C++ Russia

elbrus

Что компилятор может сделать с вашей памятью

Виктор Шампаров
МЦСТ

viktor.shamparov@yandex.ru

vshamparov

C++ Russia

elbrus

Био

- Разработчик в МЦСТ
  - Универсальные оптимизации в составе компилятора
    - В основном оптимизации расположения данных
  - Некоторые профилировщики

C++ Russia

elbrus

Структура

- Вспомним кэши
- Оптимизации порядка доступа к памяти
- Оптимизации расположения данных
- Выводы

As-if rule

Основное требование к компиляторам C++ — сохранить *наблюдаемое поведение* (гл. 4.1.2 C++23):

- Volatile
- Запись в файлы к моменту завершения программы
- Динамика ввода-вывода *interactive devices*

Note: согласно ремарке к гл. 4.1.2 компилятор может нарушать иные требования стандарта при условии сохранения *наблюдаемого поведения*

C++ Russia

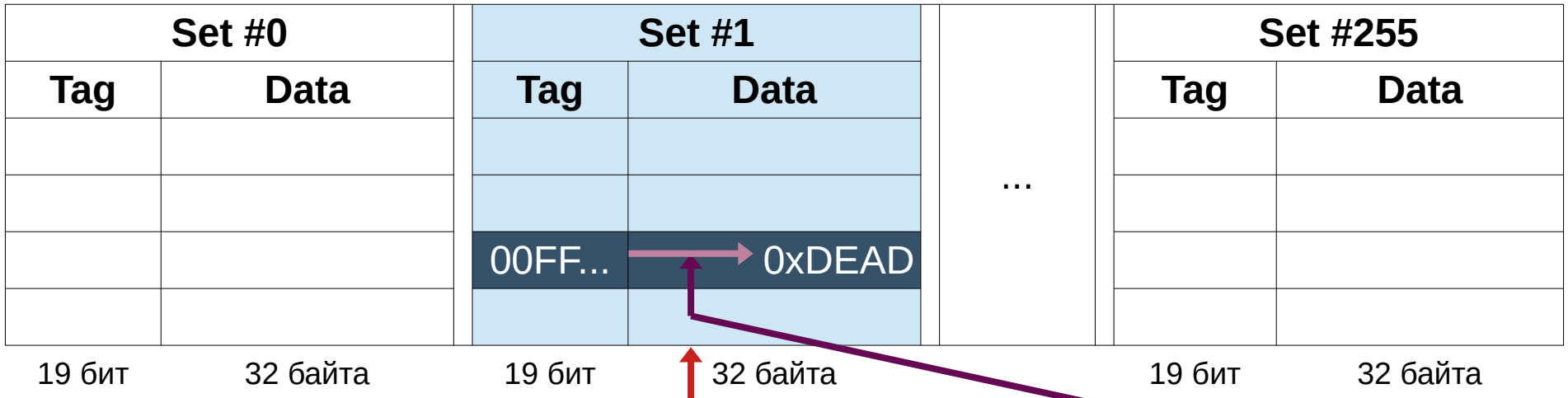
Все помнят иерархию памяти?

elbrus

Уровень	Время доступа в тактах	Примечание
Регистры	< 1	
$L1	3-5	Обычно индивидуальные кэши для каждого ядра
$L2	порядка 10	
$L3	порядка 30	Обычно общий кэш
RAM	порядка 70-100	
ROM	очень долго	

Кэш-память

Пример организации кэш-памяти: 32 Кб, 4 кэш-линии в сете, кэш-линии по 32 байта



Пример разбиения адреса для определения номера сета

Адрес	Tag	Set index	Offset
32 бита	19 бит	8 бит	5 бит
0000 1111 1010 0101 1000 0000 0011 1010	0000 1111 1010 0101 100	0000 0001	1 1010

Кэш-промах

Пример организации кэш-памяти: 32 Кб, 4 кэш-линии в сете, кэш-линии по 32 байта

Set #0		Set #1			Set #255	
Tag	**Data**	**Tag**	**Data**		**Tag**	**Data**
		0x1A...		...		
		0x2F...				
		0x05...				
		0x18...				
19 бит	32 байта	19 бит	32 байта		19 бит	32 байта

В сете нет совпадающего тега!

