Técnicas de Alto Rendimiento en el Diseño de Procesadores

“Flags de Optimización con GCC y su repercusión en el rendimiento de la CPU”

José Luis Vázquez Poletti
Doctorado 2004/2005

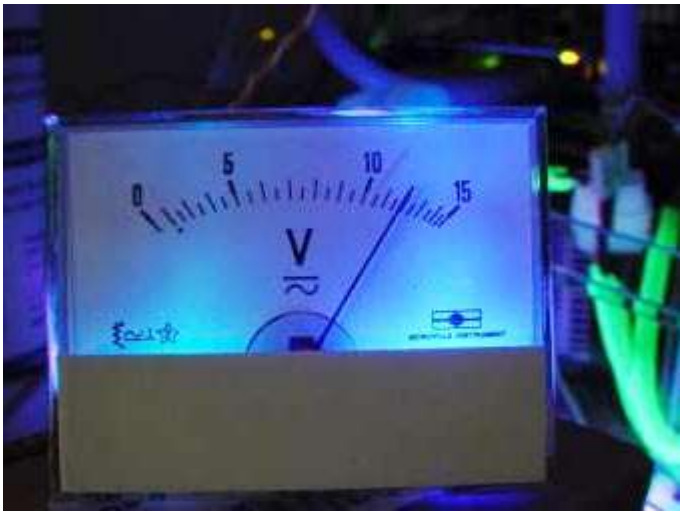
ÍNDICE

INTRODUCCIÓN A LAS OPTIMIZACIONES

Sin ninguna opción de optimización, el objetivo del compilador consiste en reducir el tiempo de compilación, y hacer posible la depuración y que ésta produzca los resultados esperados.

Si se parara el programa con un “breakpoint” entre instrucciones, se puede asignar un nuevo valor a cualquier variable o cambiar el contador de programa para que apunte a otra instrucción diferente y obtener exactamente los resultados que se deberían esperar a partir del código fuente.

Usar las opciones de optimización hace que el compilador intente mejorar el rendimiento y/o el tamaño del código a cambio de más tiempo de compilación y la imposiblidad de depurar el programa.



No todas las optimizaciones, como veremos en este pequeño estudio, consiguen su objetivo. Pero evidentemente, hay que considerar el entorno en el que se realizan.

Introducción a los benchmarks

El entorno elegido para realizar las pruebas que componen este estudio es el de SPEC2000.

SPEC es el acrónimo de Standard Performance Evaluation Corporation, y se trata de una organización sin ánimo de lucro compuesta de fabricantes, integradores de sistemas, universidades, investigadores, ... con el objetivo de establecer, mantener y promover una serie de benchmarks relevantes.

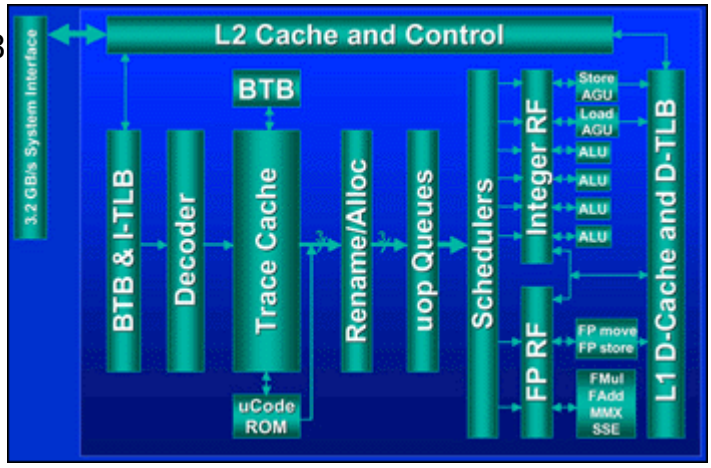
Los benchmarks son versiones particulares o reducidas de programas conocidos que permiten proporcionar datos interesantes a la hora de comparar diferentes sistemas. En este estudio se han escogido dos en particular, uno de coma flotante y otro entero: ART (Redes Neuronales para el reconocimiento de imágenes) y VPR (Ubicación y conexión de circuitos FPGA).

Evidentemente, los resultados variarían de haberse cogido otros, pero la elección de los benchmarks ha sido en relación a alguna de las actividades de Investigación y asignaturas del Plan de Doctorado en el Departamento de Arquitectura de Computadores y Automática.

El entorno de pruebas

La máquina empleada en las pruebas es un portátil ACER Aspire 1661WLM. Sus características técnicas son:

- **Procesador:** Intel Pentium 4 a 2,8 Ghz (Para las pruebas se desactivó el hyperthreading)
- **L1 I Cache (Trace Cache):** 12 K µops
- **L1 D Cache:** 8 K
- **L2 Cache:** 512 K
- **Sistema Operativo:** Mandrake (ahora, Mandriva) Linux – Kernel 2.6.8.1-12 mdkenterprise



Cabe reseñar que para mayor fiabilidad de las pruebas, se realizaron empleando la interfaz de comandos, deshabilitando las X-Windows.

El dato que se ha medido ha sido el tiempo de ejecución de los benchmarks. El procedimiento ha consistido en lanzarlos cada vez con una opción de optimización diferente, puesto que el SPEC2000 no ha sido diseñado para este modo de pruebas.

GCC

GCC es el acrónimo de GNU Compiler Collection, aunque antes la C correspondía exclusivamente a ese lenguaje de programación.

El GCC actual consiste en una colección de compiladores, diseñados dentro del Proyecto GNU. En este estudio se ha usado exclusivamente el compilador de C, aunque, como se ha dicho antes, se da soporte a otros: Ada, C++, Fortran, Java, Objective C y Pascal.

La primera versión fue escrita por Richard Stallman. GCC funciona en muchas arquitecturas y sistemas operativos, y requiere el conjunto de aplicaciones conocido como “binutils” para realizar tareas como identificar archivos objeto u obtener su tamaño para copiarlos, traducirlos o crear listas, enlazarlos, o quitarles símbolos innecesarios.

El mayor mérito técnico de GCC es quizás su arquitectura. Dividido entre “frontend” y “backend”, posee soporte para diversos lenguajes de programación. Al mismo tiempo, es facilmente adaptable a nuevas plataformas, haciéndose disponible en una multitud de combinaciones de arquitecturas de computadoras y sistemas operativos.

CONSIDERACIONES SOBRE LA CLASIFICACIÓN DE LOS RESULTADOS

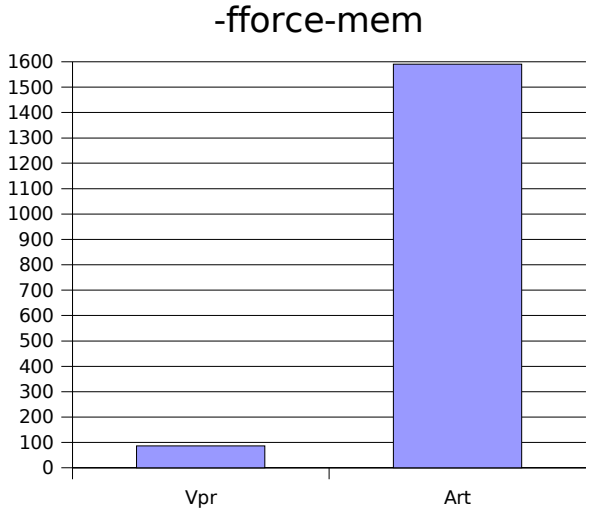
Al ser el objetivo de este trabajo, una comparativa de cada opción de optimización, se optado por clasificar inicialmente los resultados dado el pograma que más sale ganando de los dos (VPR o ART) a la hora de medir su tiempo de ejecución (en segundos), sin atender a las diferencias obvias en su estructura. En esta primera clasificación (midiéndose el tiempo en segundos), ni siquiera se ha considerado los tiempos obtenidos sin usar optimización alguna. Además, esta clasificación tiene su justificación en las desigualdades encontradas en el tiempo de ejecución de cada programa con cada optimización.

