Optimización de programas software considerando la incertidumbre del tiempo de ejecución

José M. Aragón-Jurado[1], Juan Carlos de la Torre[1], Carlos Benito-Jareño[1], and Bernabé Dorronsoro[1]

Departamento de Ingeniería Informática, Escuela Superior de Ingeniería, Universidad de Cádiz, España
{josemiguel.aragon, juan.detorre, carlos.benito, bernabe.dorronsoro}@uca.es

Abstract. Proponemos en este trabajo un nuevo problema combinatorio para la optimización automática de programas, de forma que puedan sacar el máximo rendimiento de una arquitectura hardware considerada. Calcular el tiempo de ejecución de un programa es una tarea complicada, ya que está sujeta a una elevada incertidumbre. En este trabajo se estudian siete métodos distintos para estimar el tiempo de ejecución de los programas optimizados teniendo en cuenta la incertidumbre del sistema, y se comparan los resultados obtenidos con cada uno de ellos, reflejándose el efecto de la incertidumbre y obteniéndose evaluaciones altamente precisas con los métodos basados en el peor caso. Asimismo, se ha conseguido mejorar el tiempo de ejecución del programa original en un 33%.

Keywords: Algoritmo genético celular · LLVM · optimización de programas · incertidumbre.

1 Introducción

Todo programa software (SW) debe ser compilado para poderse ejecutar en una plataforma hardware (HW). Los compiladores son las herramientas encargadas de transformar el código de alto nivel en código máquina. Además, en este proceso también modifican el código con el fin de optimizar su rendimiento. Esto se hace aplicanco una secuencia genérica de optimizaciones que usualmente lleva a considerables reducciones en el tiempo de ejecución de los programas. Al ser genérica, se aplica sin tener en cuenta las características específicas tanto del SW como del HW en el que se va a ejecutar. Sin embargo, el gran avance de la tecnología a nivel de HW hace que en el mercado existan dispositivos con elementos físicos muy diferenciados. Por tanto, es lógico pensar que la aplicación de unas optimizaciones genéricas hace inviable el aprovechamiento máximo de las características específicas de cualquier arquitectura HW existente. Se hace entonces necesaria una correcta selección de las optimizaciones ofrecidas por

los compiladores en función del SW a optimizar y la plataforma HW donde se ejecutará para obtener el mejor rendimiento posible.

El proyecto LLVM [18] es una infraestructura de compilación ampliamente utilizada, compuesta por una colección de herramientas de compilación modulares y reutilizables. Entre los subproyectos que abarca LLVM destaca principalmente su núcleo principal, cuyo objetivo es ofrecer un optimizador de código fuente independiente del lenguaje de programación o de la arquitectura subyacente donde se vaya a ejecutar. Todas estas bibliotecas están basadas en la llamada representación intermedia de LLVM (IR o *Intermediate Representation*), y permiten adaptar, de forma sencilla, un compilador de cualquier lenguaje para emplear el optimizador de LLVM. Asimismo, existen multitud de proyectos de código libre para adaptar diferentes lenguajes de programación a LLVM IR, permitiendo la aplicación sencilla de optimizaciones independientes.

Las optimizaciones de código que realizan los compiladores basados en LLVM se aplican sobre LLVM IR. Estas optimizaciones reciben en LLVM el nombre de *passes*. Cada uno de estos *passes* realiza una transformación y/o un análisis concreto en el programa a optimizar. Entre estos, son especialmente interesantes para la optimización de código los *passes* de transformación, los cuales mutan el programa cambiando el código fuente y garantizando la invariabilidad de su semántica. Así, optimizar un programa SW con la infraestructura de compilación LLVM consiste en aplicar una secuencia adecuada de *passes*.

Tal y como se muestra en [27], el efecto directo de la aplicación de un *pass* concreto en el rendimiento de un programa es realmente bajo; sin embargo, su impacto en combinación con otros *passes* puede superar el orden de magnitud. Este efecto combinado está supeditado al orden de aplicación de los distintos *passes*. Además, la secuencia de *passes* que optimiza el programa es de tamaño variable en función de éste y del HW subyacente, pudiendo contener ésta múltiples repeticiones de un mismo *pass*. A priori, no se puede saber ni la longitud, ni los *passes*, ni la relación de orden correcta que definen la secuencia que permita conseguir la versión compilada del programa que pueda ofrecer el rendimiento óptimo.

La toma del tiempo de ejecución de un SW no es una tarea trivial. Existen factores, como por ejemplo la ejecución simultánea de procesos del sistema operativo, que provocan indeterminismo en las mediciones de un programa SW cuando es ejecutado varias veces en un mismo HW.

