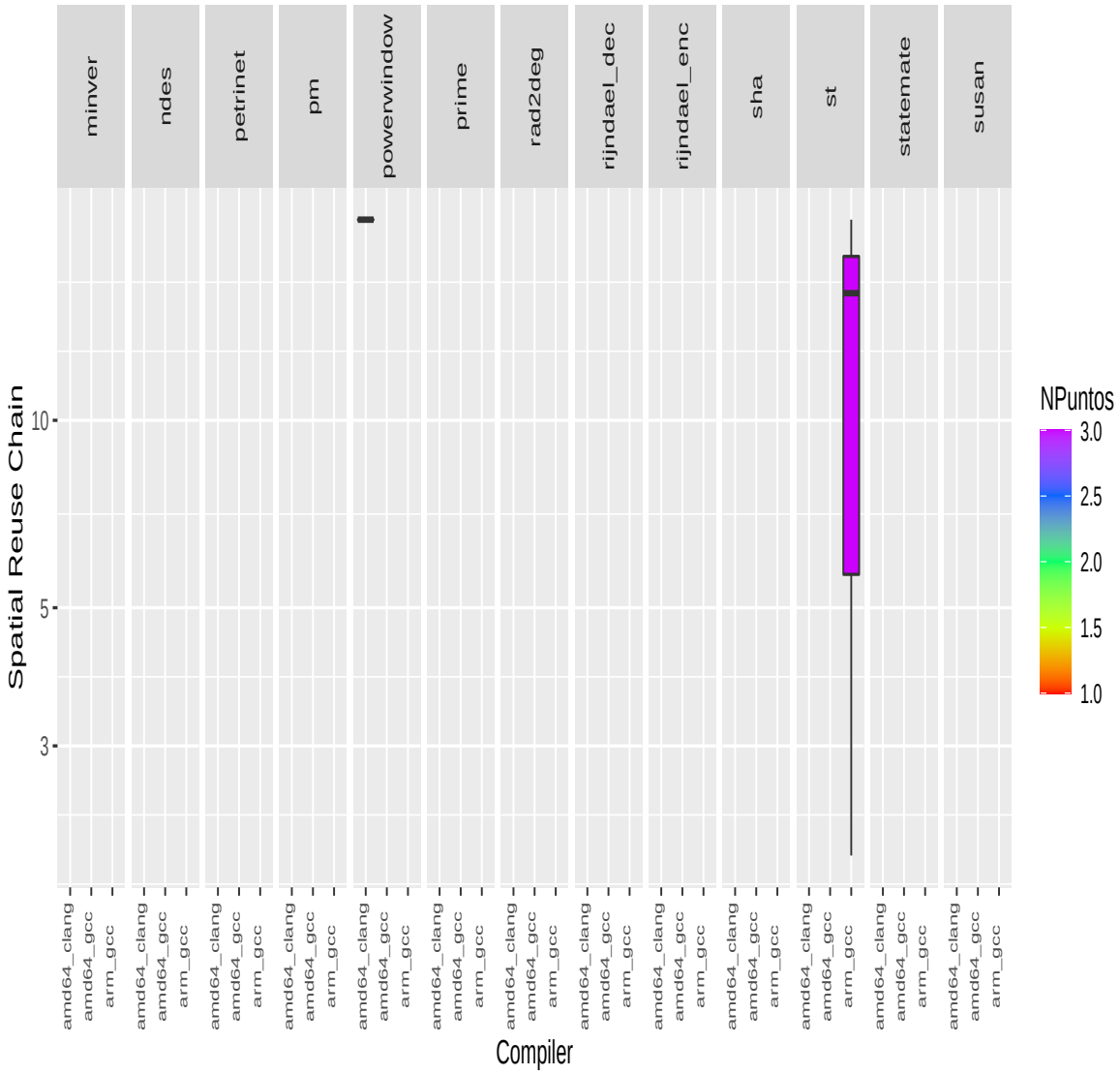


Figura E.20: Parte 4-Gráficas de reuso espacial en 00.