Un análisis más en profundidad, sobre todo a la hora de responder a la pregunta de qué optimización se debería emplear para qué tipo de programa, se encuentra en el último capítulo. En este análisis, se considera el tiempo de ejecución sin optimización, así como se identifica aquella optimización más equilibrada.

OPTIMIZACIONES PARA ENTERO (VPR)

-fforce-mem

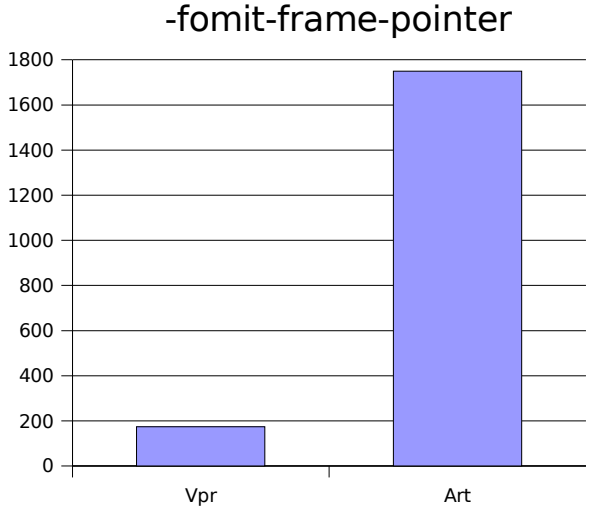
Fuerza a los operandos de memoria para que sean copiados en registros antes de realizar operaciones aritméticas sobre los mismos. Esto produce mejor código haciendo que todas las referencias a memoria se conviertan en potenciales subexpresiones comunes.



-fomit-frame-pointer

No mantiene el puntero de marco en un registro para funciones que no lo necesitan. Esto evita que las intrucciones guarden, generen y recuperen punteros de marco. Además, permite el uso de un registro adicional para muchas funciones.

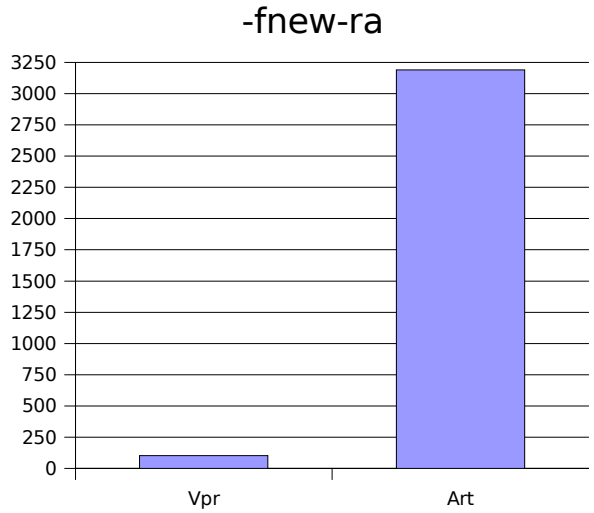
Desgraciadamente, esta opción de optimización hace que la depuración sea imposible en algunas máquinas.



-fnew-ra

Usa una nueva técnica denominada “graph coloring register allocator”.

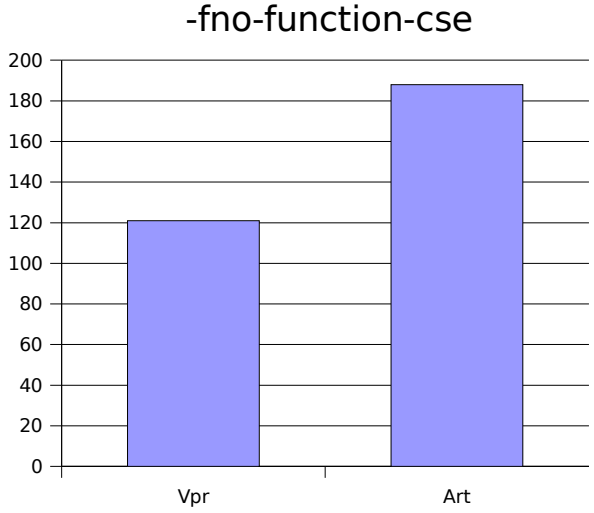
Se trata de una opción en desarrollo, con lo que GCC recomienda usarla solamente para probar, puesto que no está lista para entornos de producción.



-fno-function-cse

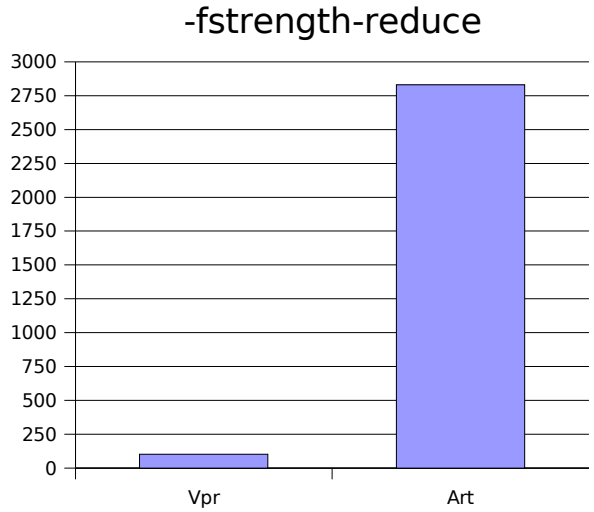
No coloca direcciones de funciones en los registros. Hace que cada instrucción que llame a una función constante, contenga la dirección de la función de forma explícita.

Esta opción hace que el código sea menos eficiente, pero algunos retoques extraños que alteran el código ensamblador resultante pueden ser confundidos por las optimizaciones realizadas por esta opción.



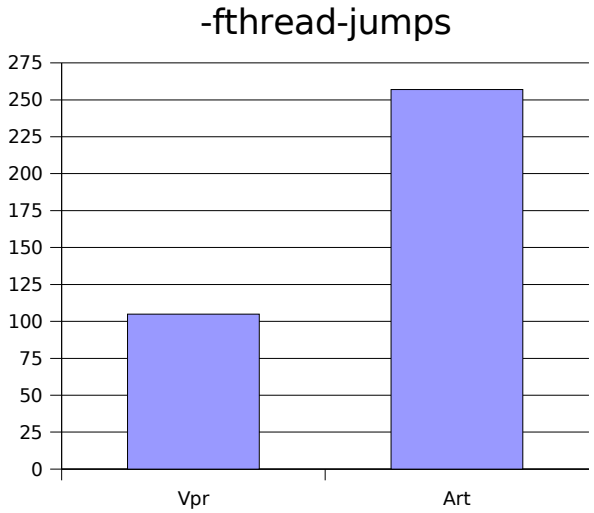
-fstrength-reduce

Realiza optimizaciones de reducción de “fuerza” de los bucles y eliminación de variables de iteración.



