gcc: компиляция на форсаже с турбо-наддувом

крис касперски ака мыщъх, aka nezumi, no-email

в оптимизации гораздо больше магии, чем науки. компилятор gcc поддерживает сотни ключей оптимизации, влияние большинства из которых на быстродействие программы весьма неоднозначно — в одном случае мы получаем колоссальный прирост производительности, в другом же — обвальное падение. причем, чем агрессивнее ведет себя оптимизатор, тем выше шансы, что откомпилированная программа развалится. поэтому разработчики, как правило, оставляют солидный "запас прочности" по оптимизации, позволяющий нам (не без риска, конечно) форсировать ключи компиляции, увеличивающие скорость работы программы в несколько раз.

введение

Наверное, всем известно, что компилятор gcc поддерживает несколько уровней оптимизации (O1 – минимальная оптимизация, O2 – умеренная оптимизация, O3 – агрессивная оптимизация), но далеко не всякий догадывается, что даже на уровне O3 задействуются далеко не все режимы оптимизации и до максимальной производительности еще пилять и пилять.

Однако, просто забить все ключи в командную строку — недостаточно. Если бы все было так просто, разработчики компилятора уже давно бы сделали это за нас. В погоне за производительностью очень легко "вылететь" с трассы и свалиться в глубокую яму. Подобрать нужную комбинацию ключей оптимизации с правильными параметрами с первой попытки получится навряд ли. Тут экспериментировать нужно! И желательно не вслепую, а осмысленно. То есть знать устройство процессора и принцип работы оптимизатора.

Разумеется, в рамках короткой журнальной статьи мыщъх не в силах рассказать обо всех ключах компилятора и потому вынужден остановится лишь на самых проблемных техниках оптимизации, не описанных ни в документации на gcc, ни в популярных faq. Заинтересованных в углублении своих знаний, мыщъх отсылает к книге "техника оптимизации: эффективное использование памяти" и серии статьей "техника оптимизации под Linux", электронные которых можно скачать с http://nezumi.org.ru/optimization.zip и http://nezumi.org.ru/optimization-pack.zip соответственно.

Ну а мы сейчас отправимся в орбитальный полет. Главным образом, мы будем говорить о самой последней версии gcc — 4.2.1 (описание которой можно найти на http://gcc.gnu.org/onlinedocs/gcc-4.2.1/gcc/), что же касается остальных версий, то… сверяйтесь с документацией!