En este trabajo proponemos el uso de un algoritmo genético celular para la optimización de programas SW. En concreto, el algoritmo busca secuencias de *passes* de LLVM que consigan hacer el mejor uso posible de la arquitectura subyacente, minimizando el tiempo de ejecución del programa. La incertidumbre antes mencionada está presente en la evaluación de cada individuo, ya que requiere realizar mediciones de los tiempos de ejecución. Es por esto que el cálculo de la calidad de la solución que representa un individuo es una tarea compleja. Teniendo en cuenta que este valor de calidad determinará la evolución del algoritmo, es de vital importancia el uso de una métrica adecuada que permita estimar la calidad de una solución, con el menor esfuerzo computacional posible.

Consecuentemente, el objetivo principal del trabajo consiste en establecer una metodología para el tratamiento de la incertidumbre durante la optimización de programas SW en función de los resultados reflejados por las diferentes técnicas comparadas.

Las contribuciones de este trabajo son varias. En primer lugar, proponemos una definición matemática para modelar el problema de optimización de programas SW como un problema de optimización combinatoria. En segundo lugar, se presenta un algoritmo genético celular para resolver dicho problema. En tercer lugar, estudiamos siete enfoques diferentes para el tratamiento de la incertidumbre presente en la medición de tiempos durante la evaluación de la calidad de las soluciones, tres de ellos propuestos por nosotros. Por último, se realizará un estudio y una comparativa de los enfoques propuestos, observando el impacto de la incertidumbre en el comportamiento del algoritmo genético, y evaluando la calidad de las soluciones reportadas.

El documento se organiza de la siguiente manera: en la Sección 2, se comenta el estado del arte actual de la optimización del SW mientras que, en la Sección 3, se habla del tratamiento de la incertidumbre en problemas de optimización. En la Sección 4, se define el problema de optimización a resolver. Posteriormente, en la Sección 5 se especifica la metodología a seguir. Consecuentemente, en la Sección 6 se citan los detalles y las configuraciones del experimento a ejecutar. Finalmente, la Sección 7 expone los resultados obtenidos, y en la Sección 8 se comentan las conclusiones del estudio y las principales líneas de trabajo futuro identificadas tras este trabajo.

2 Estado del arte

Existen en la literatura dos líneas claramente diferenciadas sobre la optimización de programas SW. Por un lado, tenemos aquellas técnicas que no mantienen las propiedades funcionales del SW, es decir, que modifican su semántica, y por otro lado, aquellas que mejoran propiedades no funcionales del programa, pero asegurando que la semántica no cambia.

2.1 Optimización del SW sin garantía de preservar sus propiedades funcionales

La mejora automática de SW se refiere tanto a las características como las capacidades del SW, y se consigue mediante la aplicación automática y autónoma de transformaciones en el código fuente. En este trabajo nos centraremos en la eficiencia, si bien existen trabajos que se centran en características tales como las especificaciones del programa [20] o la identificación y reparación de errores para evitar el comportamiento erróneo del programa [15]. En la mejora de la eficiencia, los trabajos existentes se centran fundamentalmente en la paralelización de programas, bien de forma automática [22] o bien sacando provecho de características propias del código [16].

El principal problema de estas técnicas es que no pueden garantizar la semántica del programa tras aplicar las optimizaciones, requiriendo de un conjunto de pruebas para constatar la corrección del nuevo programa obtenido, aunque no sea posible garantizarla. La programación genética se ha utilizado para evolucionar de forma automática el código fuente, mejorando de esta forma las propiedades no funcionales del SW para mejorar su rendimiento consumo energético [6] o el tiempo de ejecución [15]. En [8], se estudia la verificación de casos de pruebas mediante el uso de algoritmos genéticos para la creación de programas mutantes. Estas técnicas están limitadas por el tamaño del código fuente. En [3], consiguen extender la funcionalidad de un programa insertándole automáticamente un fragmento de código que realiza dicha funcionalidad.

2.2 Optimización del SW preservando sus propiedades funcionales

Existen tradicionalmente una serie de trabajos enfocados en la optimización de programas durante la compilación, buscando la mejora en tiempo de ejecución [16]. Estas técnicas se centran en la búsqueda de secuencias de transformaciones de código que mejoran el tiempo de ejecución de los programas, en general. Estas transformaciones de código fuente son modificaciones predefinidas que mantienen inalterada la semántica del programa.

En esta misma línea, existen trabajos más recientes que tratan de predecir la adecuada secuencia de transformaciones mediante el uso de diferentes técnicas. De esta forma, encontramos en [9] la propuesta de un método de dos niveles para elegir la selección más conveniente de optimizaciones para una aplicación autoajustable. En [2], se acelera la elección de las mejores optimizaciones de LLVM buscando subsecuencias con técnicas de *machine learning*.

Algunos trabajos han seguido estrategias basadas en heurísticas y algoritmos evolutivos para seleccionar los *flags* del compilador para la optimización de SW, basándose en distintas métricas como el tiempo de ejecución [25] o el consumo energético [29]. En [11], se utilizan técnicas estadísticas junto a algoritmos evolutivos para ayudar a seleccionar las soluciones.