Адрес	Tag	Set index	Offset
32 бита	19 бит	8 бит	5 бит
0000 1111 1010 0101 1000 0000 0011 1010	0000 1111 1010 0101 100	0000 0001	1 1010

Кэш-промах

Пример организации кэш-памяти: 32 Кб, 4 кэш-линии в сете, кэш-линии по 32 байта

Set #0		Set #1		...	Set #255	
Tag	**Data**	**Tag**	**Data**		**Tag**	**Data**
		0x1A...				
		0x2F...				
		0x05...	Самая старая			
		0x18...				
19 бит	32 байта	19 бит	32 байта		19 бит	32 байта

Вытесняем линию согласно политике (например, LRU)

Адрес	Tag	Set index	Offset
32 бита	19 бит	8 бит	5 бит
0000 1111 1010 0101 1000 0000 0011 1010	0000 1111 1010 0101 100	0000 0001	1 1010

Кэш-промах

Пример организации кэш-памяти: 32 Кб, 4 кэш-линии в сете, кэш-линии по 32 байта

Set #0		Set #1		...	Set #255	
Tag	**Data**	**Tag**	**Data**		**Tag**	**Data**
		0x1A...				
		0x2F...				
		00FF...	Данные RAM			
		0x18...				
19 бит	32 байта	19 бит	32 байта		19 бит	32 байта

Вытесняем линию согласно политике (например, LRU)

Адрес	Tag	Set index	Offset
32 бита	19 бит	8 бит	5 бит
0000 1111 1010 0101 1000 0000 0011 1010	0000 1111 1010 0101 100	0000 0001	1 1010

Классификация кэш-промахов

	Когда	Пояснение
Compulsory	**Первичная** загрузка данных в кэш	Для сокращения требуется уменьшать *количество используемой памяти*
Capacity	Линии *вытеснены* из кэша **из-за ограниченности полного размера** и вновь подгружаются	Для сокращения требуется *улучшать локальность программы*
Conflict	Линии *вытеснены* из кэша **из-за конкуренции за некоторые сеты** и вновь подгружаются	Для сокращения требуется *убрать конкуренцию за конкретные сеты*

А теперь оптимизации! Prefetching

- Расстановка инструкций **предподкачки** данных
- *Промахи* остаются, но **маскируются** полезной деятельностью

```
for (i = 0; i < n; i++) {
  a[i] = a[i] + b[i];
}
```

```
for (i = 0; i < n; i++) {
  a[i] = a[i] + b[i];
  __builtin_prefetch(&a[i+P],
                     1, 1);
  __builtin_prefetch(&b[i+P],
                     0, 1);
}
```

Пример **Prefetching** на длину *P* для элементов массива в цикле

GCC: +	**Clang:** +	**LCC:** +

Prefetching

- Перед циклом *требует «разгона»* — времени на подкачку данных с первых итераций
  - Поэтому работает для большого числа итераций
- *Требует аппаратной поддержки* — инструкций prefetch
- *Требует предвычислить адрес* данных
  - Не работает со списочными структурами данных
- Зачастую это делает сам процессор — *устройствами предподкачки данных* в кэше
  - Иногда может работать и со списочными структурами данных

GCC: +	**Clang:** +	**LCC:** +

Loop Interchange

- Перестановка циклов в **гнёздах**
- Может улучшить локальность

```
for (j = 0; j < n; j++) {
  for (i = 0; i < n; i++) {
    s += A[i][j] * B[i][j];
  }
}
```