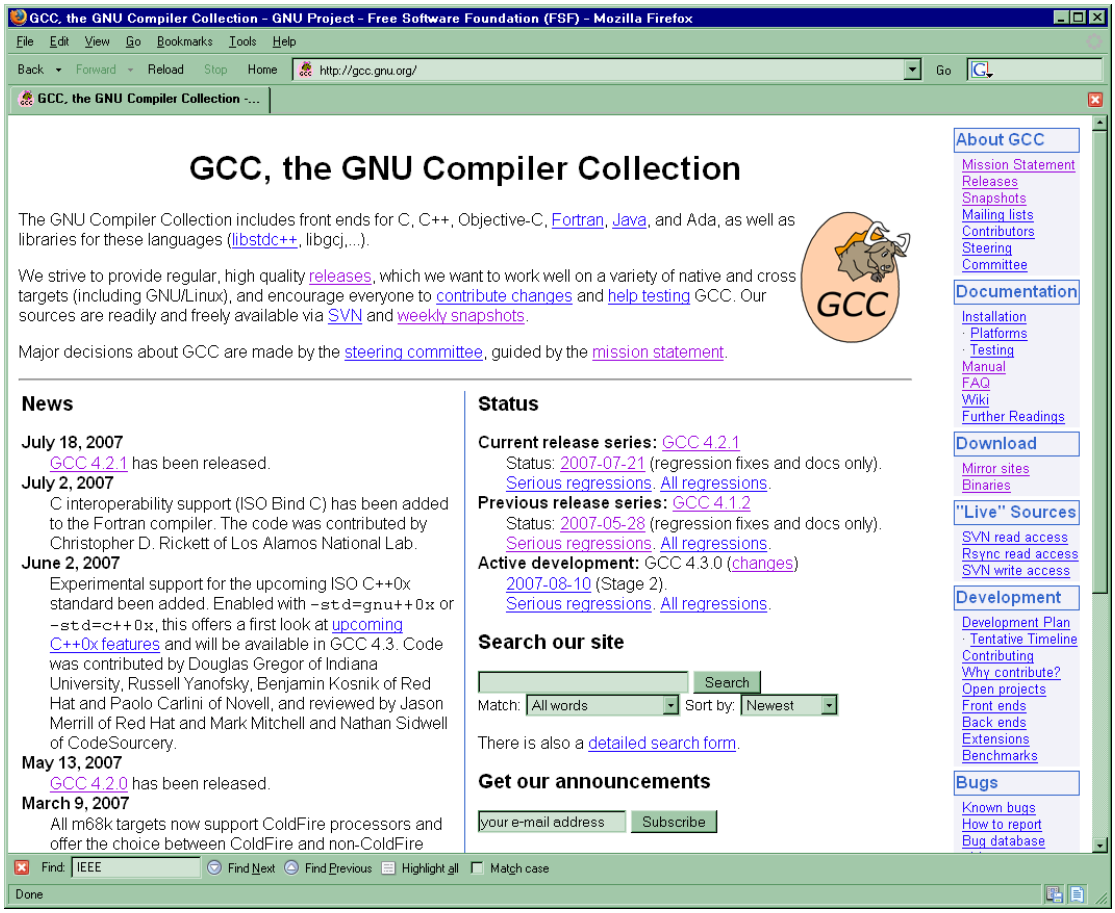

Рисунок 1 официальный сайт разработчиков компилятора gcc

разворот циклов

Процессоры семейства Intel Pentium и AMD Athlon построены по конвейерной архитектуре и в некотором смысле их можно уподобить гоночной машине, которая мощно несется по прямой дороге, но конкретно тормозит на виражах.

Циклы (особенно компактные) содержат небольшой участок линейного кода, за которым следует крутой поворот, тормозящий процессор и не дающий ему как следует разогнаться. Последние модели Penium-4, благодаря значительным архитектурам улучшениям, намного лучше справляются с циклами, чем процессоры предыдущих поколений, но потери в скорости все равно очень значительны и для повышения производительности необходимо расчистить трассу от ветвлений, что может сделать либо программист, либо компилятор.

Циклы, состоящие из нескольких команд `for(a=0;a<n;a++) *dst++= *src++;` исполняются _очень_ медленно и для повышения быстродействия оптимизаторы задействуют технику, именуемую разворотом циклов (***loops unrolling***), в процессе которой происходит многократное дублирование тела цикла, реализуемое приблизительно так:

```
for(i=1;i<n;i+)
        k += (n % i);
```

Листинг 1 цикл до разворота

```
// разворот цикла на 4 итерации
// выполняем первые n - (n %4) итераций
for(i=1;i<n;i+=4)
{
        k += (n % i) +\
        (n % i+1) + \
        (n % i+2) + \
        (n % i+3);
}
```

```
// выполняем оставшиеся итерации
for(i=4; i<n;i++) k += (n % i);
```

Листинг 2 цикл после разворота

Размер программы при этом, естественно, возрастает, а вместе с ним возрастает и риск вылететь за пределы кэша первого (и даже второго!) уровней, после чего производительность упадет так, что не поднимешь и домкратом.

Компилятор gcc не разворачивает циклы даже на уровне оптимизации -O3, и делает это только с ключом **`-funroll-loops`**, поддерживающим только цикл типа for. Ключ **`-funroll-all-loops`** (как и следует из его названия) поддерживает все виды циклов (например, do/while).

В обоих случаях, независимо от используемого ключа циклы с константным количеством итераций (т. е. известным на стадии компиляции), где const <= 32 разворачиваются полностью. При const > 32 — на ~4x (точное значение зависит от количества инструкций в теле цикла). Циклы с неизвестным количеством итераций не разворачиваются вообще! Хотя другие компиляторы (такие, например, как Intel C++) их преспокойно разворачивают.