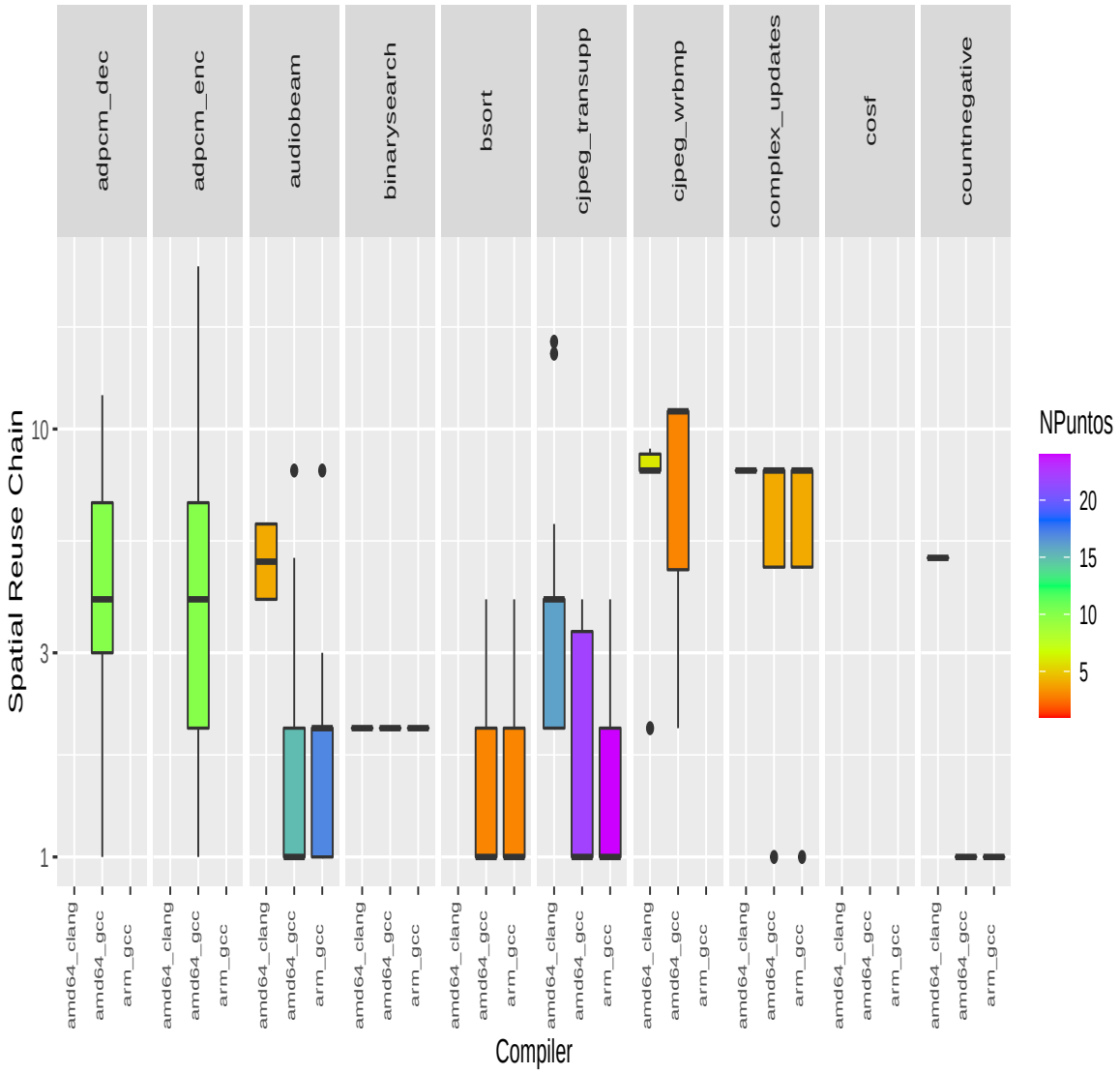


Figura E.21: Parte 1-Gráficas de reuso espacial en 02.