Una novedosa línea de investigación es el uso de técnicas estadísticas para calcular el impacto que una optimización va a provocar sobre el código. En [23], se utiliza un procedimiento basado en vectores ortogonales para proponer un algoritmo interactivo que activa o desactiva un subconjunto de *passes*. En [28] se hace uso de técnicas de análisis de sensibilidad para determinar una lista de *flags* ofrecida por fabricantes de FPGAs para optimizar diseños. En [27] se realiza un completo análisis de sensibilidad utilizando eFast sobre el impacto de los passes ofrecidos por LLVM-3.8. En este trabajo queda patente que la influencia directa de las transformaciones sobre el tiempo de ejecución del benchmark BEEBS [21] es muy baja, siendo por el contrario muy alta la influencia combinada de todos ellos, lo que demuestra la complejidad del problema al que nos enfrentamos.

3 Optimización e incertidumbre

Cuando hablamos de datos, la incertidumbre se puede definir como la propiedad cuantificable de los datos que provoca la dispersión de valores observable en los mismos. Dicho de otra forma, la incertidumbre en nuestras mediciones es el ruido que provoca que los datos se desvíen de sus valores originales. A la hora de tratar con grandes cantidades de datos, la incertidumbre aparece con relativa frecuencia. Como se especifica en [12] y en [14], dentro del ámbito de los problemas de optimización, la incertidumbre puede afectar a tres parámetros diferentes:

- **Las variables de decisión**. Las variables que empleamos para representar nuestro problema de optimización pueden contener perturbaciones en sus valores.
- **La función de fitness**. Al evaluar el valor de fitness de un individuo, dicho valor puede contener ruido o perturbaciones que lo alejan de su valor real. Estas perturbaciones pueden ser debidas a diferentes fuentes, que se pueden englobar en la siguientes:
  - **Ruido inherente a la función de fitness**. La incertidumbre proviene de errores de precisión en los instrumentos de evaluación utilizados o en las variaciones de las condiciones del entorno durante la ejecución de la función de fitness. Esta incertidumbre también recibe el nombre de imprecisión o incertidumbre epistémica.
  - **Función de fitness aproximada**. En ciertas ocasiones, debido al coste de calcular la función de fitness, se utiliza un metamodelo para aproximar el valor de esta función a partir de diferentes técnicas. El uso de esta aproximación provoca un error inherente a ella.
- **Las condiciones del entorno**. Las condiciones del entorno de nuestro problema pueden variar a lo largo del tiempo, cambiando las especificaciones del problema de optimización en cada instante de tiempo.

El problema tratado en este trabajo es un caso de imprecisión o incertidumbre epistémica, que tiene lugar durante la evaluación del individuo. El ruido o imprecisión en el cálculo de la función de fitness se trata de formas diferentes en la literatura. Los principales enfoques se dividen según como modelen el ruido, siendo los más comunes la suposición de una determinada distribución de probabilidad subyacente para el ruido y la representación de la imprecisión a través de intervalos.

Las funciones de fitness con ruido son comunes en los problemas de optimización. En [14] se exponen los métodos más habituales de tratar el ruido, asumiendo que éste sigue una distribución normal de media cero. La mayoría de ellos se basan en realizar un promedio del fitness implícita o explícitamente. Recurren a un método de muestreo y, bajo la suposición de que sigue una distribución normal, toman un estadístico. En [24] se realiza una comparativa de diferentes métodos de muestreo y en [10] se analiza el coste computacional que supone agregar dicho muestreo en el fitness a diferentes estrategias metaheurísticas.

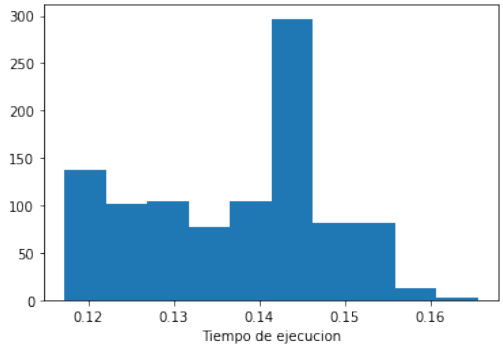

Fig. 1. Tiempos de ejecución del benchmark Polybench en 1000 ejecuciones independientes.

No obstante, existen casos en los que no podemos realizar esa suposición debido a que desconocemos la distribución real del ruido presente en la función de fitness. En estas ocasiones, la mejor forma de representar la incertidumbre existente en nuestros valores es a través de un rango o intervalo de valores. La utilización de dicho intervalo para representar el fitness del individuo requiere de algún método para establecer una relación de orden entre individuos, según su calidad. En la literatura existen diferentes técnicas para resolver este problema, como definir relaciones de orden parcial entre intervalos [7] [5] [17], la optimización del peor caso [26] [30], o convertir el problema en uno determinista facilitando su resolución [13].

