Paper review: Optimización de Compiladores para tratar ineficiencias en el código generado

Mateo Noel[1], Macarena Oyague[2], Reynaldo Rojas[3] and Francesco Uccelli[4]

***Abstract*—Este es un paper review de los papers [1], [2], [3]. El objetivo es brindar información sobre las ineficiencias que presentan los compiladores hoy en día. Además, se explicará cómo los programadores identifican y solucionan estas mismas. Finalmente, se dará un análisis de las conclusiones de los distintos puntos a tomar en cuenta de los papers junto a cómo se relacionan sus resultados.**

I. Introducción

En las siguientes páginas se realizará énfasis a analizar distintos papers con el fin de comprender diferentes métodos de optimización de compiladores. Además, se hará una crítica respectiva basándonos en las opiniones de los autores.

Nosotros conocemos que el objetivo de un compilador es generar código maquina, este código generado puede contener ineficiencias, las cuales empeorarán la calidad del código. Estas ineficiencias pueden depender de qué tan bien esté escrito el código fuente, pero también pueden existir sobre códigos fuente completamente optimizados. En esos casos el código de máquina es afectado negativamente por cómo ha sido compilado. Posteriormente en el informe veremos cómo es que se encuentran dichas ineficiencias y cómo es que podemos solucionarlas.

II. Overview

- Optimización: Mejorar un código para que su ejecución se logre de manera eficaz y eficiente.
- Ineficiencias en Compiladores: Después de compilar un código, son las instancias que tienen oportunidad de ser optimizadas en el código de máquina.
- Redundancia: Leer o escribir un valor más de una vez sobre un mismo espacio de memoria.
- Vectorización: Un compilador que trabaja con vectorización básicamente convierte loops en secuencias de operaciones de vectores.

III. Background

A continuación se presentarán los temas principales que fueron expuestos en los tres papers seleccionados para expresar el tema en cuestión.

A. What Every Scientific Programmer Should Know About Compiler Optimizations?

Liu, Jiao, Chabbi y Xu brindan una alta gama de información acerca de la optimización de compiladores. Entre ellas es posible encontrar respuestas a cuáles son las ineficiencias que los compiladores generan, el motivo por el cual ocurren las mismas, cómo es que se trabajan, cómo los programadores pueden identificarlas y el papel que cumple el código fuente para para eliminarlas.

Los compiladores usualmente generan ineficiencias o fallan en la fase de optimización por alguno de los siguientes motivos: Dead Store, Redundant Store y Redundant Load. El primero se ocasiona cuando dos operaciones de store ocurren de manera consecutiva hacia un mismo espacio de memoria y no son intervenidas por una operación load al mismo espacio. El segundo caso, por otro lado, se da si una operación store escribe un valor V, en un registro/memoria M; lo cual es redundante si M ya cuenta con el mismo valor V. El tercero, en cambio, ocurre cuando se carga un valor V de un espacio de memoria M y la carga inmediata precedente de M carga el mismo valor V.

El motivo por el cual surgen estas ineficiencias es porque los compiladores las introducen durante la generación del código ya sea durante una optimización conservativa u agresiva, debido a que no se presentan directamente en el código fuente. También existen otros tipos de ineficiencias que son esperadas en ser optimizadas pero no lo son. El paper nos muestra ejemplos de compiladores fallando en eliminar ineficiencias, desde la sección 4.1 hasta la 4.6 se pueden observar 6 distintos casos.

El tiempo de ejecución absoluto de distintos compiladores difiere, ya que estos utilizan distintas técnicas de optimización. Por otro lado, el aceleramiento y porcentaje de reducción en loads y stores es muy similar entre compiladores.

