TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE OAXACA

CARRERA: INGENIERÍA EN SISTEMAS COMPUTACIONALES

PROGRAMACION LOGICA Y FUNCIONAL

Lenguajes Y Automatas II

Alumna:

Santiago Jimenez Galilea

DOCENTE: ALONSO HERNANDEZ LUIS ALBERTO

GRUPO 7SB

OAXACA DE JUÁREZ, 10 DE OCTUBRE DE 2025

Introducción

La **optimización de código** en compiladores es el proceso de transformar un programa en otro semánticamente equivalente pero más eficiente. Su finalidad principal es mejorar el rendimiento (por ejemplo, reducir el tiempo de ejecución, el espacio de memoria o el consumo de energía) sin cambiar el resultado final. Esto es crucial para generar software rápido y ligero; sin optimización, un compilador produciría código correcto pero potencialmente ineficiente. Sin embargo, la optimización es compleja: se sabe que muchos problemas relacionados son *NP*-completos o indecidibles, por lo que los compiladores utilizan heurísticas que no garantizan una solución globalmente óptima. En la práctica, la optimización suele ser un paso adicional costoso en tiempo de compilación, pero sus beneficios a nivel de ejecución hacen que valga la pena en la mayoría de aplicaciones.

Desarrollo

Concepto de optimización de código

La optimización de código consiste en aplicar transformaciones al programa para producir otro fragmento *equivalente en funcionalidad* pero que aproveche mejor los recursoses. Así, el compilador optimizador «trata de minimizar ciertos atributos de un programa informático con el fin de aumentar la eficiencia y rendimiento». En otras palabras, se busca reducir el tiempo de ejecución, el espacio en memoria y/o el tamaño final del programa. Estas transformaciones son cuidadosamente diseñadas para que el código resultante se comporte idénticamente al original, excepto por ser más rápido o ocupar menos espacio. Dado que lograr un óptimo global es inviable en general, las optimizaciones se basan en reglas y análisis (como análisis de flujo de datos) que ofrecen mejoras apreciables sin cambiar la salida del programa.

Optimización local vs. global

Las optimizaciones se clasifican comúnmente según su alcance (o **ámbito**). En general:

- **Local:** se aplica dentro de un bloque básico, es decir, a un fragmento de código con una sola entrada y salida sin saltos internos. Al no cruzar los límites de esos bloques básicos, estas técnicas requieren muy poco análisis adicional, por lo que son rápidas de ejecutar. Sin embargo, su impacto global es limitado, pues sólo mejoran trozos pequeños de código. Ejemplos son el plegamiento de constantes, propagación de constantes y copias dentro de un bloque, o la eliminación de subexpresiones comunes locales.
- **Global (intraprograma):** considera una función o procedimiento completo. Este análisis es más costoso computacionalmente, pues debe seguir caminos de control entre bloques básicos y calcular datos globales. La ventaja es que con más información puede realizar transformaciones más potentes (por ejemplo, propagar

constantes a través de todo el método) y lograr mayores ganancias de rendimiento. La desventaja es que aumenta considerablemente el tiempo de compilación y el uso de memoria del compilador.

En resumen, las optimizaciones locales son más sencillas pero con beneficios pequeños, mientras que las globales son más complejas y costosas pero normalmente ofrecen mejoras más notables. Existen además las **interprocedimentales**, que analizan todo el programa entre funciones, pero éstas suelen ser muy lentas y se usan principalmente en entornos donde el rendimiento final es crítico.

Regiones básicas (Bloques básicos)

Un **bloque básico** es una secuencia de instrucciones que se ejecutan de manera consecutiva y lineal, con un único punto de entrada y uno de salida. Esto significa que, dentro del bloque, no hay saltos condicionales o incondicionales hasta su final. Los bloques básicos son unidades fundamentales en el diseño de compiladores porque simplifican el análisis: se garantiza que si se entra en el bloque se ejecutan todas sus instrucciones en orden. Gracias a esto, muchas optimizaciones (como propagación de constantes o algebraicas) pueden aplicarse dentro de cada bloque básico con la seguridad de que el código no diverge inesperadamente. Por ejemplo, para dividir el código en bloques básicos se identifica cada primer instruction (líder) y se agrupan las instrucciones hasta el siguiente salto. Utilizar bloques básicos permite construir el grafo de flujo de control, donde cada nodo es un bloque, lo que facilita análisis posteriores como determinación de valores activos o puntos de inserción para optimizaciones.