Для тонкой настройки оптимизатора существуют ключи:

- **`max-unrolled-insns:`**
  - максимальное кол-во инструкций, при котором цикл может быть развернут; оптимальное значение зависит как от типа процессора, так и от конструктивных особенностей компилируемой программы, поэтому определять его приходится экспериментальным путем, мыщъх рекомендует начинать "плясать" от значения 69;
- **`max-average-unrolled-insns:`**
  - максимальное оценочное количество инструкций, которое цикл будет иметь после разворота; это число так же подбирается экспериментальным путем и мыщъх рекомендует "плясать" от значения 96;
- **`max-unroll-times:`**
  - максимальная степень разворота, по умолчанию равная 4x, это довольно разумное значение, но в некоторых случаях выбор 2x или 8x существенно увеличивает быстродействие программы;

Остальные ключи, связанные с разворотом циклов, описаны в документации на gcc. Они не имеют решающего значения и потому к ним прибегают только умудренные опытом гуру, да и то лишь время от времени.

выравнивание

Вплоть до появления Pentium Pro, процессоры крайне болезненно относитесь к не выровненным переходам и вызовам функций, чей адрес не был кратен 4 байтам, и давали за это штрафные такты (называемые "пенальти"), что объяснялось несовершенством микропроцессорной архитектуры тех лет.

Начиная с Pentium II+ и AMD K6+ процессоры уже не требуют постоянного выравнивания переходов/вызова функций, исключая тот случай, когда целевая инструкция или команда перехода пересекает границу линейки кэш-памяти первого уровня, "расщеплялась" напополам, за что выдается пенальти, причем, Pentium-4, компилирующий x86-инструкции в микро-код, выдает его значительно реже и совершенно непредсказуемым образом — микроинструкции абсолютно недокументированны, их длинна неизвестна, следовательно, управлять их выравниванием мы не можем.

Несмотря на то, что Pentium-4 де-факто является самым популярным процессором, и непоколебимым лидером рынка, большинство компиляторов упорно продолжают заниматься выравниванием, располагая инструкции перехода по кратным адресам и заполняя образующиеся "дыры" незначащими инструкциями, такими как NOP, MOV EAX, EAX и др. Естественно, это увеличивает размер кода, снижая его производительность, и агрессивное выравнивание только вредит.

Применительно к Pentium-II/Pentium-III и AMD K6+ машинная команда требует выравнивания в тех, и только тех случаях, когда следующее условие становится истинным (`(addr % cache_len + sizeof(ops)) > cache_len`) Здесь: addr – линейный адрес инструкции, cache_len размер кэш-линейки (в зависимости от типа процессора равный 32-, 64- или 128 байтам), ops – целевая машинная инструкция или инструкция перехода/вызова

функции. Количество выравнивающих байт рассчитывается по формуле: (`cache_len - (addr % cache_len)`).

Компилятор Intel C++ вообще не выравнивает ни переходов, ни циклов, что является лучшей стратегией для Pentium-4, а вот на более ранних процессорах мы получаем неустойчивый код с "плавающей" производительностью, быстродействие которого зависит от того, расщепляются ли глубоко вложенные переходы или нет. А это в свою очередь зависит от множества трудно прогнозируемых обстоятельств, включая фазу луны и количество осадков.

Компилятор gcc задействует выравнивание уже на уровне оптимизации O2, автоматически отключая его при задании ключа Os, причем осуществляет выравнивание по ужасно тупой схеме. Вышеприведенная формула остается незадействованной. Функции, циклы, метки (labels) и условные переходы выравниваются по фиксированной границе, управляемой следующими ключами: **`-falign-functions`**, **`-falign-loops`**, **`-falign-labels`** и **`-falign-jumps`** соответственно.

Каждый ключ принимает целочисленный аргумент, равный степени двойки и задающий кратность выравнивания, например, `-falign-functions=32` форсирует выравнивание функций по границе 32х байт. Значение "1" отключает выравнивание, а "0" задает выравнивание по умолчанию, специфичное для данного типа процессора.