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    s += A[i][j] * B[i][j];
  }
}
```

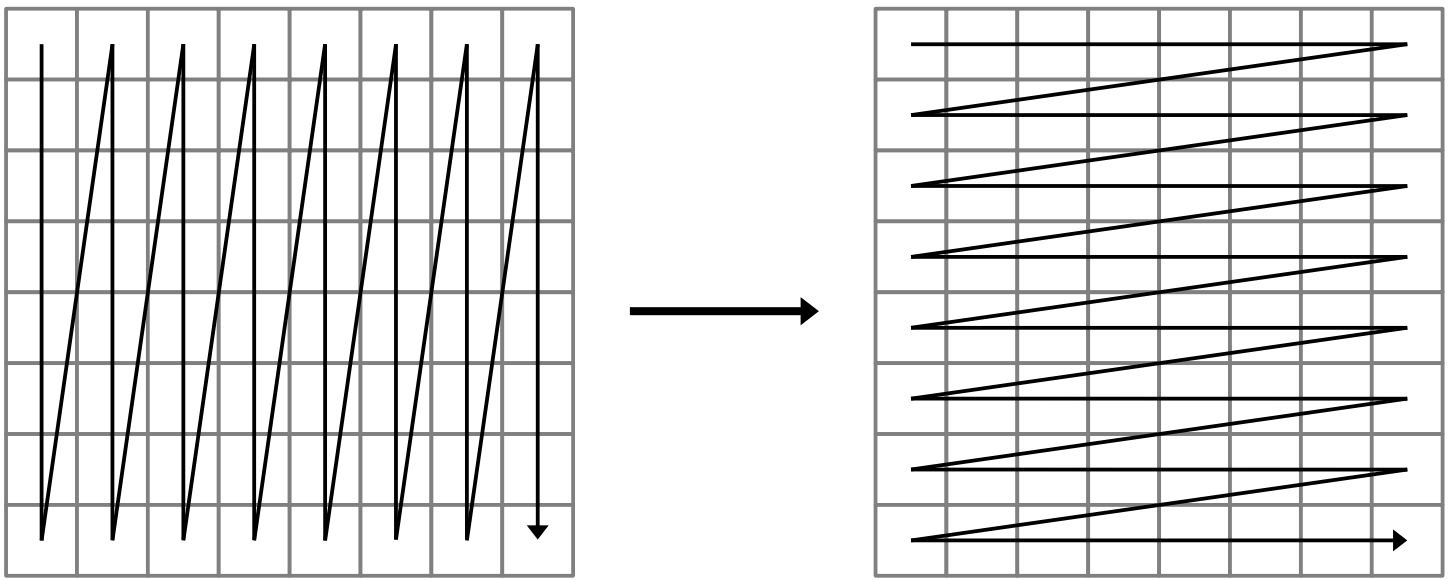
Пример **Loop Interchange** в гнезде из 2 циклов (обходе матрицы)

GCC: +	**Clang:** -floop-interchange	**LCC:** +

Loop Interchange

Было:
~1 **промах** на итерацию
(для очень больших матриц)

Стало:
~ **sizeof**(element) / **sizeof**(cacheline)
промахов на итерацию



Пример направления обхода матрицы A до и после **Loop Interchange**

GCC: `+`	**Clang:** `-floop-interchange`	**LCC:** `+`

Loop Interchange

- Легальна, если *вектора зависимостей* сохраняют лексикографическую положительность
- Но плохо помогает, если в гнезде обходы данных в разных направлениях

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    C[i][j] = 0;
    for (k = 0; k < n; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Пример гнезда циклов (умножения матриц) с разным направлением обхода данных

GCC: +	**Clang:** -floop-interchange	**LCC:** +

Loop Fusion

Слияние циклов с одинаковым пространством итераций

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
}
for (i = 0; i < n; i++) {
  s += C[i]/D[i] - E[i]/F[i];
}
```

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Пример **Loop Fusion** двух циклов

GCC: +	**Clang:** +	**LCC:** +

Loop Fusion

- Легальна при условии отсутствия таких *зависимостей*, при которых данные с ранних итераций второго цикла зависят от поздних итераций первого цикла
- Хотя предназначена не для оптимизации работы с кэшем, изменяет порядок обращений в память и может улучшить или ухудшить **локальность**

GCC: +	**Clang:** +	**LCC:** +

Внимательно смотрим

Допустим:

- Массивы A, B, C, D, E, F имеют тип **`uint32_t[2048]`** => размер *8 Кб*
- Массивы расположены в памяти подряд друг за другом
- Кэш: размер 32 Кб, 4 линии в сете, 32 байта в линии

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Всё хорошо?