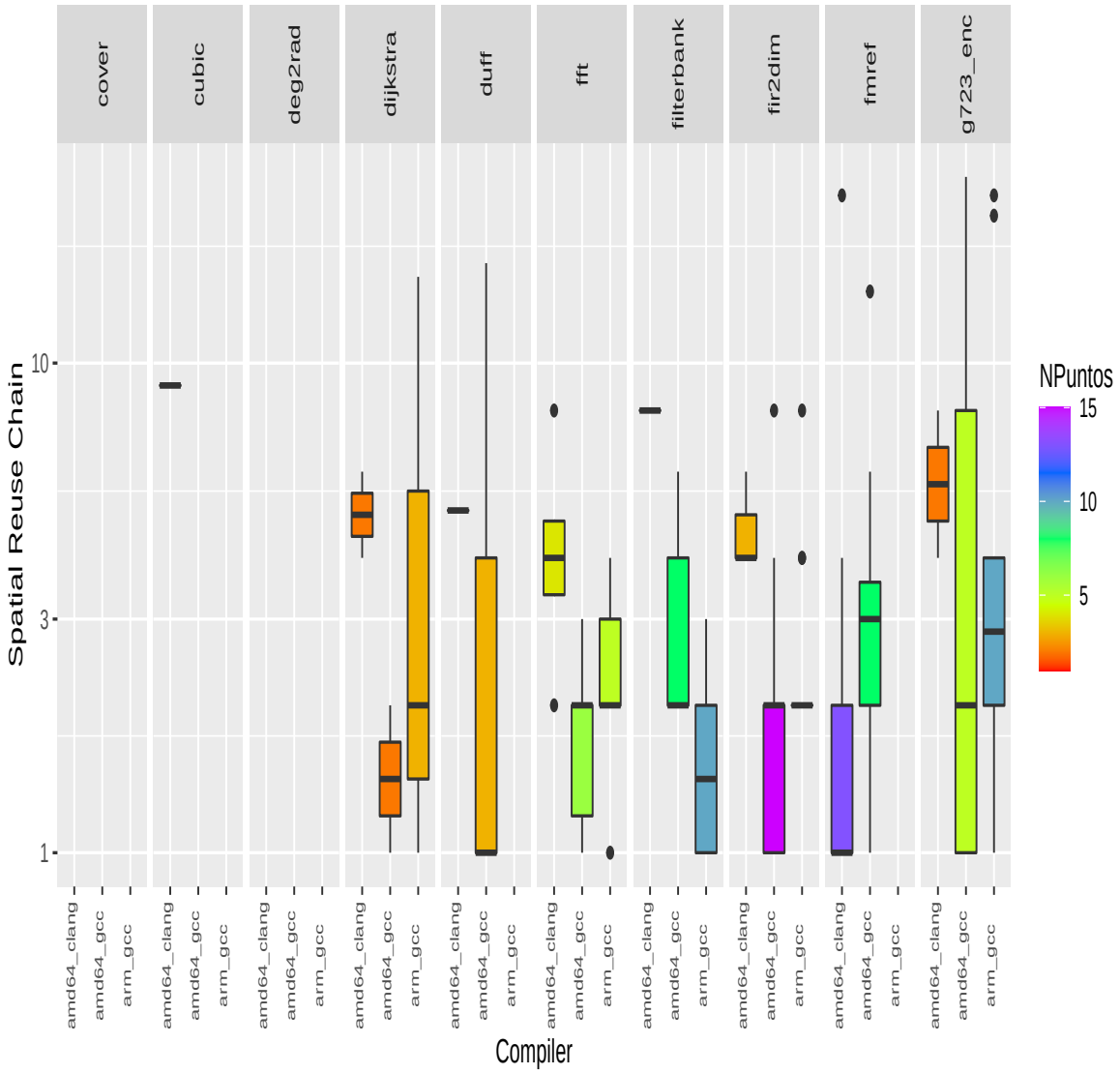


Figura E.22: Parte 2-Gráficas de reuso espacial en 02.