Los programadores podrían identificar estas ineficiencias que no son optimizadas por el compilador a partir del reconocimiento de patrones, aunque no sea fácil encontrarlos. Por ello, existen herramientas que nos ayudan a encontrarlos. Estos no existen en ineficiencias inducidas por los compiladores. Uno puede extraer algunos patrones a partir de un análisis binario, pero punteros y nombres de variables complican el análisis. Una herramienta como CIDetector es

[1]M. Noel es estudiante del Departamento de Ciencia de la Computación, Universidad de Ingeniería y Tecnología - UTEC Lima, Perú mateo.noel@utec.edu.pe

[2]M. Oyague es estudiante del Departamento de Ciencia de la Computación, Universidad de Ingeniería y Tecnología - UTEC Lima, Perú macarena.oyague@utec.edu.pe

[3]R. Rojas es estudiante del Departamento de Ciencia de la Computación, Universidad de Ingeniería y Tecnología - UTEC Lima, Perú reynaldo.rojas@utec.edu.pe

[4]F. Uccelli es estudiante del Departamento de Ciencia de la Computación, Universidad de Ingeniería y Tecnología - UTEC Lima, Perú francesco.uccelli@utec.edu.pe

*Paper review correspondiente al curso Compiladores

necesaria. Esta puede guiar una revisión código fuente en orden de evitar ineficiencias del compilador o producir código que acceda a optimizaciones importantes.

Modificar el código fuente puede eliminar ineficiencias ya sea para compiler-introduced o compiler-missed, ya que pueden ser removidas optimizando el código fuente. En la sección 4.1 del paper se muestra que necesitas cambiar unas cuantas líneas de código (entre 2 y 62) para lograr la optimización.

La optimización del código fuente seguirá siendo eficiente en compilaciones de distintos compiladores en algunos casos, como el problema Hmmer del paper no se optimiza en el compilador icc. A veces la vectorización sobrepasa el tiempo de lidiar con redundancias. A pesar de ello, al fin y al cabo siempre hay que escribir código para el compilador que se piensa usar.

De esta fuente es posible rescatar que existen ineficiencias en códigos optimizados de forma binaria; compiladores modernos no logran eliminar varias ineficiencias, peor aún, algunos de estos introducen más. Estos compiladores varían entre distintas versiones del compilador gcc y la última versión del gcc, icc y el llvm. Para los expertos, es importante tener en cuenta estas ineficiencias que solo se encuentran visibles en el código binario, se pueden usar herramientas para arreglar dichos errores, como:

- DeadSpy: A Tool to Pinpoint Program Inefficiencies
- REDSPY: exploring value locality in software
- Runtime Value Numbering: A Profiling Technique to Pinpoint Redundant Computations.

Además, los autores crearon dos herramientas: CIDetector y CIBench, las cuales estudian optimizaciones compiler-introduced y compiler-missed en código binario.

Tabla de optimizaciones en distintas estructuras de código:

Table 2: Optimization capabilities of different compilers.

Program Name	Hmmer	Srad_v2	Bzip2	LavaMD	Backprop	H264ref	Hotspot3D	Hoard	MSB1	MSB2	FFT	Povray	Chroma
gcc 9.3	N	S	S	N	N	N	P	N	S	N	N	N	N
icc 19.1	N	S	S	N	N	N	P	N	P	N	N	N	N
llvm 9.0	N	S	S	N	N	P	P	N	S	N	N	S	N

N: Not Optimized; P: Partially Optimized; S: Successfully Optimized.

B. A Collaborative Filtering Approach for the Automatic Tuning of Compiler Optimizations

En este estudio, se propone una técnica para encontrar un set de optimizaciones para la compilación de un programa que se basa en información disponible de los tiempos de ejecución de programas compilados con distintos sets de optimizaciones. Esto se debe a que la selección de las optimizaciones correctas tiene un gran impacto en la performance de un programa. Además, la técnica propuesta utiliza una similitud entre dos programas que se calcula según las mejoras en tiempo de ejecución que experimentan con el mismo set de optimizaciones. La técnica propuesta logra mejores resultados que muchas técnicas del estado del arte con menos iteraciones. Esto es, necesita realizar menos compilaciones y tests de performance.