4 Definición del problema

El problema a resolver trata de encontrar la mejor secuencia de *passes* LLVM que minimicen el tiempo de ejecución de un programa determinado ejecutado sobre una arquitectura específica. A cada uno de los *passes* ofrecidos por LLVM se les asocia un número entero, y una solución al problema consiste en una cadena de enteros, representando los *passes* que deben aplicarse y el orden en que deben hacerlo.

Para evaluar la calidad de una solución es necesario compilar el programa con la secuencia de *passes* dada y ejecutarlo, tomando medida del tiempo de ejecución. La alta incertidumbre en la medición del tiempo de ejecución dificulta la asignación del valor de fitness a las soluciones. Esta incertidumbre es fácilmente comprobable ejecutando varias instancias independientes del mismo programa y comparando los valores obtenidos. En este sentido, la Figura 1 muestra el histograma obtenido al medir del tiempo de 1000 ejecuciones independientes del mismo SW. En ella se aprecia que las mediciones de tiempo de ejecución no siguen una distribución normal, extremo confirmado con tests de normalidad.

La ausencia de normalidad en la distribución es claro indicio de que la metodología consistente en tomar la media de tres o cinco ejecuciones indepen-

dientes (común en la literatura) probablemente no sea un buen estadístico que represente el tiempo de ejecución de un programa determinado. Por lo tanto, el objetivo que nos planteamos es encontrar una representación adecuada del tiempo de ejecución que permita manejar la incertidumbre indicada.

5 Metodología

En esta sección presentamos el algoritmo genético utilizado, así como las distintas funciones de fitness propuestas.

5.1 Algoritmo genético celular

Un algoritmo genético celular (cGA) [1] es un tipo de GA descentralizado en el que se establece una relación de distancia entre los individuos de la población organizándolos en una retícula. De esta forma, los individuos solo interactúan con sus vecinos, aplicándose las operaciones de selección, cruce y mutación entre ellos. Si el individuo resultante es de mayor (o similar) calidad al de la celda concreta que se está actualizando, entonces lo reemplaza. En general, el cGA preserva la diversidad por más tiempo que un genético panmíctico, siendo de gran utilidad para nuestro problema debido a que, mediante estudios preliminares, determinamos que las soluciones de nuestro problema convergen rápidamente.

En un GA, los individuos de la población codifican soluciones al problema tratado. En nuestro problema, representamos nuestra secuencia de transformaciones como un array de enteros. Cada entero representa un *pass* de LLVM concreto, y el orden que ocupen dentro del array representa el orden en el que las transformaciones serán aplicadas sobre el código fuente. Por último, la función de fitness consiste en medir el tiempo de ejecución del programa optimizado empleando ese individuo.

5.2 Funciones de fitness

Estudiamos en este trabajo siete enfoques diferentes para calcular el valor de fitness de una solución al problema considerado. Nótese que cuatro de ellas suponen diferentes técnicas para tratar la incertidumbre (mediana, intervalo, pcaso, y pcaso15), mientras que otras dos tienen por objetivo la reducción de la incertidumbre en las mediciones de tiempo (delay, numa). Asimismo, tres técnicas han sido propuestas por nosotros (pcaso15, delay, numa) a diferencia de las técnicas restantes que fueron extraídas de la literatura (mediana, intervalo, pcaso).

- **Una ejecución (1toma)**. Consiste en ejecutar el programa una única vez y tomar su tiempo de ejecución.
- **Mediana de 5 ejecuciones (mediana)**. Consiste en realizar cinco ejecuciones independientes del programa y obtener la mediana de los tiempos de ejecución. Al no tratarse de una distribución normal, la mediana es un estadístico más apropiado que la media.

- **Intervalo de 5 ejecuciones (intervalo)**. Consiste en tomar mediciones de tiempo de cinco ejecuciones independientes. Posteriormente, esta muestra de datos se normaliza empleando el método de remuestreo de bootstrap, que amplía la muestra mediante el reemplazo aleatorio. Para ello, se realizan 500 réplicas de bootstrap con el fin de reducir el error de variabilidad que proviene de su aleatoriedad. Posteriormente, a partir de la muestra ampliada se construye el intervalo de confianza con un nivel de significancia del 5% mediante el método del percentil. Para comparar los intervalos dentro de nuestro algoritmo genético se emplea la relación de orden parcial definida en la Ecuación 1 para un problema de minimización y dos intervalos $A = [a_l, a_r] = \langle a_c, a_w \rangle$ y $B = [b_L, b_R] = \langle b_c, b_w \rangle$, siendo x_c el centro del intervalo y x_w la anchura del intervalo.

$$A \leq^{min} B \Longleftrightarrow \begin{cases} a_c < b_c & \text{si } a_c \neq b_c \\ a_w \leq b_w & \text{si } a_c = b_c \end{cases} \tag{1}$$

- **Peor caso de 5 ejecuciones (pcaso)**. Toma como valor de fitness el peor de un total de 5 ejecuciones.
- **Intervalo sobre las 5 peores ejecuciones de 15 (pcaso15)**. Este enfoque combina la optimización del peor caso con los intervalos. Se realizan un total de 15 ejecuciones y se utilizan las 5 peores en la construcción de un intervalo de confianza. La construcción y el tratamiento del intervalo es idéntico al especificado previamente.
- **Intervalo de 5 ejecuciones (con retardo de un segundo entre mediciones) (delay)**. Intervalo de confianza construido a partir de un total de 5 ejecuciones, pero introduciendo entre cada medición un retardo de un segundo.
- **Intervalo de 5 ejecuciones (NUMAs diferentes) (numa)**. Intervalo de confianza construido a partir de un total de 5 ejecuciones, pero la ejecución del algoritmo genético y la ejecución del programa a evaluar se realizan en NUMAs diferentes.