Для всех типов процессоров: выравнивание переходов и меток лучше всего отключить, а функции выравнивать на величину от 4х до 32х байт (оптимальное значение подбирается экспериментально). Для не Penium-4 процессоров имеет смысл задействовать выравнивание циклов на величину кратную 4х байтам, однако, в некоторых случаях, отключение выравнивания позволяет увеличить производительность на 30%. Для Penium-4 все виды выравнивания лучше всего отключить, естественно, убедившись, что это не вызывает деградации производительности.

компиляция с обратной связью

Техника оптимизации на самом деле очень тесно связана с черной магией, астрологией и прочим колдовством. Ведь для генерации эффективного кода, компилятор вынужден заниматься спекулятивными предсказаниями, пытаясь опередить частоту срабатываний условных переходов, приблизительные значения аргументов, переданных функциями и т. д. и т. п. Естественно, техника предсказаний далека от совершенства и компилятор очень часто ошибается. В результате чего мы имеем "плавающую" производительность и прочие радости.

Настоящий прорыв произошел, когда разработчики догадались объединить профилировщик (инструмент для измерения производительности) с оптимизатором, в результате чего получился компилятор с обратной связью, которому уже не нужно гадать на кофейной гуще: стоить ли разворачивать цикл или нет — достаточно просто перебрать все возможные значения и, проанализировав информацию, возвращенную профилировщиком, выбрать оптимальную кратность разворота.

Компилятор gcc поддерживает компиляцию с обратной связью, но не использует ее даже на самых агрессивных уровнях оптимизации, хотя она уже давно вышла из экспериментальной стадии и готова к промышленному применению. Так почему же мы до сих пор вынуждены задействовать ее вручную?! Ответ прост: во-первых, использование профилировщика многократно увеличивает время компиляции, и сборка многих "серьезных" проектов растягивается более чем на стуки (сюрприз, да?). Во-вторых, информация, полученная по обратной связи, завязана на конкретную аппаратную конфигурацию и для других процессоров результат скорее всего окажется совершенно иным (то есть, бинарные сборки, откомпилированные подобным образом, лучше использовать только для себя и не распространять). Наконец, в-третьих, большинство программ львиную долю машинного времени тратят на ввод/вывод и достигают максимальной скорости своего выполнения уже на уровне O2, после чего прирост быстродействия можно обнаружить разве что хронометром.

Тем не менее, для экстремалов и любителей поэкспериментировать, компиляция с обратной связью открывает огромные возможности, которыми грех не воспользоваться.

Ключ **`-fprofile-use`** задействует обратную связь (profile feedback) и оказывает воздействие на следующие ключи оптимизации, которые обязательно должны быть указаны в командной строке (или заданы уровнем оптимизации от O1 до O3), иначе, вместо ожидаемого выхлопа мы получим "пшик":

- **`-funroll-loops`**:

    - разворот циклов будет выполняться на оптимальное количество итераций или не будет выполняться вообще, если для конкретно взятого цикла это невыгодно;
- **`-fpeel-loops`**:
    - "шелушение циклов" (разновидность разворота) будет выполняться в соответствии с показаниями профилировщика;
- **`-fvpt`**:
    - заставляет профилировщик запоминать значения переменных, собираемые по ходу выполнения программы, которые в дальнейшем могут быть использованы для более эффективной оптимизации;
- **`-fbranch-probabilities`**:
    - в комбинации с ключом `-fvpt` форсирует подсчет частоты "срабатывания" каждого условного перехода, записывая полученные данные в файл sourcename.gcda, после чего оптимизатор сможет реорганизовать код таким образом, чтобы сократить накладные расходы на ветвления и уменьшить трассу выполнения (***примечание***: *в текущих версиях gcc оптимизатор ограничивается лишь более эффективным распределением переменных по регистрам, но даже это дает существенный прирост производительности*);
- **`-ftracer:`**
    - форсирует хвостовую дубликацию (tail duplication) — довольно прогрессивный, и, кстати говоря, запатентованный метод оптимизации, подробнее о котором можно прочитать на http://www.freepatentsonline.com/20050183079.html (см. так же рис. 2).