La técnica propuesta, conocida como Collaborative Filtering (CF), es derivada del campo de Recommender Systems (RS), pero adaptada al problema de autotuning en términos de la selección de flags. Esta técnica es contrastada con otras técnicas también derivadas del mismo campo como son Top Popular y Content-Based Filtering. Esta técnica hace de un approach llamado Reaction Matching (RM). Este approach, a diferencia de muchos otros utilizados, no categoriza los programas según alguna métrica de performance, sino en términos de cómo reacciona a diferentes optimizaciones. Además, se combinan distintos algoritmos del estado del arte.

La tarea principal de un RS es realizar sugerencias basándose en similitudes entre ítems o usuarios. Una primera idea de un RS, puede ser sugerir los ítems populares. En el caso de optimizaciones, esta idea sería probar las optimizaciones populares. Es decir, utilizar optimizaciones que resultan en una buena performance en muchos programas. Esta técnica es conocida con el nombre Top Popular (TP). Una segunda idea es la de usar vectores característicos obtenidos mediante métricas MICA como en Cobayn (citado por los autores) para los programas e incluir el uso de PCA con el fin de realizar una medida de la distancia entre dos programas mediante los componentes principales de sus vectores característicos. Esta técnica es usada en Content-Based Filtering. Finalmente, tenemos al principal algoritmo propuesto: Collaborative Filtering. Como ya se mencionó, este algoritmo utiliza RM, lo cual quiere decir que se asume que si dos programas reaccionan similar a un mismo conjunto de optimizaciones, estos reaccionarán de forma similar a otros conjuntos de optimizaciones. Cabe señalar que CF utiliza la información adquirida en iteraciones pasadas y por eso sus resultados suelen mejorar con cada iteración. Además, se conjetura que el funcionamiento interno de ambos guarda similitudes debido a esto.

Para terminar el estudio, se realizaron benchmarks con cBench y PolyBench en los que se comparan las distintas técnicas propuestas aquí, además de Cobayn. Los resultados indican que CF encuentra consistentemente las mejores optimizaciones con solo 5 iteraciones de su algoritmo.

C. Compiler 2.0: Using Machine Learning to Modernize Compiler Technology

Este paper trata de optimización en compiladores y plantea, en términos generales, un método de optimización y de medición de rendimiento en compiladores modernos. Es un paper sobre un keynote en MIT que se dio en junio del 2020.

El paper comienza justificando la necesidad de optimización en compiladores al plantear la similitud que tienen los compiladores actuales con los usados en los años 80, específicamente de la arquitectura del compilador LLVM. A continuación, el autor expone todos los recursos que se han adquirido en los últimos años que podrían ser aplicados a optimización de compiladores, como por ejemplo, mejor uso de datos y más poder computacional.

En segundo lugar, discute sobre las dificultades de optimizar un compilador manualmente. Acerca de ello resalta que puede resultar muy tedioso de desarrollar y mantener, y que son susceptibles a "bit rot" y a errores de modelado.

En tercer lugar, desarrolla ideas sobre técnicas de optimización pasadas. Particularmente, la vectorización que, en pocas palabras, es una transformación de un loop a un vector de instrucciones. El punto principal es plantear un modelo de vectorización actualizado usando machine learning para obtener mejores resultados de performance en modelos de testeo actuales, los cuales también se encuentran planteados en el paper. Para desarrollar este modelo, se ha usado ILP (integer linear programming) y se ha mejorado el performance de compiladores que antiguamente usaban vectorización. Continuando con la idea de mejorar actualizar y mejorar el modelo de vectorización, el autor propone un modelo con machine learning, usando redes neuronales para calcular el costo de cada operación de vectorización. Para empezar a evaluar el modelo, dice el autor, que es necesario modificar los modelos de costo, en este caso priorizando la reducción de runtime. Entonces surge la siguiente pregunta: ¿se puede entrenar a una NN para calcular el costo de ejecución dada una secuencia de instrucciones? La respuesta resulta ser afirmativa, ya que es significativamente más rápido y reduce la probabilidad de error humano y de modelo. Este modelo de costo es el que se va a usar para evaluar el compilador con vectorización modernizada.