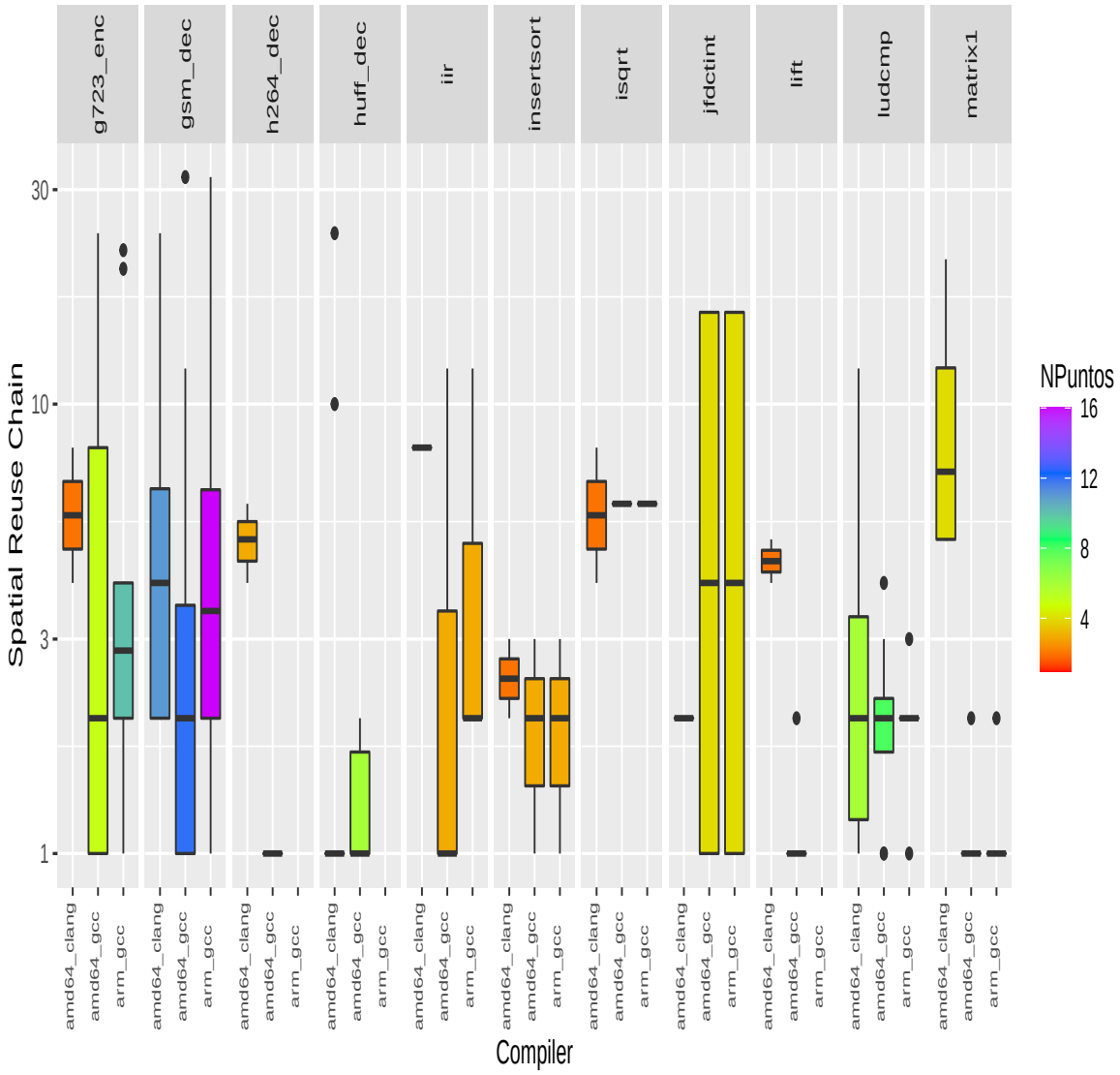


Figura E.23: Parte 3-Gráficas de reuso espacial en 02.