-fthread-jumps

Realiza optimizaciones en las que se comprueba si una condición con salto se dirige a una localización donde se haya otra comparación sumada a la primera. De ser así, la primera condición es redirigida, bien al destino de la segunda condición, o bien a un punto inmediatamente después, dependiendo de si el resultado de la condición es conocido (verdadero o falso).

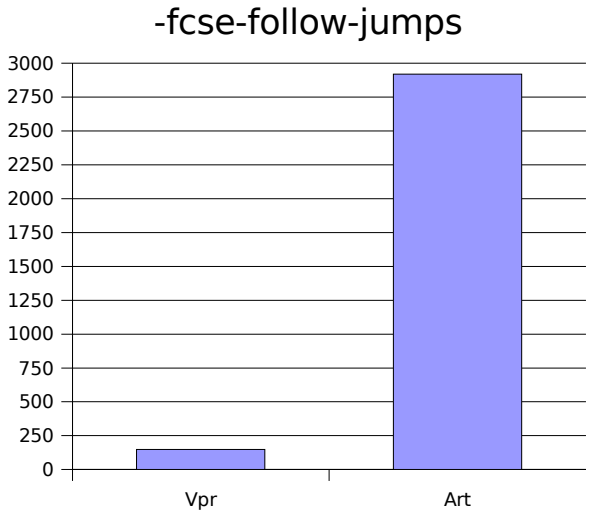


-fcse-follow-jumps

Durante la eliminación de subexpresiones comunes, escanea a través de instrucciones de salto mientras que el objetivo del mismo no es alcanzado por

otro camino.

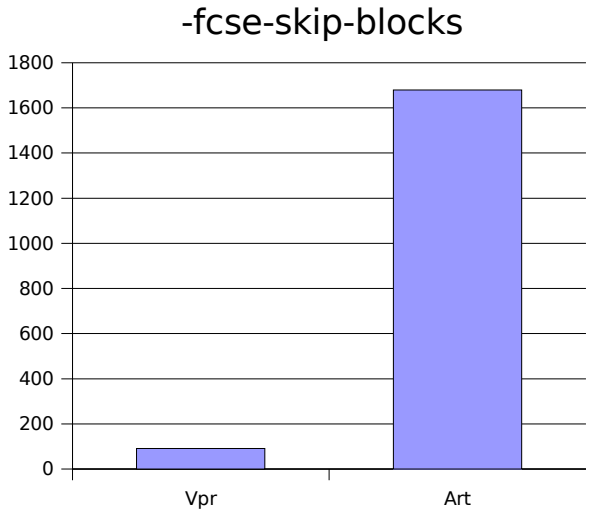
Por ejemplo, cuando se encuentra un IF con un ELSE, se seguirá el salto cuando la condición verificada sea falsa.



-fcse-skip-blocks

Es similar al anterior, con la variante de que se siguen los saltos que condicionalmente saltan sobre bloques.

Por ejemplo, cuando se encuentra un IF sin un ELSE, se seguirá el salto que evitará el contenido del IF.



-fgcse

Realiza una pasada en la que se eliminan subexpresiones globales comunes. Además, en esta pasada se realiza propagación de constantes globales y copias.

Esta optimización tiene activadas por defecto otras:

-fgcse-lm

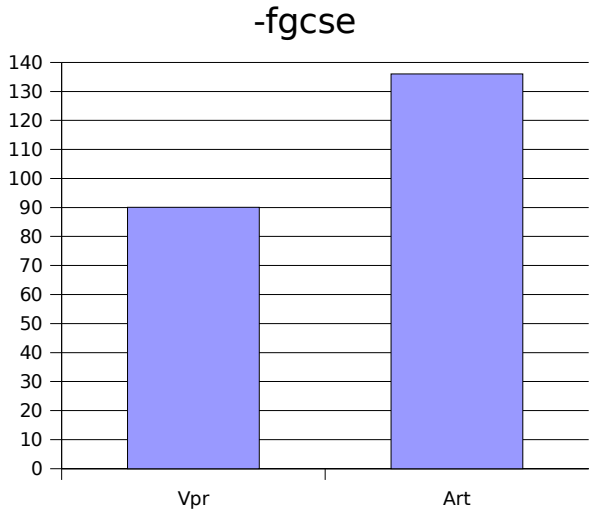
Se intentan mover los “loads” que pueden emparejarse con “stores”.

Esto permite que un bucle que contenga una secuencia “load/store”, cambie a un “load” fuera del bucle y a un “copy/store”, dentro.

-fgcse-sm

Se mueven todos los “stores” fuera de los bucles.

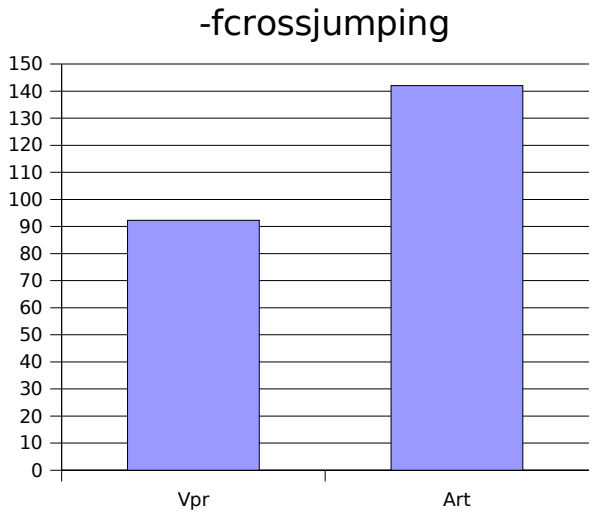
En los bucles que contienen una secuencia de “load/store”, dicha secuencia podrá cambiarse a un “load” antes del bucle y un “store”, después.



-fcrossjumping

Emplea la técnica de transformación de “Cross-Jumping”. Esta transformación unifica código equivalente y además, disminuye el tamaño del código.

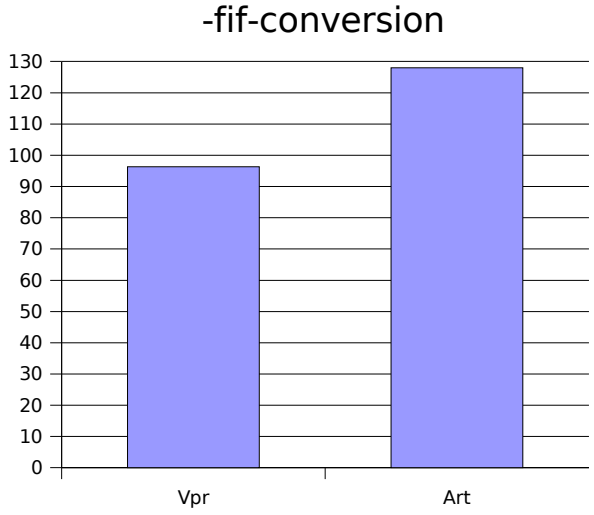
Aun así, GCC avisa de que el código resultante de emplear esta optimización puede no ser mejor que sin ella.