6 Detalles de Experimentación

Los parámetros y operadores fijados para el cGA son:

- **Operador de cruce:** cruce por dos puntos.
- **Criterio de terminación:** $10,000$ evaluaciones.
- **Operador de selección:** torneo binario.
- **Operador de mutación**: cambiar el valor del gen a mutar por un entero aleatorio en el rango entre 0 y el número de *passes* disponibles menos uno.
- **Tamaño de la población**: 100 individuos.
- **Tamaño del cromosoma**: 300, similar a la longitud de las secuencias utilizadas en los flags de optimización en Clang.
- **Probabilidad de mutación** de cada gen: 1/300.
- **Probabilidad de cruce**: 1.0.

– **Relación de vecindad**: Von Neumann.

Para la implementación y uso de los algoritmos genéticos se ha empleado la biblioteca JMetalPy [4]. Es una biblioteca centrada en estrategias metaheurísticas que contiene una numerosa cantidad de algoritmos genéticos y operadores. La versión de LLVM que se ha utilizado es la 9.0.1.

Como plataforma de ejecución hemos seleccionado un computador masivamente paralelo que se compone de un total de 96 núcleos ARM, distribuidos en 4 nodos NUMA. Se trata de un computador con una alta capacidad paralela y una gran potencia de cómputo, donde se espera una menor variabilidad en las mediciones de tiempo que en computadores con un bajo número de núcleos. Este computador tiene instalado el sistema operativo Ubuntu 20.04.1 LTS.

La selección del programa a optimizar puede influir notablemente en los resultados. Es necesario seleccionar un benchmark lo suficientemente potente para que se vean reflejadas las transformaciones en el tiempo de ejecución del programa, de forma que podamos evidenciar la evolución del algoritmo genético en los diferentes enfoques. Por otro lado, dicho benchmark debe ser rápido de ejecutar, puesto que el GA requerirá de la evaluación de decenas de miles versiones del programa.

Polybench [19] fue el conjunto de benchmarks seleccionado por la razón expuesta previamente. Polybench es una colección de diferentes benchmarks que tratan de evaluar diferentes propiedades de la arquitectura y monitorizar el procesador. Empleamos el conjunto de datos pequeño para no añadir demasiada carga computacional al programa y realizamos 10 iteraciones del benchmark.

7 Resultados

Se han realizado cinco ejecuciones independientes del cGA para cada uno de los enfoques para evaluar el fitness considerados, tomándose la mejor solución global encontrada para cada enfoque. Para evaluar la calidad de dichas soluciones, se han realizado un total de $1,000$ ejecuciones independientes del programa luego de ser optimizado por la secuencia de *passes* obtenida como solución. Con los tiempos medidos hemos trazado una gráfica de caja y bigotes, donde comparamos las $1,000$ mediciones realizadas a posteriori con la estimación tomada por el cGA al evaluar la solución. Dicha estimación se representa como uno o varios puntos, en función del enfoque empleado durante la evaluación. Una estimación será considerada como precisa si se encuentra dentro de la caja, es decir, entre el primer y tercer cuartil. Este estudio se ha hecho con cada uno de los enfoques reportados como se puede ver en la Figura 2.

La mayoría de los enfoques propuestos afrontan un problema de sobreestimación del valor de fitness del individuo reportado. Aún así, se puede percibir claramente que el que realiza una sola ejecución del programa y el que emplea la mediana de 5 ejecuciones son los que sobreestiman en mayor medida el valor de la solución reportada. De igual forma, los enfoques basados en el peor caso, tanto el básico como el que se emplea en combinación con intervalos, son los que