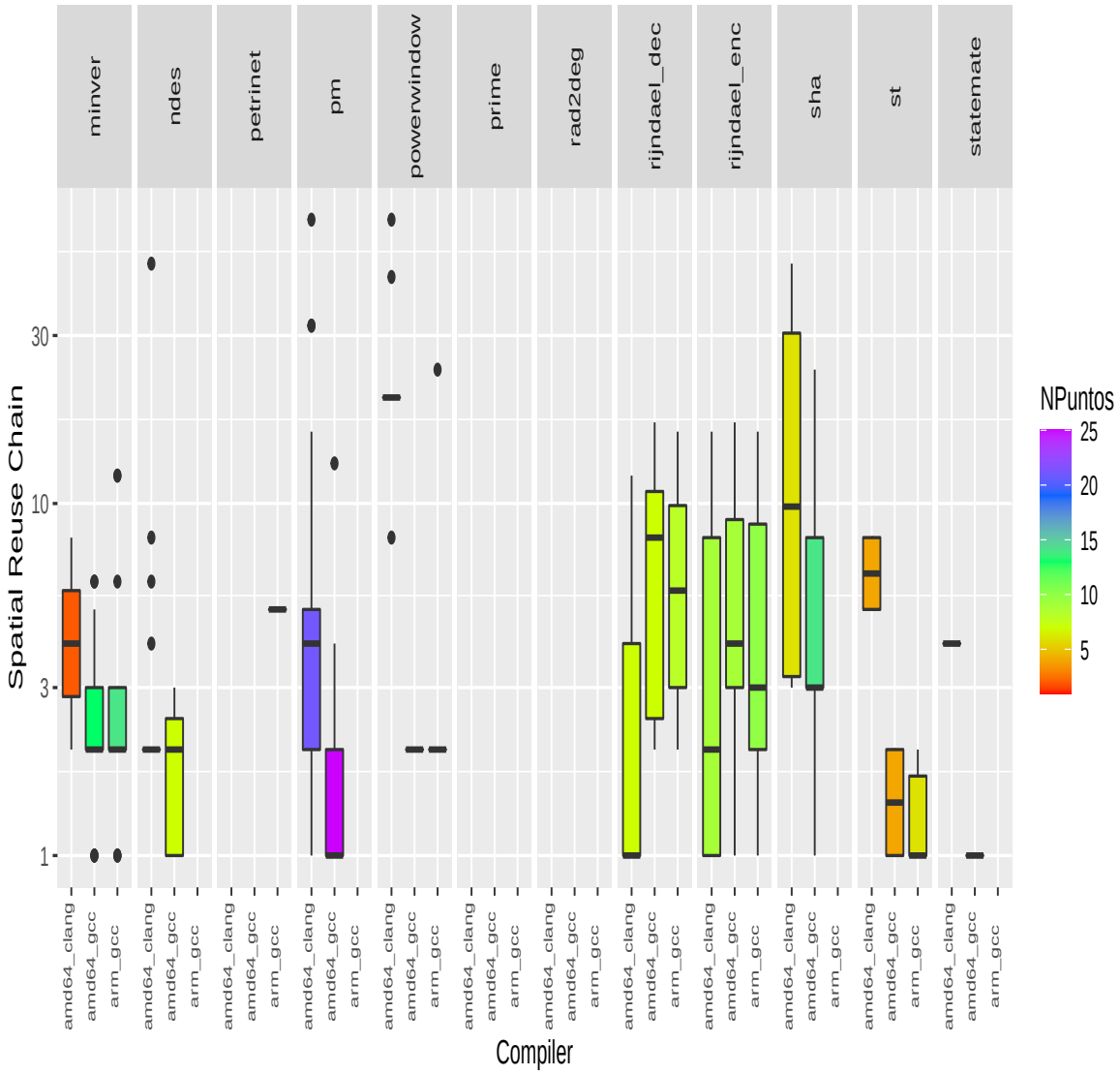


Figura E.24: Parte 4-Gráficas de reuso espacial en 02.