D. Visión general de técnicas de optimización de código

En el capítulo Nro. 8 de [4] se expresan técnicas de optimización de código. Dentro de las cuales es posible encontrar la asignación de registros, el análisis del código y el comportamiento del programa predictivo.

Lo que realizan los compiladores es un análisis del código fuente en conjunto con las estructuras de datos que lo representan para así implementar transformaciones de mejoramiento de código. Sin embargo, es importante tomar en cuenta que, a pesar de ser un proceso complejo, las optimizaciones que produce pueden ser relativamente pequeñas. Debido a que la calidad de un código generado por un compilador tiene como unidad de medida fundamental la velocidad de ejecución o el tamaño del código generado, resulta importante considerar el tradeoff que se origina entre la complejidad y el tiempo que agrega al compilador, y los resultados de optimización.

La asignación de registros dependerá de los estándares de uso determinados que el compilador debe definir. Por un lado, es importante realizar un buen uso de los registros, ya que son las variables que se encuentran más accesibles al microprocesador y, por ello, son también aquellas que el mismo puede procesar y operar de manera más rápida. Por otro lado, otro factor a tomar en cuenta es la disponibilidad de registros del procesador y el uso eficaz de las operaciones que posee la arquitectura, debido a que no todas las plataformas y arquitecturas disponen de la misma cantidad de registros y las mismas operaciones en el set de instrucciones. Con el paso del tiempo y la evolución de los microprocesadores, ha aumentado el número de registros y la diversidad de instrucciones. Por ello, los compiladores deben buscar el provecho de utilizar más registros para mejorar el performance y utilizar aquellas instrucciones más específicas que ha ido adquiriendo, ya que se puede reducir el número de instrucciones que se realizan de una manera más directa.

Respecto al análisis de código, el compilador debe evitar que se genere código destino para operaciones que son redundantes o innecesarias. Para ello, se puede optar por nuevas reglas para la fase de optimización, de tal modo que se puedan eliminar sub-expresiones que ya hayan sido calculadas dentro del scope, que aparecen repetidas en un segmento de código con un valor que no será utilizado o que permanecerá constante y operaciones que no serán alcanzadas que se les conoce como *código muerto.* Como se dijo anteriormente, hay ocasiones en las que se debe atravesar mayor complejidad para identificar este tipo de operaciones que ahorrarán pocas instrucciones, como se dijo anteriormente. Por ello, dependerá mucho el nivel de profundidad que se quiera tener en la implementación para evaluar si es que se decide considerar dicha complejidad.

Otra técnica que repercute a lo que es análisis de código es evitar operaciones costosas. Para ello, se debe hacer un uso de las instrucciones de la manera más eficientemente posible, tomando en consideración cuáles serían las más apropiadas para incluir en el diseño de generación de código máquina de tal modo que estas consuman una menor cantidad de ciclos de reloj del microprocesador, y a su vez reduzcan el costo de ciertas operaciones. Esto va directamente relacionado con el punto que se expuso anteriormente del uso eficaz de las operaciones que posee la arquitectura. En la fuente se puede leer a mayor detalle ejemplos de la presente técnica, como por ejemplo como es *constant folding, constant propagation, procedure inlining, tail recursion, entre otras.*

Por último, la técnica de comportamiento del programa predictivo se basa en grabar de manera estadística qué funciones son llamadas más veces, en la compilación de qué funciones pasa un mayor tiempo, en cuáles uno menor, las

trayectorias que tienen mas probabilidad de ser tomadas, entre otras. El motivo de ello es realizar un análisis para realizar optimización tan solo las secciones de código en las cuáles está habiendo una demora significativa, en lugar de en aquellas en las que estas optimizaciones puedan ser imperceptibles. De este modo, la información información sirve para optimizar los puntos más críticos.

IV. Realimentación

Luego de entender el enfoque de cada investigación obtenemos distintas críticas y opiniones que se formularán a lo largo de este segmento. Empezaremos describiendo las relaciones entre los 3 papers y terminaremos aclarando información resaltante obtenida.