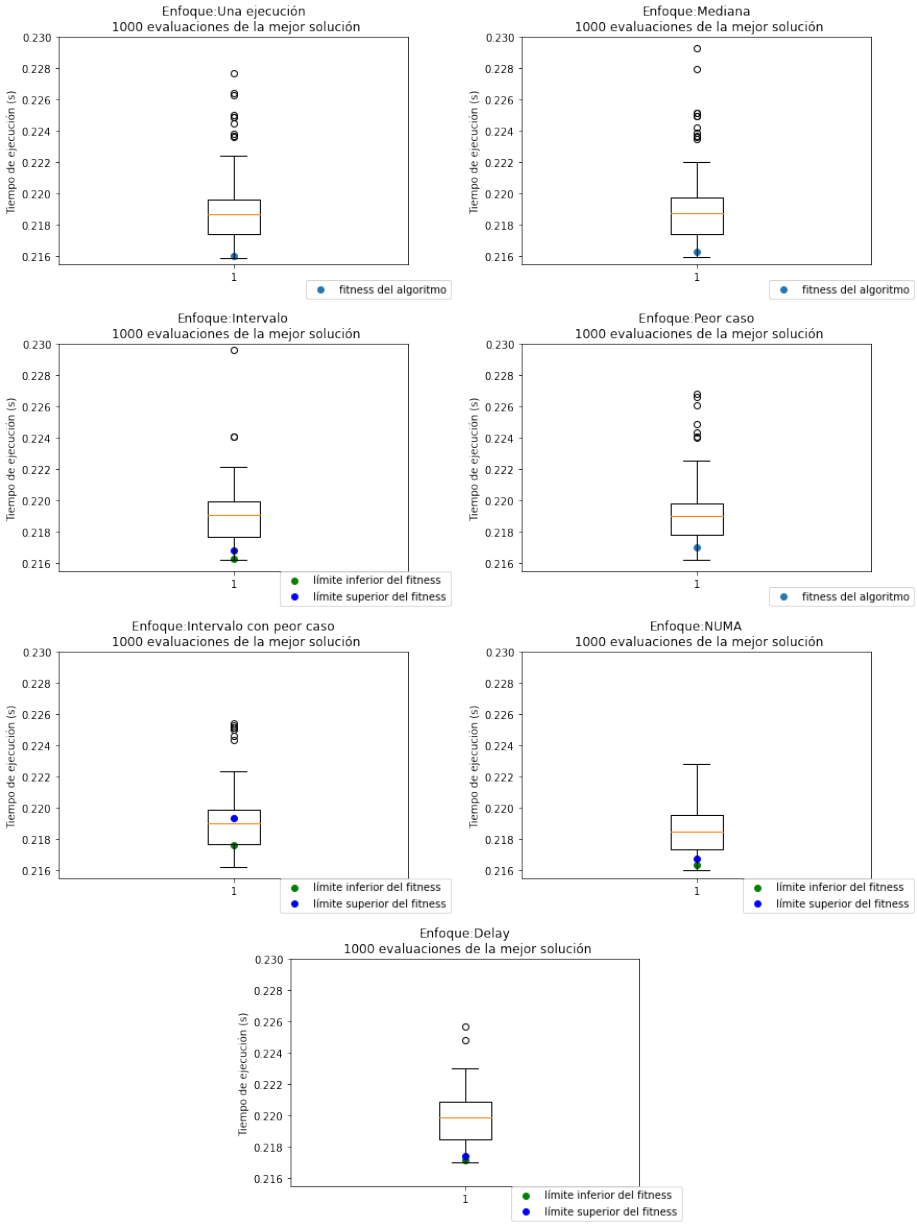

Fig. 2. Gráficas de caja y bigotes de la mejor solución obtenida para cada uno de los valores de fitness considerados.

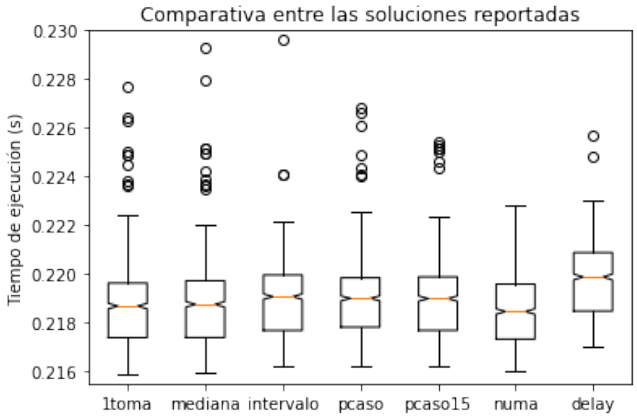

Fig. 3. Comparativa entre las soluciones reportadas por cada enfoque

mejor aproximan el valor de fitness. Se puede apreciar como este último realiza una evaluación altamente precisa del valor de fitness real de la solución durante la ejecución del cGA.

La razón detrás del problema que podemos observar es el número de ejecuciones que realizamos del programa a evaluar. Realizar un número pequeño de ejecuciones nos lleva a la posibilidad de obtener un valor poco representativo del fitness si gran parte de esas ejecuciones se corresponden con tiempos muy reducidos, dentro de su rango de incertidumbre. Además, estas soluciones serán muy competitivas, por lo que difícilmente serán descartadas por el GA. Esto explica el hecho de que los métodos que consideran el peor caso sufran menos de este problema, y que al aumentar el número de ejecuciones disminuya el grado de este problema. No obstante, aumentar el número de ejecuciones conlleva a un mayor coste computacional del cGA.