Anexos F

Dedicación al proyecto

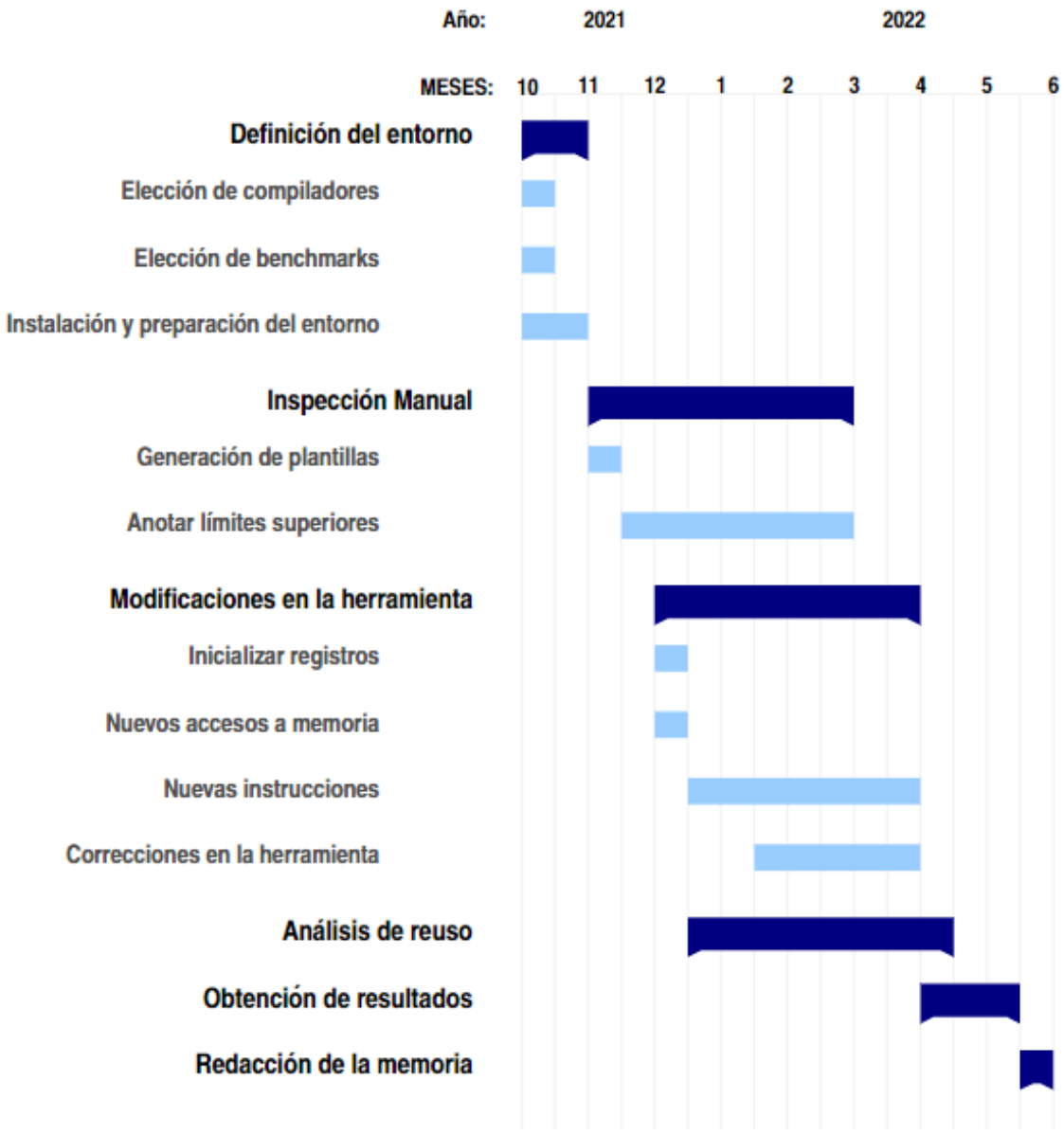


Figura F.1: Diagrama de Gantt

Anexos G

Contenido de la Inspección Manual

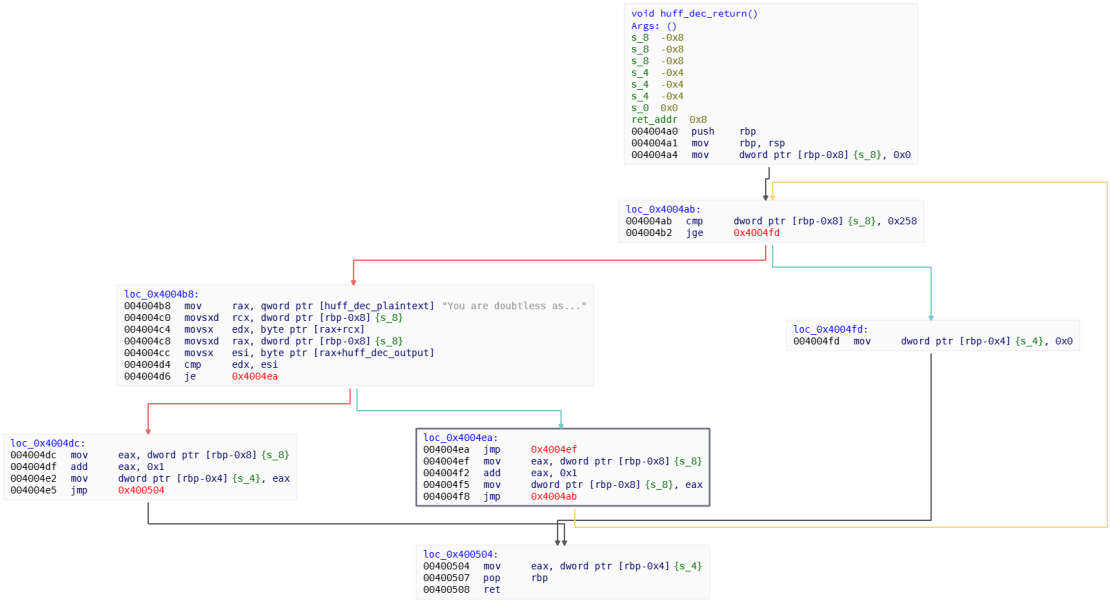


Figura G.1: CFG de huff_dec_return en clang O0.

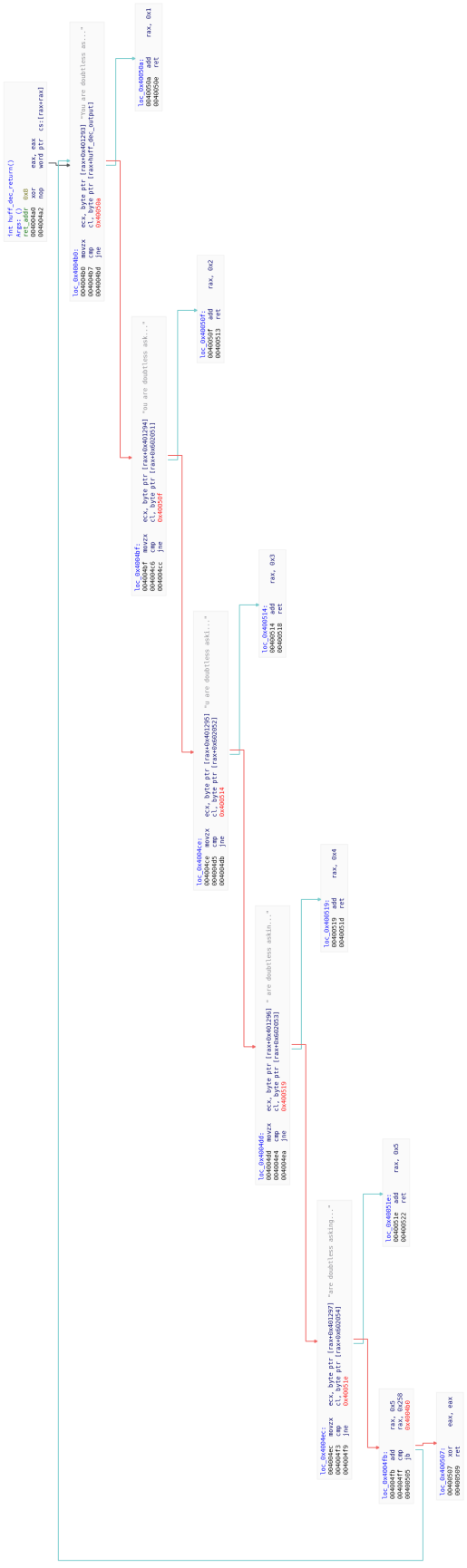


Figura G.2: CFG de huff_dec_return en clang O2.