-fif-conversion

Intenta reducir el número de saltos condicionales.

Esta optimización incluye el uso de movimientos condicionales, además de algún que otro truco basado en aritmética básica.

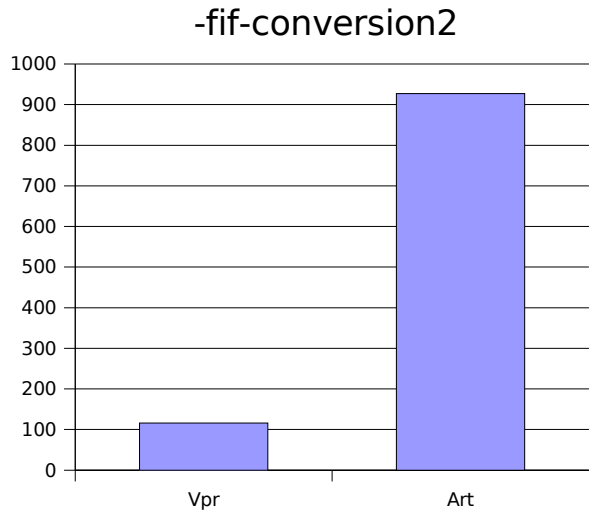


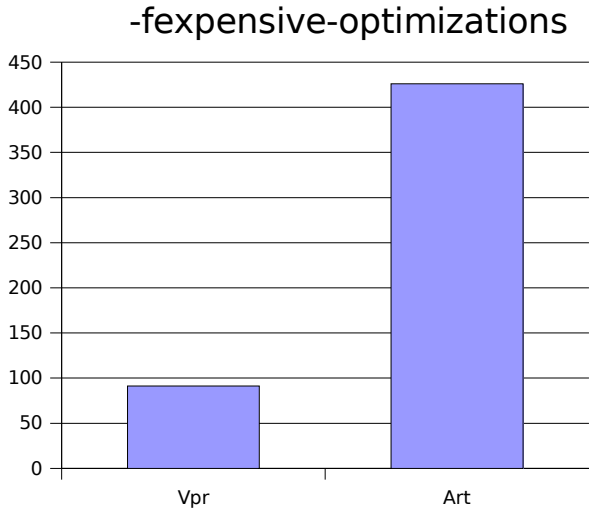
-fif-conversion2

Usa ejecución condicional para transformar saltos condicionales en instrucciones de salto equivalentes pero menos numerosas.

-fexpensive-optimizations

Realiza una serie de optimizaciones menores pero que en la práctica consumen mucho tiempo de compilación.





-fschedule-insns

Intenta reordenar las instrucciones para eliminar esperas en la ejecución debidas a que hay datos no disponibles.

Esto, en un principio, ayudaría a las máquinas que operan léntamente con punto flotante o con “loads”, ya que permite la ejecución de otras instrucciones mientras los resultados de las primeras no sean necesarios.

Esta opción de optimización tiene otras asociadas, que se activan por defecto:

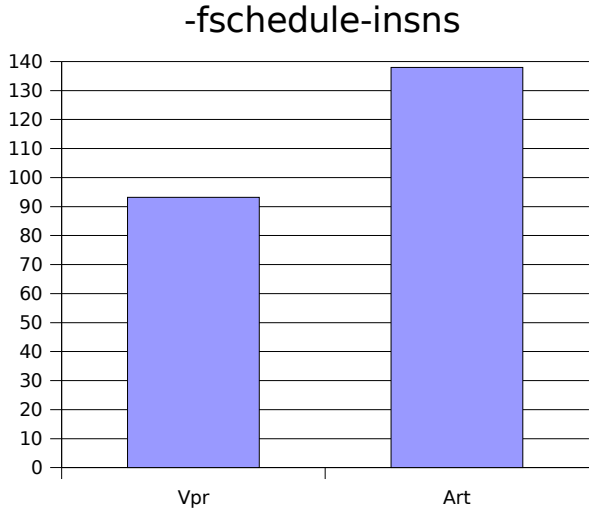
-fno-sched-interblock

No planifica las instrucciones que pertenezcan a bloques básicos.

-fno-sched-spec

No permite los cambios especulativos de instrucciones que no sean

“loads”.

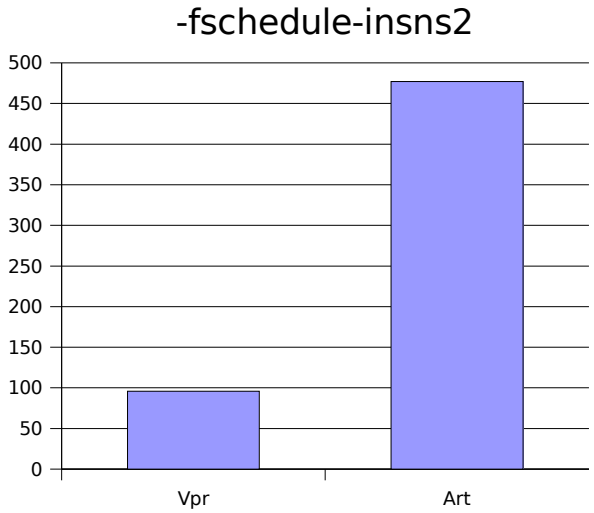


-fschedule-insns2

Es similar al anterior, pero requiere de una pasada adicional del planificador de instrucciones antes de que la reasignación de registros se dé por completada.

La idea es favorecer a las máquinas con pocos registros y donde los “loads” tarden más de un ciclo.

También aquí se activan las opciones de optimización antes mencionadas.

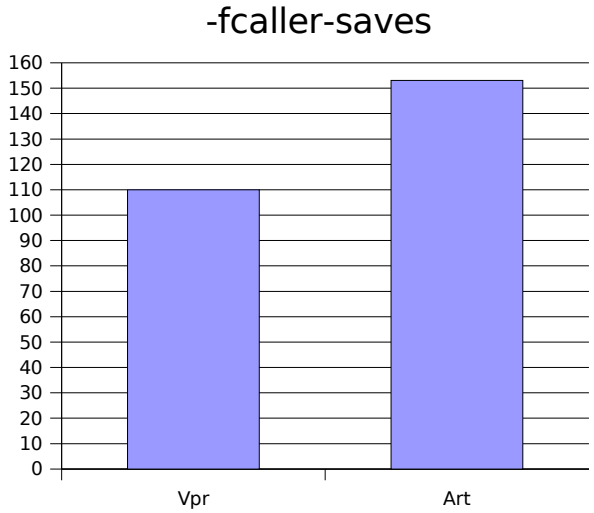


-fcaller-saves

Activa los valores para que sean asignados en registros que serán sobreescritos por llamadas a funciones.

Esto se realiza emitiendo instrucciones adicionales que guardan y recuperan los registros antes y después de estas llamadas.

Esta reasignación se realiza solamene cuando existe la posiblidad de obtener código.