En la Figura 3 se puede observar una comparativa en relación a la calidad de las diferentes soluciones reportadas por los algoritmos. Para ello se ha ejecutado $1,000$ veces la mejor solución encontrada por cada enfoque, midiendo los tiempos de ejecución. Las cajas cuyos estrechamientos se solapan se corresponden con distribuciones entre las que no hay diferencias significativas. Cabe destacar en este punto que el tiempo de ejecución del programa sin optimizaciones se encuentra por encima de 0.3 segundos. Podemos observar que, independientemente de la precisión del valor de fitness que ofrecen las técnicas empleadas, la calidad de las soluciones reportadas por los enfoques varía en el orden de la milésima. En cualquier caso, se observan diferencias entre los distintos enfoques: ejecutar el cGA y las evaluaciones en diferentes NUMAs lleva, significativamente, a los mejores resultados, mientras que los peores resultados se obtienen con el método que introduce un retardo entre las distintas evaluaciones.

La mejor solución reportada por el cGA mejora en un 33% el tiempo que tarda en ejecutarse el programa sin optimizar.

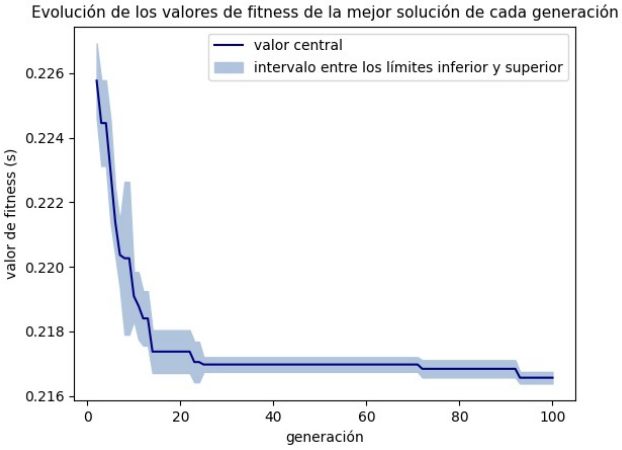

Fig. 4. Curva de convergencia del cGA para el enfoque de NUMAs diferentes

En la Figura 4 se muestra la mejor solución en la población (valor central y su intervalo) del mejor enfoque obtenido en nuestra comparación: numa. Se puede observar cómo la población sigue evolucionando en las últimas generaciones, lo que nos hace pensar que el algoritmo podría encontrar soluciones mejores en caso de contar con un mayor número de evaluaciones límite.

8 Conclusiones y trabajo futuro

En este trabajo proponemos la optimización de programas para una arquitectura específica, modelando el problema como un problema de optimización combinatoria. Se han propuesto y comparado siete métodos diferentes para la estimación del tiempo de ejecución del programa optimizado, que está sujeto a una gran incertidumbre debido a los procesos que coexisten en el sistema, propios del sistema operativo.

Los resultados obtenidos evidencian este problema al sufrir la mayoría de los métodos estudiados una sobreestimación del valor de fitness real del individuo. De entre todas las técnicas utilizadas, la más precisa ha resultado ser aquella en la que el cGA y las evaluaciones de fitness son ejecutadas en NUMAs diferentes, minimizando de esta forma la influencia entre estos procesos.

Como trabajo futuro, nos planteamos ampliar el estudio con la optimización de otros programas SW de diferentes características, así como la evaluación de diferentes arquitecturas HW. También consideramos muy interesante trabajar hacia nuevos métodos capaces de estimar de forma más precisa y con un menor coste computacional el tiempo de ejecución de los programas.

Agradecimientos

Trabajo financiado por Ministerio de Ciencia, Innovación y Universidades y FEDER (RTI2018-100754-B-I00, iSUN), FEDER (FEDER-UCA18-108393, OPTIMALE), y Junta de Andalucía y FEDER (P18-2399, GENIUS). J.C. de la Torre agradece su ayuda al Ministerio de Ciencia, Innovación y Universidades a través del programa FPU (FPU17/00563).