Figura G.3: Notificación de error en h264_dec en el github de TACLeBench.

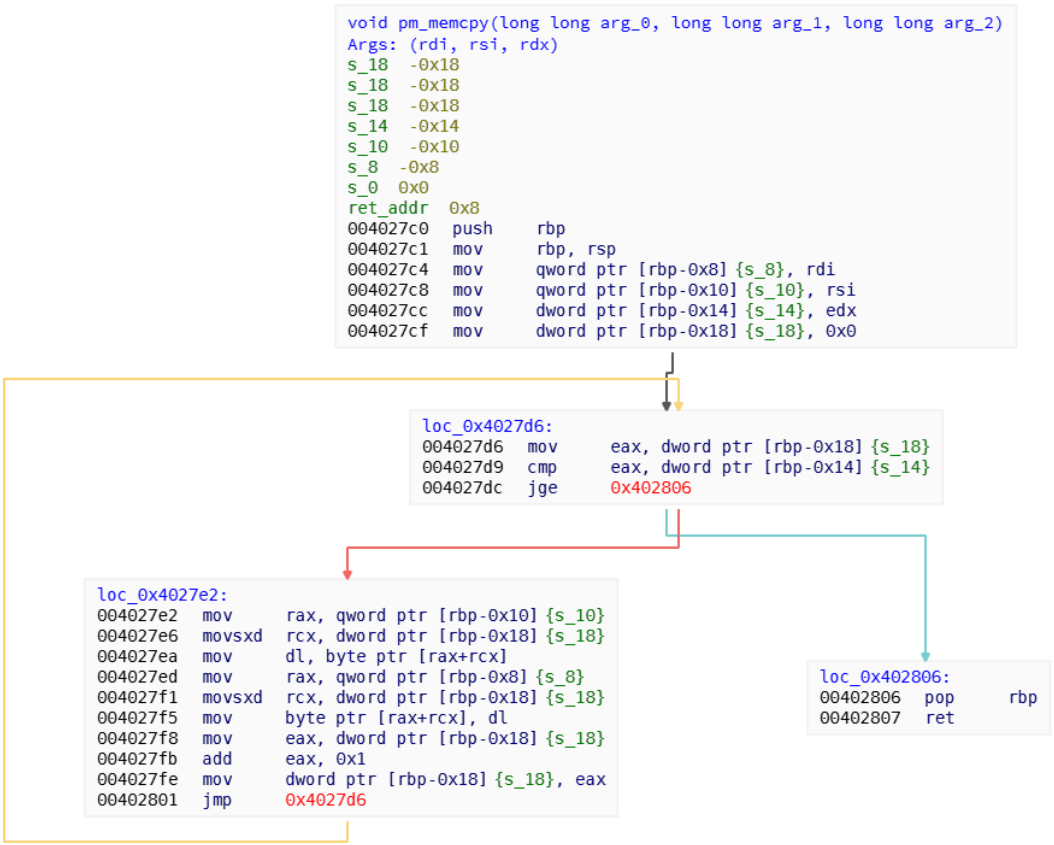


Figura G.4: CFG de pm_memcpy() en clang O0.

Figura G.5: Bloque básico del bucle 1 en pm_memcpy() en clang O2.

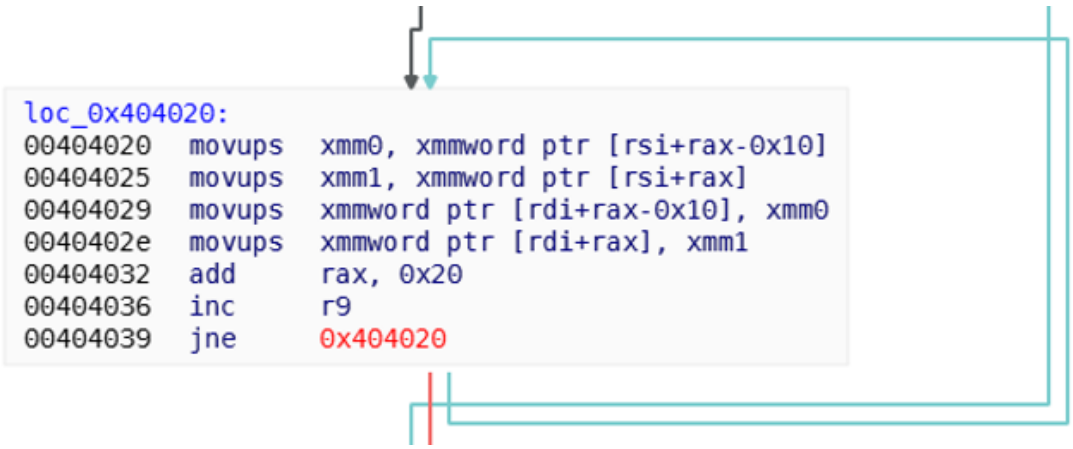


Figura G.6: Bloque básico del bucle 2 en pm_memcpy() en clang O2.