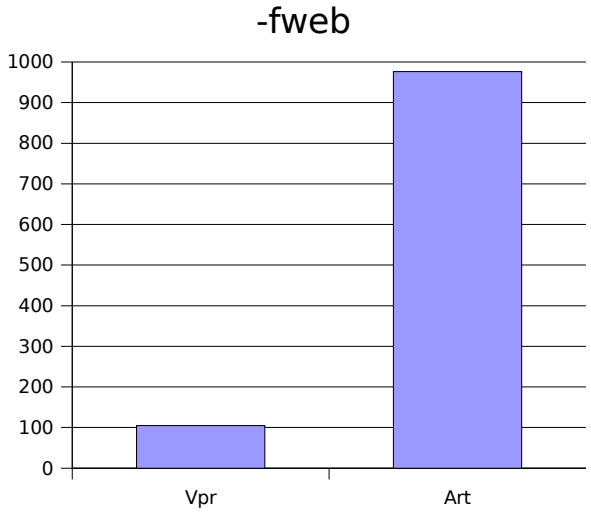


-fweb

Utiliza un algoritmo de “construcción de redes” normalmente usado para asignación de registros. La mecánica consiste en asignar cada “pseudo-registro” individual de la red.

Permite que la pasada en la que se asignan los registros, opere directamente con los “pseudo-registros”, y también refuerza muchas otras optimizaciones como CSE, mejora de bucles, ...

El inconveniente de esta optimización radica en la imposibilidad de depurar código, debido a que las variables ya no se encontrarán en un “registro base”.

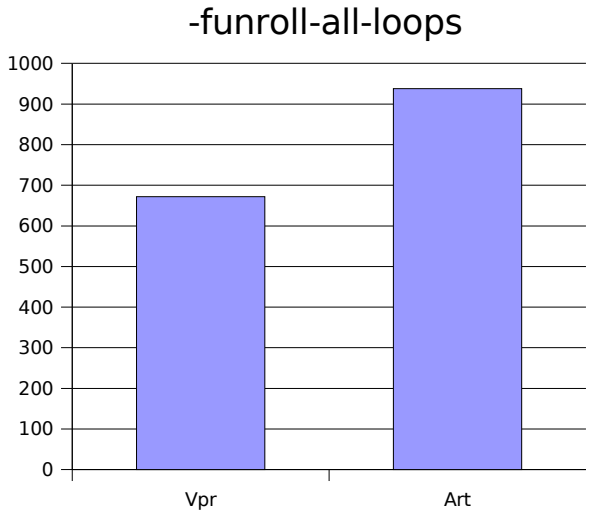


-funroll-all-loops

Desenvuelve todos los bucles, incluso cuando su número de iteraciones es incierto una vez que se entra en el mismo.

Por norma, esto hace que los programes se ejecuten más lentamente.

Esta opción de optimización implica otra más, **-frerun-cse-after-loop**. Es decir, lanza las optimizaciones CSE (Eliminación de Subexpresiones Comunes) al final.

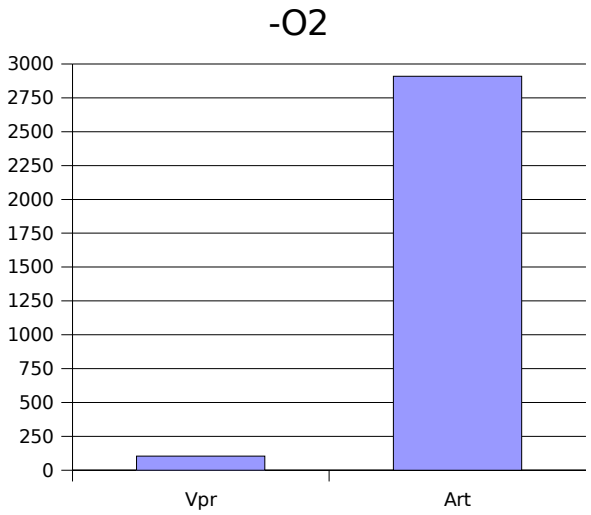


-O2

Se trata de una optimización comodín que incorpora a su vez otras optimizaciones bastante dispares, algunas de las cuales están reflejadas en este estudio.

Dichas optimizaciones son (en negrita están las tratadas en este estudio):

-fdefer-pop
-fmerge-constants
-fthread-jumps
-floop-optimize
-fif-conversion
-fif-conversion2
-fdelayed-branch
-fguess-branch-probability
-fcprop-registers
-fforce-mem
-foptimize-sibling-calls
-fstrengh-reduce
-fcse-follow-jumps
-fcse-skip-blocks
-frerun-cse-after-loop
-frerun-loop-opt
-fgcse
-fgcse-lm
-fgcse-sm
-fgcse-las
-fdelete-null-pointer-checks
-fexpensive-optimizations
-fregmove
-fschedule-insns
-fschedule-insns2
-fsched-interblock
-fsched-spec
-fcaller-saves
-fpeephole2
-freorder-blocks
-freorder-functions
-fstrict-aliasing
-funit-at-a-time
-falign-functions
-falign-loops
-falign-labels
-fcrossjumping

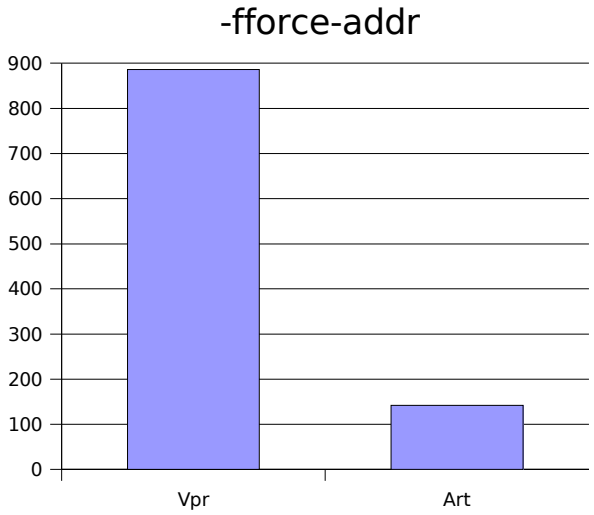

-O2
3000
2750
2500
2250
2000
1750
1500
1250
1000
750
500
250
0
Vpr
Art

OPTIMIZACIONES PARA PUNTO FLOTANTE (ART)

-fforce-addr

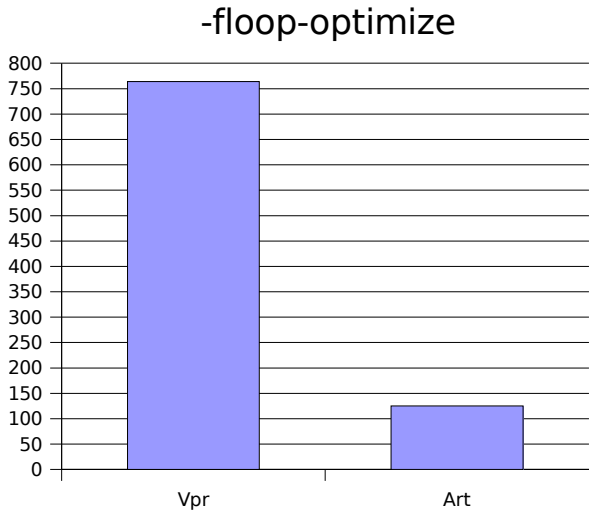
Fuerza la copia de las direcciones de memoria a registros antes de realizar operaciones artiméticas con ellas.