Ключ **`-fprofile-generate`** задействует дополнительные возможности, заставляя профилировщик собирать еще больше данных, что позволяет использовать следующие ключи оптимизации:

- **`-fprofile-arcs`**:
    - собирает информацию о ходе выполнения программы, записывая ее в AUXNAME.da, и воздействует на следующие ключи оптимизатора, позволяющие генерировать более эффективный код: `-fno-guess-branch-probability`, `-fbranch-probabilities` и `-freorder-functions` — все эти ключи автоматически задействуются на уровнях оптимизации от O2 и выше, а потому нет никакой необходимости дописывать их вручную;
- **`-fprofile-values`**:
    - в комбинации с ключом `-fprofile-arcs` форсирует сбор значений переменных и выражений, позволяя отыскивать инварианты (т. е. значения, независимые от обрабатываемых программой данных) и оптимизировать процедуру вычисления многих вещественных выражений;

Текущие версии gcc только начинают осваивать компиляцию с обратной связью, делая в этом направлении свои первые шаги и, если эта идея не заглохнет, в обозримом будущем следует ожидать настоящего прорыва в область высоких скоростей и высокой компактности кода. Впрочем, не будем заговаривать наперед. Как говориться, поживем — увидим. А пользоваться компиляцией с обратной связью можно уже сейчас.

