Анализ производительности оптимизированной функции

Замеры

Для начала хочется привести результаты окончательных измерений.

foo_0 – real time для исходной функции, в сек

foo_1 – для оптимизированной

foo_2 – для функции, компилированной с флагом -O3

Хочется отметить, что для тестирования использовался постоянный неизменяемый набор аргументов для функции foo для того, чтобы на конкретных данных измерить производительность.

	1	2	3	average
foo_0	25.203	25.392	25.390	25.328
foo_1	5.476	5.488	5.472	5.479
foo_2	5.084	5.084	5.098	5.089

Оптимизации

В таблицах измеренные времена для каждой оптимизации, которую я применял для исходного кода функции foo. Ограничимся измерением времени для улучшенного кода, это позволит подтвердить вывод о том, что та или иная оптимизация прокачала перф. Notice : в таблицах самая левая колонка будет обозначать номер оптимизации.

0.1 Замена постфиксного инкремента на префиксный

Префиксный эффективнее, так как не создает временного объекта, а возвращает ссылку на инкрементированное значение.

	1	2	3	average
1	25.212	25.242	25.227	25.227

0.2 Вынос проверки первых двух итераций цикла наружу

Данная оптимизация позволила уменьшить кол-во тактов процессора, которые тратились для проверки условия (ALU unit) на более полезную работу. Кроме того, как минимум 1 branch miss происходил в каждой итерации внешнего цикла в main, из-за этого branch. Сразу стоит сказать, что branch miss – очень дорогая ошибка, потому что процессор должен откатываться на прежнее состояние регистров до проверки условия, на что тратятся дополнительные такты.

	1	2	3	average
2	21.594	21.573	21.580	21.582

0.3 Оптимизация bar

В данной оптимизации мне пришлость отказаться от функции bar в одной из веток из-за того, что было очевидно, что при любом аргументе bar для этого branch она будет возвращать сам аргумент, поэтому тратить время на выделения стекового фрейма для этой бесполезной функции - не царское дело. Кроме того на проверку условия и branch miss мы не хотим тратить дополнительные такты процессора.

Кроме этого, для другого branch мне пришлось сделать bar встроенное функцией, т.е. приказал компилятору встроить тело функции на место ее вызова, для того чтобы не тратить время на выделение стекового фрейма.

	1	2	3	average
3	13.207	13.219	13.230	13.219

0.4 Введение новой локальной переменной для обозначения середины n / 2

Данная оптимизация позволила уменьшить некоторое количество тактов процессора, которое уходило на подсчет середины n —> т.е. ввёл локальную переменную mid = n / 2, в противном случае на каждой итерации приходилось вычислять данное значение – что неэффективно.

	1	2	3	average
4	11.350	11.365	11.369	11.361

0.5 Хитрая манипуляция с внутренним циклом

Данная оптимизация является поистинне интересной, так как удалось в 2 раза уменьшить количество итераций во внутреннем цикле в функции foo, благодаря обнаружению того факта, что два brancha в цикле можно считать параллельно, не нарушая никакие религии. Это позволило не только избавиться от некоторого количества branch misses (что является дорогим для перфа), но и в разы уменьшить количество тактов, которые тратились в самом цикле для увеличения счетчика, переходов, проверки условия, другой рутинной работы. Пришлось добавить небольшой кусок кода для прибавления двух забытых значений и обработки случая, когда аргумент n – нечетный, но большая часть данных инструкций выполняется параллельно на уровне инструкций (благодаря pipeline).

	1	2	3	average
5	7.654	7.640	7.633	7.642

0.6 Loop Unrolling

Стандартный прием для оптимизации цикла, который заключается в том, что на каждой итерации производим (В данном случае у меня сделано 4 шага) несколько шагов по выполнению похожих действий, причем записываем результаты каждого в независимые локальные переменные, которые помещаются в регистры процессора. Данный трюк позволяет в 4 раза уменьшить количество итераций во внутреннем цикле, что избавляет от траты тактов на рутинную работу и помогает максимально использовать параллелизм на уровне инструкций, благодаря независимым вычислениям каждого из 4 значений (pipeline)

	1	2	3	average
6	5.476	5.488	5.472	5.479

Выводы

Как можно заметить из анализа с уменьшением количества неоптимального использования тактов процессора на рутинную работу по проверке условий (branches), работе в итерациях цикла, выделения памяти для стековых фреймов, чрезмерно большого количества вычисления локальных переменных ведет к уменьшению работы алгоритма, а следовательно к улучшению перфа. Есть еще интересный факт по оптимизации производитлеьности процесса – это уменьшение количеств cpu-migrations. Можно указать тот логический процессор, на котором будет выполняться данный процесс, следоватьельно уменьшить количество тактов, которые тратятся на переключение контекстов планировщиком, но в уже оптимизирванном алгоритме, который выполняется не так долго, по сравнению с исходным, мы не сможем увидеть большой разницы в перфе, однако на первоначальных этапах оптимизации функции, используя данный метод, у меня получилось ускорить функцию на 2 секунды.