Esta optimización debería producir mejor código que su complementaria **-fforce-addr** (vista en los enteros).



-floop-optimize

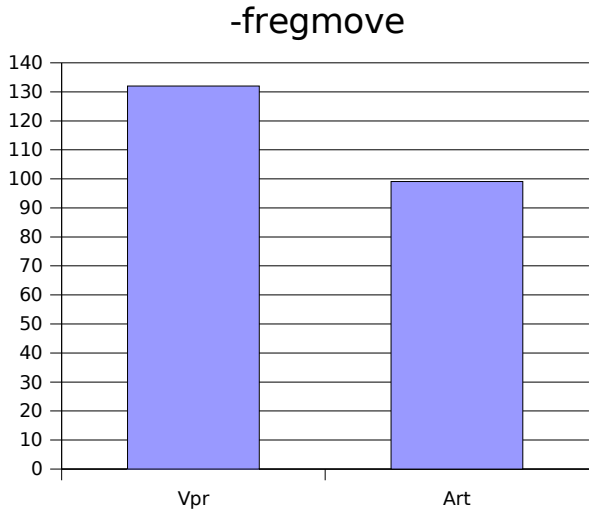
Realiza optimizaciones en los bucles como: mover todas las expresiones constantes fuera de los mismos, simplificar las condiciones de salida y opcionalmente, realizar desenrollado de bucles.



-fregmove

Intenta reasignar los números de registro en los “move” y en aquellas instrucciones en los que se manejan como operandos.

El objetivo es maximizar la permanencia de datos en cada registro.

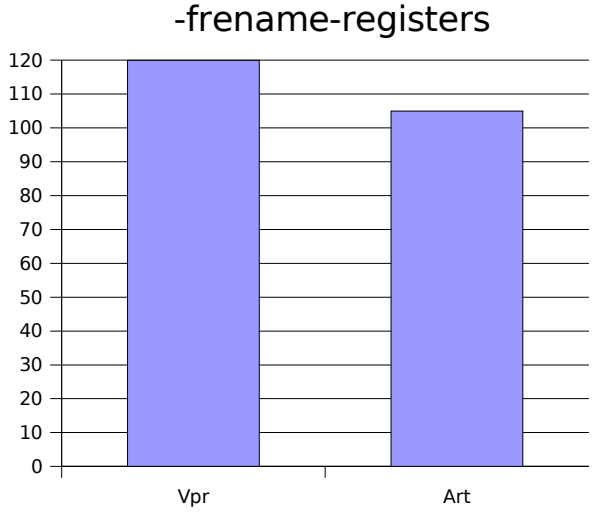


-frename-registers

Intenta evitar falsas dependencias en el código que ha sido planificado. Esto se consigue usando registros que han sido descartados tras cada asignación.

Teóricamente, esta optimización beneficiará a aquellos procesadores con muchos registros.

El inconveniente de esta optimización radica en la imposibilidad de depurar código, debido a que las variables ya no se encontrarán en un “registro base”.



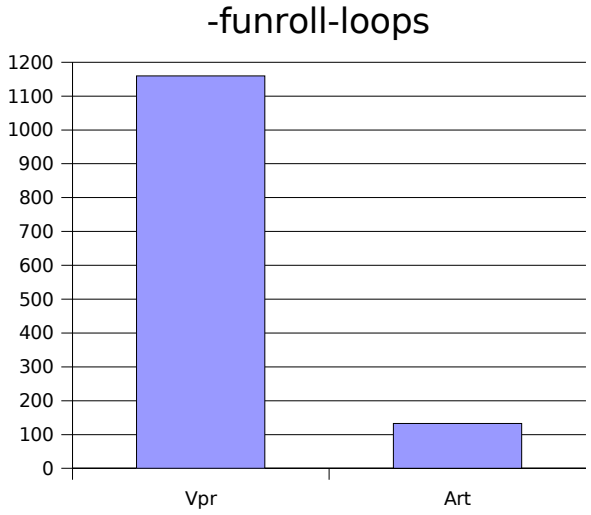
-funroll-loops

Desenrolla todos aquellos bucles cuyo número de iteraciones puede determinarse en tiempo de compilación o antes de entrar en el mismo.

Esta opción implica, al igual que en **-funroll-all-loops** (visto en la sección de entero), la optimización **-frerun-cse-after-loop**.

Además, activa el “pelado” de bucles que se verifica, por ejemplo, con la total eliminación de bucles con un número constante y pequeño de iteraciones.

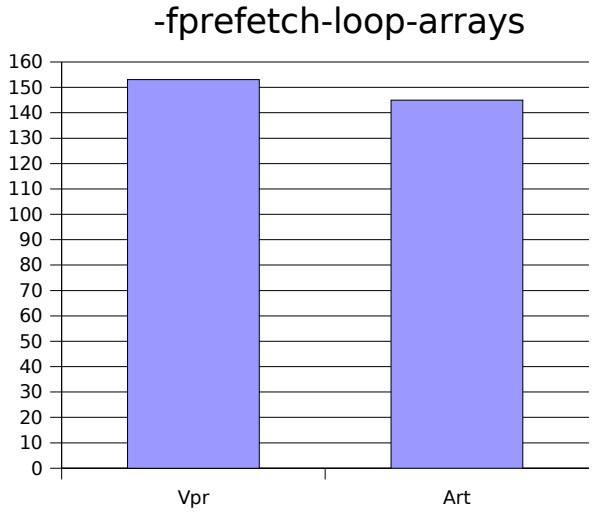
Esta opción hace que el código aumente de tamaño, pero el programa pudiera ejecutarse más rápidamente.



-fprefetch-loop-arrays

Genera instrucciones para realizar “prefetch” en la memoria, y así aumentar el rendimiento de los bucles que acceden a arrays grandes.

En la práctica, esta opción no funciona en todas las máquinas.



-O3

Se trata de una optimización comodín que incorpora a su vez otras optimizaciones bastante dispares, algunas de las cuales están reflejadas en este estudio.