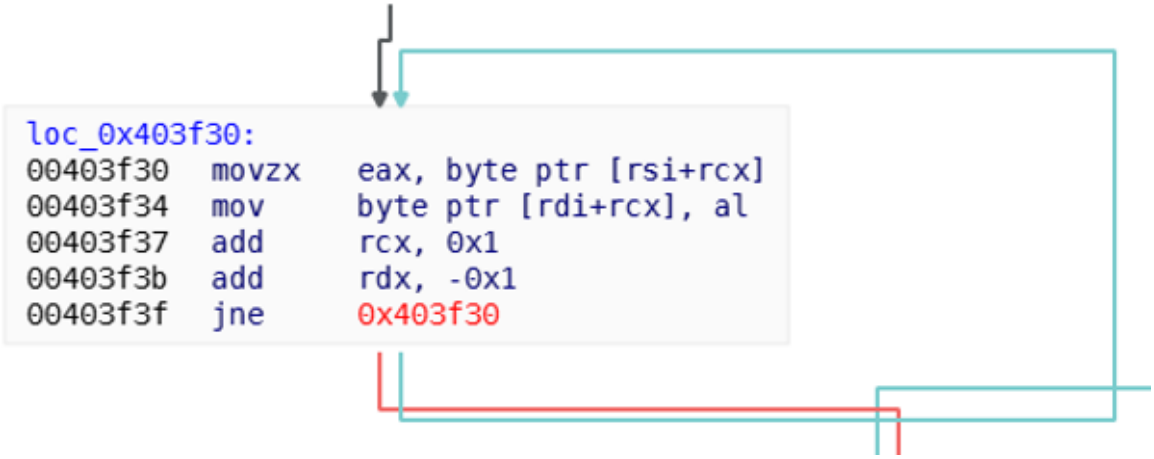


Figura G.7: Bloque básico del bucle 3 en pm_memcpy() en clang O2.

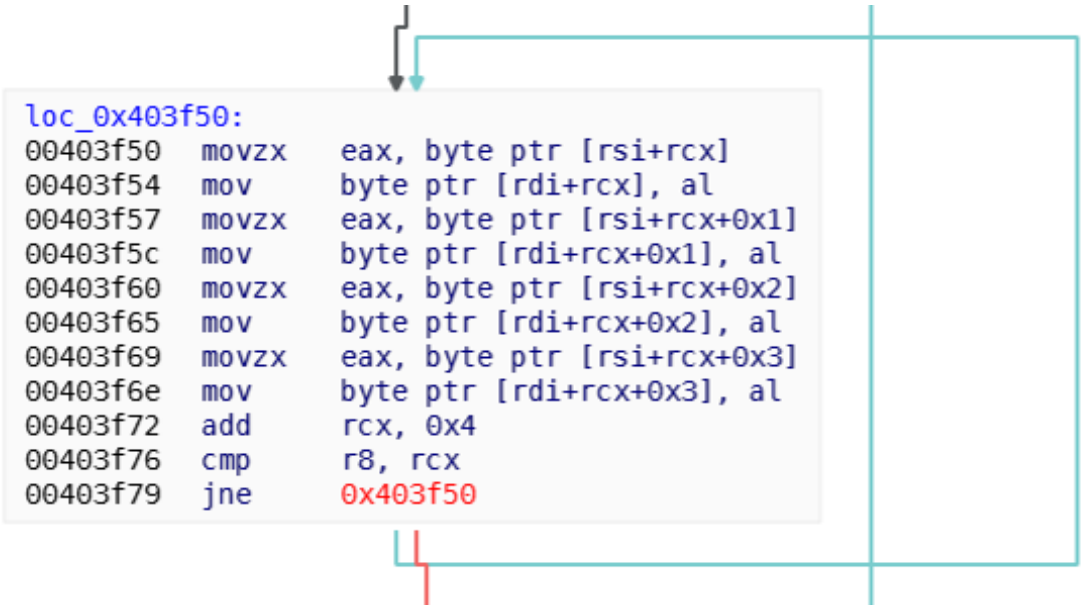


Figura G.8: Bloque básico del bucle 4 en pm_memcpy() en clang O2.