Referencias

1. Alba, E., Dorronsoro, B.: Cellular genetic algorithms, vol. 42. Springer Science & Business Media (2009)
2. Ashouri, A., Bignoli, A., Palermo, G., Silvano, C., Kulkarni, S., Cavazos, J., Bignoli, A., Palermo, G., Silvano, C.: MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. ACM Transactions on Architecture and Code Optimization **14**(29) (2017)
3. Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 257–269. ACM (2015)
4. Benitez-Hidalgo, A., Nebro, A.J., Garcia-Nieto, J., Oregi, I., Del Ser, J.: jmetalpy: A python framework for multi-objective optimization with metaheuristics. Swarm and Evolutionary Computation **51**, 100598 (2019)
5. Bhunia, A.K., Samanta, S.S.: A study of interval metric and its application in multi-objective optimization with interval objectives. Computers & Industrial Engineering **74**, 169–178 (2014)
6. Bruce, B.R., Petke, J., Harman, M.: Reducing energy consumption using genetic improvement. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation. pp. 1327–1334. ACM (2015)
7. Chanas, S., Kuchta, D.: Multiobjective programming in optimization of interval objective functions—a generalized approach. European Journal of Operational Research **94**(3), 594–598 (1996)
8. Delgado-Pérez, P., Medina-Bulo, I., Palomo-Lozano, F., García-Domínguez, A., Domínguez-Jiménez, J.J.: Assessment of class mutation operators for c++ with the mucpp mutation system. Information and Software Technology **81**, 169–184 (2017)
9. Ding, Y., Ansel, J., Veeramachaneni, K., Shen, X., O'Reilly, U.M., Amarasinghe, S.: Autotuning algorithmic choice for input sensitivity. Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) pp. 379–390 (2015)
10. Friedrich, T., Kötzing, T., Krejca, M.S., Sutton, A.M.: The benefit of recombination in noisy evolutionary search. In: International Symposium on Algorithms and Computation. pp. 140–150. Springer (2015)
11. Garciarena, U., Santana, R.: Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference (GECCO). pp. 1159–1166. ACM (2016)
12. He, Z., Yen, G.G., Lv, J.: Evolutionary multiobjective optimization with robustness enhancement. IEEE Transactions on Evolutionary Computation **24**(3), 494–507 (2019)

13. Ishibuchi, H., Tanaka, H.: Multiobjective programming in optimization of the interval objective function. European journal of operational research **48**(2), 219–225 (1990)
14. Jin, Y., Branke, J.: Evolutionary optimization in uncertain environments-a survey. IEEE Transactions on evolutionary computation **9**(3), 303–317 (2005)
15. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. Ieee transactions on software engineering **38**(1), 54 (2012)
16. Lee, C., Lee, J.K., Hwang, T., Tsai, S.C.: Compiler optimization on vliw instruction scheduling for low power. ACM Transactions on Design Automation of Electronic Systems (TODAES) **8**(2), 252–268 (2003)
17. Limbourg, P., Aponte, D.E.S.: An optimization algorithm for imprecise multi-objective problem functions. In: 2005 IEEE Congress on Evolutionary Computation. vol. 1, pp. 459–466. IEEE (2005)
18. LLVM: The LLVM Compiler Infrastructure, llvm.org, Último acceso el 28 de Abril de 2022
19. Louis-Noel Pouchet: PolyBench/C – Homepage of Louis-Noël Pouchet, http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/, Último acceso el 28 de Abril de 202
20. Nimmer, J.: Automatic generation and checking of program specifications. Ph.D. thesis, Massachusetts Institute of Technology, Boston,USA (2002)
21. Pallister, J., Hollis, S., Bennett, J.: BEEBS: Open benchmarks for energy measurements on embedded platforms. arXiv preprint arXiv:1308.5174 (2013)
22. Pinel, F., Dorronsoro, B., Bouvry, P.: The virtual savant: Automatic generation of parallel solvers. Information Sciences **432**, 411–430 (2018)
23. Pinkers, R., Knijnenburg, P., Haneda, M., Wijshoff, H.: Statistical selection of compiler options. Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS pp. 494–501 (2004)
24. Qian, C., Bian, C., Yu, Y., Tang, K., Yao, X.: Analysis of noisy evolutionary optimization when sampling fails. Algorithmica **83**(4), 940–975 (2021)
25. Sandran, T., Zakaria, N., Pal, A.: An optimized tuning of genetic algorithm parameters in compiler flag selection based on compilation and execution duration. In: Proceedings of the International Conference on Soft Computing for Problem Solving (SocProS 2011) December 20-22, 2011. pp. 599–610. Springer (2012)
26. Soares, G.L., Guimarães, F.G., Maia, C.A., Vasconcelos, J.A., Jaulin, L.: Interval robust multi-objective evolutionary algorithm. In: 2009 IEEE Congress on Evolutionary Computation. pp. 1637–1643. IEEE (2009)
27. Torre, J.C.d.l., Ruiz, P., Dorronsoro, B., Galindo, P.L.: Analyzing the influence of llvm code optimization passes on software performance. In: International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems. pp. 272–283. Springer (2018)
28. Tuzov, I., Andrés, D., Ruiz, J.C.: Tuning synthesis flags to optimize implementation goals: Performance and robustness of the LEON3 processor as a case study. Journal of Parallel and Distributed Computing **112**, 84–96 (2018)
29. Varrette, S., Dorronsoro, B., Bouvry, P.: An LLVM-based approach to generate energy aware code by means of MOEAs. In: 7th European Symposium on Computational Intelligence and Mathematics (ESCIM). pp. 198–204 (2015)
30. Wang, N.F., Zhang, X., Yang, Y.: A hybrid genetic algorithm for constrained multi-objective optimization under uncertainty and target matching problems. Applied Soft Computing **13**(8), 3636–3645 (2013)