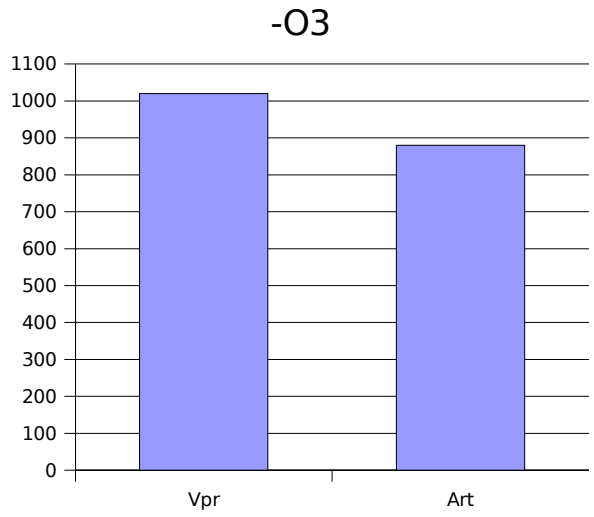
Activa las optimizaciones de **-O2**, que son (en negrita están las tratadas en este estudio):

-fdefer-pop
-fmerge-constants
-fthread-jumps
-floop-optimize
-fif-conversion
-fif-conversion2
-fdelayed-branch
-fguess-branch-probability
-fcprop-registers
-fforce-mem
-foptimize-sibling-calls
-fstrengh-reduce
-fcse-follow-jumps
-fcse-skip-blocks
-frerun-cse-after-loop
-frerun-loop-opt
-fgcse
-fgcse-lm
-fgcse-sm
-fgcse-las
-fdelete-null-pointer-checks
-fexpensive-optimizations
-fregmove
-fschedule-insns
-fschedule-insns2
-fsched-interblock

-fsched-spec
-fcaller-saves
-fpeephole2
-freorder-blocks
-freorder-functions
-fstrict-aliasing
-funit-at-a-time
-falign-functions
-falign-loops
-falign-labels
-fcrossjumping

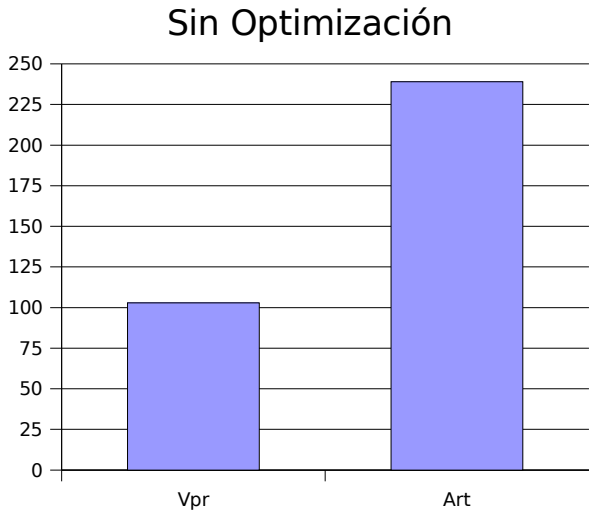
Y añade las siguientes (en negrita están las tratadas en este estudio):

-finline-functions
-fweb
-frename-registers

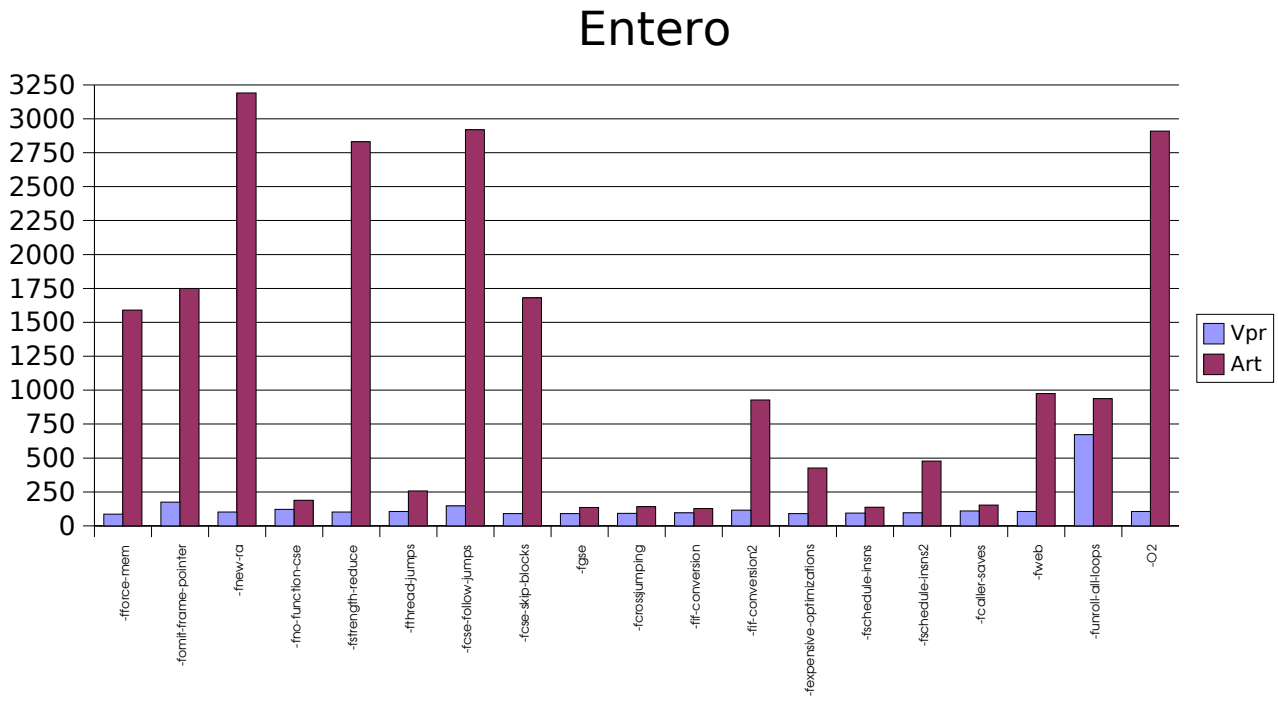


RESULTADOS Y CONCLUSIONES

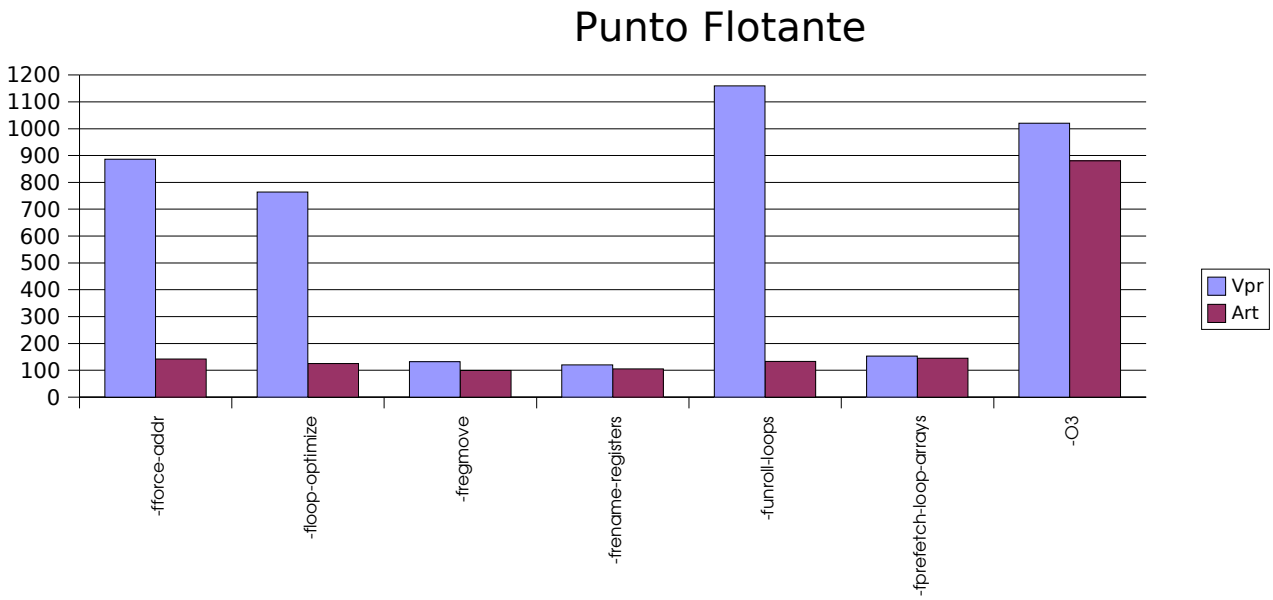
Para poder alcanzar alguna conclusión, necesitaremos tomar como base el resultado obtenido por la ejecución sin ninguna optimización:



A continuación se pueden apreciar los mejores resultados para el benchmark entero (vpr) en un mismo gráfico y a modo de resumen:



Y el mismo tipo de gráfico pero para los de punto flotante (art):



Podemos observar que no todas las optimizaciones actúan por igual en cada benchmark, esto es obvio. Pero lo que resulta llamativo es el hecho de que en la mayoría de los casos, el efecto de la optimización no está para nada equilibrado.