Eliminación de subexpresiones comunes

La **eliminación de subexpresiones comunes** (Common Subexpression Elimination, CSE) detecta expresiones aritméticas o lógicas idénticas que se calculan más de una vez en un mismo contexto, y reemplaza los cálculos redundantes por el uso de un único valor precomputado. Por ejemplo, considere el código:

```
a = b * c + g;

d = b * c * e;
```

Aquí la subexpresión b * c se repite. Aplicando CSE se introduce una variable temporal para almacenarla una vez, obteniendo:

```
tmp = b * c;

a = tmp + g;

d = tmp * e;
```

Con esto se evita recalcular b * c dos veces. Como señala la literatura, esto es rentable siempre que el coste de almacenar y leer la variable temporal sea menor que volver a calcular la expresión. En la práctica la CSE reduce el número de operaciones costosas y acelera el programa, aunque introduce la sobrecarga de manejar variables adicionales. Además, sólo se aplica cuando se comprueba que las subexpresiones repetidas son efectivamente idénticas (mismo orden de operaciones y valores) y seguras de reordenar. Una desventaja de la CSE es que, si el cálculo es muy barato o la expresión tiene efectos secundarios, puede no convenir reemplazarla. Por ello, los compiladores suelen aplicar CSE en etapas intermedias donde se garantiza la pureza de operaciones y en contextos donde el ahorro sea significativo.

Propagación de constantes y copias

Propagación de constantes: Esta técnica sustituye variables que tienen valores constantes conocidos por esos mismos valores. Por ejemplo, si se determina que x = 5 en un punto anterior del código, luego cualquier uso de x en una expresión se reemplaza por 5. El compilador puede incluso pre-calcular expresiones constantes (plegamiento de constantes). Como indica IBM, "las constantes utilizadas en una expresión se combinan y se generan otras nuevas" durante la optimización. Así, las expresiones se simplifican en tiempo de compilación, reduciendo cálculos en tiempo de ejecución. El beneficio es una reducción directa en el trabajo posterior; la limitación es que sólo se puede aplicar cuando el valor de la variable realmente es constante en ese punto.

Propagación de copias: Esta técnica elimina variables intermedias que son copias simples de otras. Es decir, si existe una asignación f = a, el compilador reemplaza las referencias a f usando directamente a en las expresiones subsecuentes, siempre que sea seguro hacerlo. Por ejemplo:

```
a = b + c;
  f = a;
e = f + d;
```

se puede transformar en

```
a = b + c;
e = a + d;
```

eliminando la variable f. Como explica GeeksforGeeks, la propagación de copias "reemplaza las asignaciones irrelevantes por los valores previamente calculados". Esto limpia el código, reduce variables temporales innecesarias y puede habilitar otras optimizaciones (por ejemplo, convertir copias redundantes en código muerto). La desventaja es que, si hay alias (dos nombres que refieren al mismo dato) o dependencias complejas, no siempre es seguro propagar la copia. Además, en ámbitos globales es más complejo rastrear copias entre

bloques. En general, ambas propagaciones (constantes y copias) mejoran la eficiencia, pero se aplican donde los análisis de flujo de datos así lo permiten.