Los 3 papers están relacionados en el ámbito que son distintos acercamientos a la optimización de compiladores. El paper [1], [2], [3] se enfocan en optimizar el código de máquina generado por el compilador. Cada investigación resulta en la creación de herramientas utilizadas para atacar ineficiencias:

- Se crea la herramienta CIDetector, la cuale hace un análisis binario del código a procesar y menciona dónde existen las ineficiencias de tipo redundante ya sean compiler-missed o compiler-itroduced. La herramienta CIBench se encarga de modificar el código del compilador y arreglar las ineficiencias.
- En el segundo paper se plantea el uso de goSLP , un nuevo método de Super Word Level Parallelism , que utiliza técnicas de programación lineal para encontrar un empaquetamiento óptimo de instrucciones. Además, se habla de usar Vemal, un política aprendida de vectorización en vez de con heurísticas (como se hacia antes).
- En el paper de Cereda y los demás autores, se propone Collaborative FIltering como un algoritmo para encontrar un set de optimizaciones óptimo basándose en el campo de Recommender Systems. Este algoritmo permite encontrar de manera eficiente un conjunto de optimizaciones para un programa en específico.

Los papers nos proveen métodos y herramientas para la generación más óptima de código de máquina. Estos métodos no solamente generan código que se ejecute de manera eficiente, sino también encuentran estas optimizaciones de manera rápida. En el caso del paper de Machine Learning, nos da una perspectiva nueva de cómo se puede atacar el problema de optimización. En el paper de Collaborative Filtering se introduce por primera vez el campo de Recommender Systems al campo de optimización de compiladores. Finalmente, en el paper en el que se introduce CIBench, se realiza las optimizaciones necesarias de una manera más adhoc.

Con respecto a las conclusiones de los papers, podemos mencionar lo siguiente. El paper de Compiler 2.0 y el de CF comparten la similitud de que introducen la intersección de dos campos. En el primer caso, introducen nuevas ideas traídas de machine learning y en el segundo, introducen el campo de Recommender Systems ambos a la optimización de compiladores. Además el paper de CF, agrega 3 algoritmos basados en el campo mencionado. Es importante esta intersección debido a que da paso al estudio de nuevas técnicas de optimización y a un nuevo campo de especialización.

El paper [1] otorga respuestas directas a dudas planteadas por ejemplo a las preguntas ¿Es posible remover ineficiencias al modificar el código fuente? o ¿Existen ineficiencias en códigos binarios completamente optimizados?. También te otorga herramientas específicas para solucionar los problemas que ellos han tratado. Es muy amigable para quien desee tratar estos problemas a futuro. Te plantea el problema, te define soluciones para algunas ineficiencias específicas. Luego te otorga herramientas que encuentran las ineficiencias.

V. Insights

Generamos insights respectivos a lo que sería optimización de compiladores. La información que se otorgará puede ser utilizada por científicos de la computación para entender sobre ineficiencias dentro de compiladores.

Los compiladores cuentan con ineficiencias. A partir de lo investigado obtenemos que no solo los compiladores casi siempre cuentan con ineficiencias, sino que existen distintos tipos de estos. En el paper [1] se habla sobre ineficiencias de redundancia, donde un valor se lee o se escribe más de una vez en un mismo espacio de memoria. En el paper [3] se habla sobre la ineficiencia de los modelos de vectorización antiguos, específicamente, la necesidad de que hayan vectores demasiado grandes para que la optimización valga la pena.Además ,estos modelos de vectorización necesitan vectorizar necesariamente un loop completo para generar estos vectores largos. En el paper [2], no señalan ineficiencias específicas del código generado, pero sí mencionan el hecho que el código generado por las optimizaciones genéricas no siempre es el eficiente y por ello tratan de solucionar esto mediante la exploración y proposición de algoritmos nuevos para encontrar optimizaciones.