En muchos de los casos, nos damos cuenta que usar opciones de optimización no es del todo producente que se esperaba. La diferencia en los resultados para la misma opción llega a ser escandalosa, así como, en el mejor de los casos (siempre para dicha opción), el no usar optimizaciones se revela como lo mejor.

Aparte del gran desequilibrio en los resultados para la misma optimización, está el hecho de que, en el mejor de los casos, se arañan unos pocos segundos. En otros, se desaconsejaría el uso de esa opción en particular.

Para el caso del benchmark de entero (vpr), las mejores opciones (cuyo resultado es mejor que sin optimización alguna) y la mejora son:

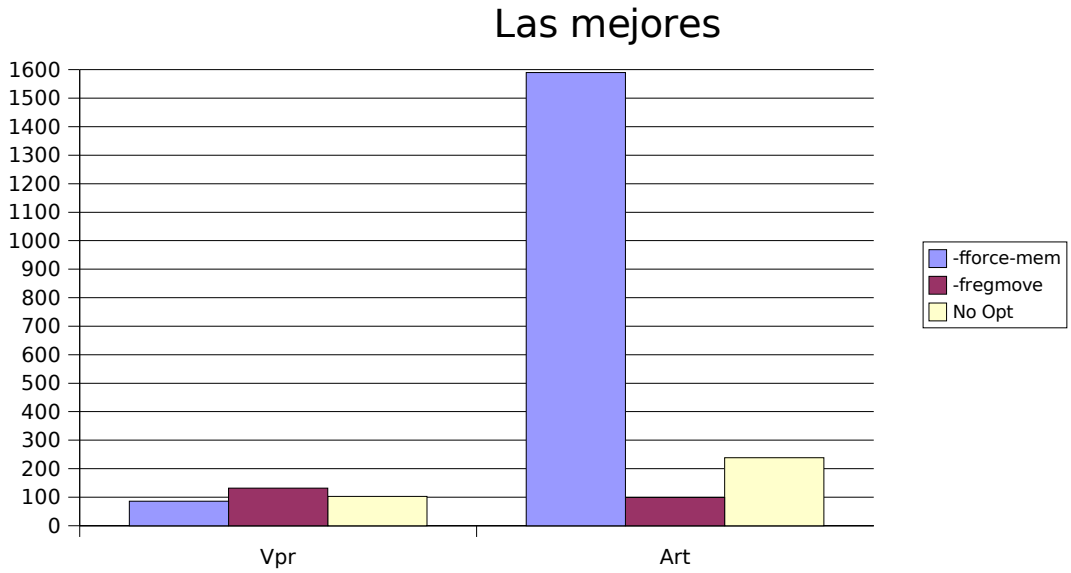
-fforce-mem (16,8%)
-fcse-skip-blocks (12.5%)
-fgse (12,9%)
-fcrossjumping (10.7%)
-fif-conversion (6,7%)
-fexpensive-optimizations (11,9%)
-fschedule-insns (9,8%)
-fschedule-insns2 (7,2%)

Y para el caso de los de punto flotante (art):

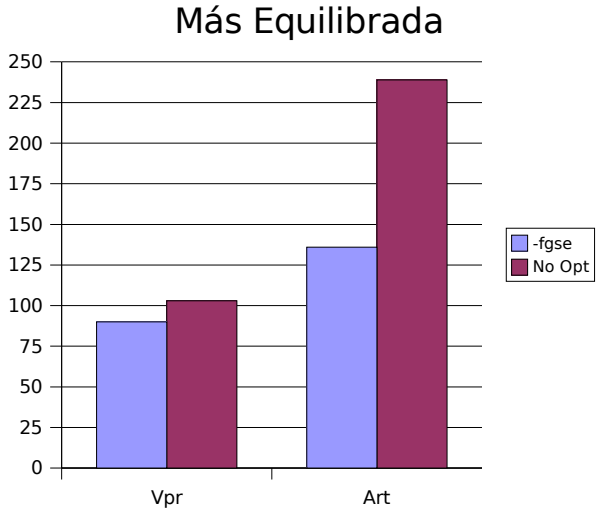
-fforce-addr (97%)
-fno-function-cse (51%)
-fgcse (103%)
-floop-optimize (114%)

-fcrossjumping (97%)
-fif-conversion (111%)
-fregmove (139,9%)
-fschedule-insns (101%)
-fcaller-saves (86%)
-frename-registers (139%)
-funroll-loops (106%)
-fprefetch-loop-arrays (94%)

De estas dos listas, podemos decidir qué optimización se debería usar para cada uno de los dos programas según el tipo de operaciones tratadas y estructura:



Cabe formularnos la pregunta de ¿cuál será la optimización más equilibrada? La respuesta es **-fgcse**, debido a que la mejora en el tiempo de ejecución es bastante alta en los dos benchmarks.



Este estudio, dentro del campo de visión escogido, desaconseja el uso de optimizaciones agrupadas, esto es, **-O2** y **-O3**. Mientras algunas de las opciones que contienen, mejoran claramente la ejecución, otras echan por tierra dichas mejoras.

¿Qué ventajas obtiene el programador con **-O2** y **-O3**? La única que podría ver es el hecho de poder probar suerte sin tener que conocer el funcionamiento de su programa y/o las optimizaciones. Si dejamos que GCC piense por nosotros y active las opciones más populares, quizás ganamos tiempo. Aún así, se ha visto en este estudio que, siempre considerando el entorno de trabajo empleado, **-O2** irá muy bien para programas que trabajan en entero y **-O3**, para aquellos de punto flotante.

En este estudio se ha omitido la opción **-Os**, cuyo beneficio exclusivo es el de la reducción del tamaño del programa. Esta opción activa aquellas optimizaciones de **-O2** que no implican un aumento de tamaño (en negrita está la opción usada en este estudio):

-falign-functions
-falign-jumps
-falign-loops
-falign-labels
-freorder-blocks
-frpefetch-loop-arrays

Para finalizar, se quisiera recalcar la importancia de conocer el medio en el que uno trabaja y las herramientas de las que se dispone. Dicho conocimiento es de vital importancia a la hora mejorar el rendimiento de lo programado y así, obtener mejores resultados en las siguientes fases de cualquier investigación.