Eliminación de código muerto

La **eliminación de código muerto** descarta instrucciones cuyos efectos no influyen en el resultado del programa. Por ejemplo, asignaciones a variables que luego nunca se usan, o ramas a las que nunca se accede. Según IBM, esto implica "eliminar el código al que no se puede acceder o cuyos resultados no se utilizan posteriormente". Para identificar código muerto se utiliza típicamente el *análisis de variable viva*: se determina qué variables son utilizadas en el futuro y se borra todo lo demás. Esto reduce el tamaño del programa y evita ejecuciones inútiles, mejorando el rendimiento global. Sin embargo, hay que tener cuidado: si el código eliminado tenía efectos colaterales (por ejemplo, lanzaba excepciones o modificaba un estado global), su remoción podría alterar el comportamiento del programa. La literatura advierte que *la eliminación de código muerto puede cambiar la salida del programa* si, por ejemplo, el código eliminado incluía una operación de división por cero que daba un error. Por ello, los compiladores solo eliminan código muerto cuando se verifica que es seguro (por ejemplo, que no involucra operaciones con efectos).

Reordenamiento de instrucciones

La **planificación o reordenamiento de instrucciones** (instruction scheduling) consiste en cambiar el orden de las instrucciones *manteniendo la semántica*, con el fin de mejorar la ejecución en tiempo de ejecución. Por ejemplo, se puede intercalar instrucciones independientes para aprovechar mejor la unidad aritmética de la CPU o evitar "huecos" en la canalización (pipeline). IBM describe esta optimización como: "Reordena las instrucciones para minimizar el tiempo de ejecución". Un caso concreto es el *code motion* dentro de bucles: si una expresión dentro de un bucle usa únicamente variables que no cambian en cada iteración, puede calcularse una sola vez fuera del bucleibm.com. Esto reduce trabajo repetido en cada iteración. La ventaja principal del reordenamiento es aprovechar paralelismo a nivel de instrucción, especialmente en arquitecturas pipelined o con varios núcleos. Sin embargo, la desventaja es que hay que respetar todas las dependencias de datos y control; no se puede mover una instrucción que escriba en memoria antes de otra que lea de ella, por ejemplo. Además, esta optimización se realiza típicamente en la etapa final del compilador (back-end) porque depende de la arquitectura destino (registros, latencias, etc.). En resumen, la reordenación acelera el código al explotar mejor el hardware, pero su aplicación está limitada por dependencias internas de las instrucciones.

Conclusiones

La optimización de código es, sin duda, una etapa fundamental para obtener programas eficientes. He constatado que cada técnica tiene un propósito claro: reducir cálculos redundantes, eliminar instrucciones inútiles o mejorar el uso de recursos del CPU. Entiendo ahora que las optimizaciones locales son simples y rápidas de aplicar, pero suelen aportar mejoras modestas, mientras que las globales exigen mayor esfuerzo analítico pero pueden incrementar mucho el rendimiento global del programa. En el futuro, considero que usaría estas técnicas combinadas: aplicar primero optimizaciones locales (como plegamiento de constantes y propagación en bloques básicos) para "limpiar" el código, y luego análisis globales más costosos donde realmente haga falta (como propagación global de constantes o análisis interprocedimental). Además, aprendí que es crucial validar que no se altere la semántica, especialmente en optimizaciones agresivas como eliminación de código o reordenamiento. En conclusión, la optimización en compiladores busca un balance entre coste de compilación y beneficio en ejecución, y conocer cada técnica me permite elegir sabiamente cuándo usar cada una.

Referencias

- Compilador optimizador. (s.f.). *Wikipedia, la enciclopedia libre*. Recuperado de https://es.wikipedia.org/wiki/Compilador_optimizador
- Eliminación de subexpresiones comunes. (s.f.). *Wikipedia, la enciclopedia libre*. Recuperado de https://es.wikipedia.org/wiki/Eliminaci%C3%B3n_de_subexpresiones_comunes
- Código muerto. (s.f.). *Wikipedia, la enciclopedia libre*. Recuperado de https://es.wikipedia.org/wiki/C%C3%B3digo_muerto
- IBM Corporation. (s.f.). *Compilación con optimización*. IBM Knowledge Center. Recuperado de https://www.ibm.com/docs/es/aix/7.2.0?topic=techniques-compiling-optimization
- Bello, R. (s.f.). *Antología de compiladores*. Instituto de Ciencias de la Computación, BUAP. Recuperado de https://www.cs.buap.mx/~pbello/antcomp.pdf