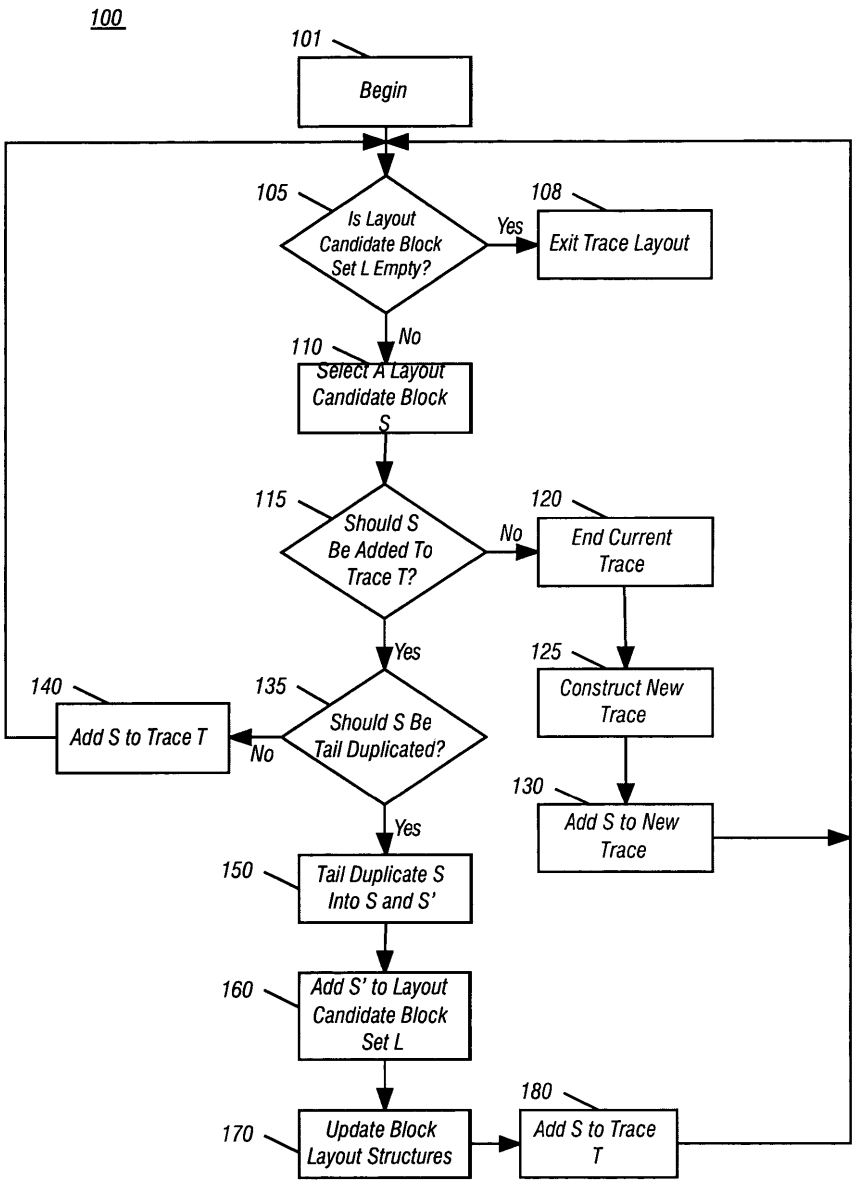

Рисунок 2 блок-схема, поясняющая суть хвостовой дубликации

быстрая вещественная математика

Вещественная математика (особенно двойной точности) до сих пор остается одним из узких мест, с которым не могут справится даже современные сопроцессоры с их улучшенной архитектурой. Компилятор gcc поддерживает ряд ускоренных методик вещественных вычислений, однако, не задействует их даже на уровне оптимизации O3, поскольку, они отклоняются от стандарта IEEE, а потому потенциально небезопасны и в некоторых случаях приводят к развалу программы.

С другой стороны, программы, интенсивно перемалевывающие вещественные числа, могут существенно повысить свою производительности и потому стоит попробовать задать ключ **`-ffast-math`**, активирующий следующие ключи (`-ffloat-store`, `-fno-math-errno`, `-funsafe-math-ptimizations`, `-fno-trapping-math`, `-ffinite-math-only`, `-fno-rounding-math`, `-fno-signaling-nans` и `fcx-limited-range`), после чего выполнить полный цикл тестирования программы и если что-то не ОК, то забить на `-ffast-math` и начать перебирать различные комбинации вышеупомянутых ключей по отдельности.

В некоторых случаях это дает двух-трехкратный прирост производительности.

заключение

Вот мы и рассмотрели наиболее значимые ключи оптимизации, получив широкий оперативный простор для экспериментов. Конечно, кто-то может резонно возразить: а стоит ли пыхтеть над какой-то программой несколько дней, чтобы увеличить ее производительность на 30%-60%?! Если измерять время деньгами, то дешевле купить более быстрый процессор, но Linux всегда привлекал большое количество энтузиастов, проводящих дни и ночи наполет в бессмысленных (для окружающих) ковыряниях системы. Так что, дерзайте! Наш ждут великие дела!

>>> врезка свободная таблица ключей оптимизации

Ниже приводится сводная таблица ключей оптимизации компилятора gcc 4.2.1. Несколько пояснений. Колонка "антиключ" содержит ключ, отменяющий данный. В колонках O1/O2/O3/Os значок "+" означает, что данный ключ задействуется на этом уровне оптмизации, а "-" – принудительно дезактивируется.

Колонка "действие" описывает влияние ключа на производительность: "+++" — значительное повышение, "++" — умеренное повышение, "+" слабое повышение, "X" – не влияет. Знаки "---", "--" и "-" означают тоже самое, только наоборот ("---" — существенное падение производительности").

Знак "?" указывает, что влияние данного ключа на производительности неоднозначно (то есть мы можем получить как ускорение, так и замедление), а "!" – указывает, что ключ может развалить программу.

Полужирным шрифтом выделены ключи с которыми рекомендуется поэкспериментировать.

ключ	антиключ	O1	O2	O3	Os	действие
-falign-functions=n	**-falign-functions=1**		**+**	**+**	**-**	**?**
-falign-jumps=n	**-falign-jumps=1**		**+**	**+**	**-**	**?**
-falign-labels=n	**-falign-labels=1**		**+**	**+**	**-**	**?**
-falign-loops=n	**-falign-loops=1**		**+**	**+**	**-**	**?**
-fbranch-count-reg	**-fno-branch-count-reg**					**-**
-fbranch-probabilities						**++**
-fbranch-target-load-optimize						**++**
-fbranch-target-load-optimize2						**++**
-fbtr-bb-exclusive						--
-fcaller-saves			+	+	+	++
-fcprop-registers		+	+	+	+	++
-fcrossjumping			+	+	+	?
-fcse-follow-jumps			+	+	+	++
-fcse-skip-blocks			+	+	+	++
-fcx-limited-range						**+++/!**
-fdata-sections						---
-fdelayed-branch		+	+	+	+	+
-fdelete-null-pointer-checks			+	+	+	X
-fearly-inlining		+	+	+	+	++
-fexpensive-optimizations			+	+	+	++
-ffast-math						**+++/!**
-ffinite-math-only						**+++/!**
-ffloat-store						---
-fforce-addr						++
-ffunction-cse	-fno-function-cse	+	+	+	+	++
-ffunction-sections						---
-fgcse			+	+	+	++
-fgcse-after-reload						++
-fgcse-las						**+++/!**
-fgcse-lm			+	+	+	++
-fgcse-sm						**+++/!**
-fif-conversion		+	+	+	+	++