Compiladores utilizados hoy en día son ineficientes hasta cierto nivel. Los compiladores realizan ciertas optimizaciones que generalmente, son las que otorgan buenos resultados con la mayoría de los programas con los que se prueban. Esta forma de realizar la compilación es muy similar también, a uno de los métodos propuestos: Top Popular. No obstante, esta selección de optimizaciones no resulta ser la adecuada en muchos casos. Por otro lado, se puede agregar que los compiladores modernos aun usan tecnología que existía hace décadas. Entre estas, algoritmos básicos, técnicas de parseo, análisis de dependencias de datos, vectorización y varias

más. El paper [3] habla sobre las dificultades de optimizar un compilador manualmente, puede ser muy tedioso de desarrollar y mantener, son susceptibles a "bit rot" y a errores de modelado.

Encontrando ineficiencias. En el paper [2] no se centran en encontrar las ineficiencias del código de máquina. Es más como una búsqueda ciega utilizando similitudes de las reacciones de los programas a distintos sets de optimizaciones. El paper [1] no menciona sobre un análisis dinámico binario, el cual su herramienta CIDetector, realizará. Esta herramienta obtiene información del código binario del código fuente e identifica las ineficiencias que el compilador puede generar sobre este. En el paper [3] no se habla específicamente de encontrar ineficiencias en el código, sino cambia el modelo completo de compilación. Se usa goSLP, que a diferencia de SLP, es paralelismo globalmente optimizado. Con este modelo se soluciona el problema de empaquetar bloques de instrucciones de manera eficiente.

Estrategias para resolver dichas ineficiencias. La estrategia utilizada en CF [2] entonces, ya que se basa en RS, se trata de encontrar similitudes entre programas para aplicar las optimizaciones que se saben que funcionan en un conjunto de programas. Se prueban con distintas optimizaciones y se va aprovechando información de iteraciones pasadas para encontrar similitudes. En el paper [1] te otorgan la herramienta CIDetector, la cuales ayuda a detectar las ineficiencias, a travez de esto puedes usar distintas técnicas en el código del compilador para llegar a optimizar el código de máquina, CIBench soluciona algunas de estas.

VI. Acercamiento

En el paper [2] a diferencia de lo aprendido en clase, la optimizaciones se realizan a más alto nivel. Esto es, no se examina a detalle el código de máquina y solo se ejecutan simulaciones con distintos conjuntos de flags de compilación, y se va encontrando similitudes entre programas según RM. Finalmente, como un RS, se obtiene un conjunto de optimizaciones para un programa basado en información pasada y en similitudes.

VII. Conclusión

El problema de optimización en compiladores se puede resolver desde varios enfoques y es un área de investigación aún bastante activa por la gran cantidad de acercamientos que se pueden tomar. Hemos visto que las técnicas usadas pueden abarcar desde el análisis del código de manera adhoc programada por un especialista en compiladores para una arquitectura en específico. No obstante, también se puede realizar un análisis a alto nivel con el uso de otros algoritmos para la selección de flags de compilación, como Collaborative Filtering, Top Popular, Cobayn, entre otros. También se puede usar Machine Learning para la optimización de código y entrenar modelos según la arquitectura target que se tenga y de esta forma se evita también, que los especialistas en compiladores tengan que estar de una forma u otra compitiendo con hardware nuevo.

References

[1] Jialiang Tan, Shuyin Jiao, Milind Chabbi, and Xu Liu. What Every Scientific Programmer Should Know About Compiler Optimizations?. Barcelona, Spain. June 29 - July 2, 2020.

[2] Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. 2020. A Collaborative Filtering Approach for the Automatic Tuning of Compiler Optimisations. In Proceedings of 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, London, United Kingdom, June 16, 2020 (LCTES '20), 10 pages.

[3] Compiler 2.0: Using Machine Learning to Modernize Compiler Technology , Saman Amarasinghe

[4] K. C. Louden. Compiler Principles and Practice. USA: PWS Publishing Company, 1997. Capítulo 8.9.