-fif-conversion2		+	+	+	+	++
-finline-functions				+		**?**
-finline-functions-called-once		+	+	+	+	**?**
-finline-limit=n						**?**
-fivopts						++
-fmerge-all-constants						++
-fmerge-constants		+	+	+	+	++
-fmodulo-sched						?
-fmove-loop-invariants		+	+	+	+	+
-fmudflap						---
-fmudflapir						---
-fmudflapth						---
-fno-default-inline						**?**
-fno-defer-pop		-	-	-	-	---
-fno-guess-branch-probability		-	-	-	-	-
-fno-inline						**?**
-fno-math-errno						**++/!**
-fno-toplevel-reorder						---
-fno-trapping-math						**+++/!**
-fno-zero-initialized-in-bss						**+++/!!!**
-fomit-frame-pointer		+	+	+	+	+++
-foptimize-register-move			+	+	+	++
-foptimize-sibling-calls			+	+	+	++
-fpeel-loops						**+++**
-fprefetch-loop-arrays					**-**	**?**
-fprofile-generate						**+++**
-fprofile-use						**+++**
-fprofile-values						**+++**
-fregmove			+	+	+	++
-frename-registers						++
-freorder-blocks			+	+	-	++
-freorder-blocks-and-partition					**-**	**?**
-freorder-functions			+	+	+	++
-frerun-cse-after-loop			+	+	+	++
-frtl-abstract-sequences						**?**
-fsched2-use-traces						**?**
-fsched-spec-load						**?**
-fsched-spec-load-dangerous						**?**
-fsched-stalled-insns=n						**?**
-fsched-stalled-insns-dep=n						**?**
-fschedule-insns	-fno-sched-spec	+	+	+	+	++
-fschedule-insns2						?
-fsection-anchors						++
-fsee						++
-freschedule-modulo-scheduled-loops						**?**
-fsingle-precision-constant						**+++/!**
-fsplit-ivs-in-unroller		+	+	+	+	++
-fstack-protector						---
-fstack-protector-all						---
-fstrict-aliasing			+	+	+	++
-fstrict-overflow			+	+	+	++
-fthread-jumps			+	+	+	++
-ftracer						++
-ftree-ccp		+	+	+	+	++
-ftree-ch		+	+	+		++
-ftree-copy-prop		+	+	+	+	++
-ftree-copyrename		+	+	+	+	++

-ftree-dce		+	+	+	+	+
-ftree-dominator-opts		+	+	+	+	++
-ftree-dse		+	+	+	+	+
-ftree-fre		+	+	+	+	++
-ftree-loop-im						+++
-ftree-loop-ivcanon						?
-ftree-loop-linear						?
-ftree-loop-optimize		+	+	+	+	++
-ftree-lrs		+	+	+	+	++
-ftree-pre			+	+		++
-ftree-salias		+	+	+	+	++
-ftree-sink		+	+	+	+	++
-ftree-sra		+	+	+	+	++
-ftree-store-ccp			+	+	+	++
-ftree-store-copy-prop			+	+	+	++
-ftree-ter		+	+	+	+	++
-ftree-vect-loop-version		**+**	**+**	**+**	**-**	**?**
-ftree-vectorize						**?**
-funroll-all-loops						**?**
-funroll-loops						**?**
-funsafe-loop-optimizations						**+++/!**
-funsafe-math-optimizations	**-fno-unsafe-math-optimizations**					**+++/!**
-funswitch-loops						**?**
-fvariable-expansion-in-unroller						**?**
-fvpt						+++
-fwhole-program						+++