На первый взгляд:

- Ожидаем в среднем 6/8 **промаха** на итерацию, так как в линию помещается 8 элементов массива
- Есть переиспользование элементов массивов C и D => можно сэкономить на лишних load-store

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 1 итерации:

До начала в кэше нет нужных данных

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
unknown	unknown
unknown	unknown
unknown	unknown
unknown	unknown

Пример работы сета кэша

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 1 итерации:

Доступ к A[0]: *compulsory* **промах**

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
A[0-7]	0
unknown	unknown
unknown	unknown
unknown	unknown

Пример работы сета кэша

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 1 итерации:

Доступ к B[0]: *compulsory* **промах**

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
A[0-7]	1
B[0-7]	0
unknown	unknown
unknown	unknown

Пример работы сета кэша

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 1 итерации:

Доступ к C[0]: *compulsory* **промах**

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
A[0-7]	2
B[0-7]	1
C[0-7]	0
unknown	unknown

Пример работы сета кэша

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 1 итерации:

Доступ к D[0]: *compulsory* **промах**

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
A[0-7]	3
B[0-7]	2
C[0-7]	1
D[0-7]	0

Пример работы сета кэша

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 1 итерации:

Доступ к C[0]: попадание

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
A[0-7]	4
B[0-7]	3
C[0-7]	0
D[0-7]	1

Пример работы сета кэша

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 1 итерации:

Доступ к D[0]: попадание

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
A[0-7]	5
B[0-7]	4
C[0-7]	1
D[0-7]	0

Пример работы сета кэша

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 1 итерации:

Доступ к E[0]: *compulsory* **промах**

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
E[0-7]	0
B[0-7]	5
C[0-7]	2
D[0-7]	1

Пример работы сета кэша

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 1 итерации:

Доступ к F[0]: *compulsory* **промах**

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
E[0-7]	1
F[0-7]	0
C[0-7]	3
D[0-7]	2

Пример работы сета кэша

Конфликты

Заметим, что здесь **все 6** элементов массивов в итерации конкурируют за один сет

Представим себе работу сета кэша на 2 итерации:

Доступ к A[1]: *conflict* **промах**

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
E[0-7]	2
F[0-7]	1
A[0-7]	0
D[0-7]	3

Пример работы сета кэша

Конфликты

И так далее…

Получаем **~6 *conflict* промахов** на итерацию!

Представим себе работу сета кэша на 2 итерации:

Доступ к B[1]: *conflict* **промах**

```
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Тот же пример после **Loop Fusion**

Данные	Возраст
E[0-7]	3
F[0-7]	2
A[0-7]	1
B[0-7]	0

Пример работы сета кэша

Array Padding

Вставка небольших промежутков между массивами

```
uint32_t A[2048], B[2048];
uint32_t C[2048], D[2048];
uint32_t E[2048], F[2048];

for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

```
uint32_t A[2048], _sh0[8];
uint32_t B[2048], _sh1[8];
uint32_t C[2048], _sh2[8];
uint32_t D[2048], _sh3[8];
uint32_t E[2048], _sh4[8];
uint32_t F[2048];
for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

Пример **Array Padding** для цикла с конфликтами

GCC:	Clang:	LCC: +

Array Padding

- Конфликты возникают в специфических обстоятельствах
  - Размеры находящихся рядом массивов кратны размеру кэша, поделённому на кол-во линий в сете
  - Массивов достаточно, чтобы вытеснить друг друга из сета
- Вариации:
  - **Inter-array Padding** для массивов на стеке — как в примере
  - **Inter-array Padding** для массивов в структуре
  - **Intra-array Padding** — увеличение одной из размерностей многомерного массива

GCC:	Clang:	LCC: +

Array Transpose

- Перестановка размерностей в многомерных массивах
- Перестановка размерностей в обращениях к этим массивам

```
uint32_t A[n2][n1];
uint32_t B[n2][n1];

for (j = 0; j < n1; j++) {
  for (i = 0; i < n2; i++) {
    s += A[i][j] * B[i][j];
  }
}
```

```
uint32_t A[n1][n2];
uint32_t B[n1][n2];

for (j = 0; j < n1; j++) {
  for (i = 0; i < n2; i++) {
    s += A[j][i] * B[j][i];
  }
}
```

Пример **Array Transpose** для оптимальной работы с памятью в гнезде из 2 циклов (обходе матриц)

GCC:	Clang:	LCC: `-fdata-reorg`

Array Transpose

- Цель оптимизации — улучшить локальность
- Но если многомерный массив используется в многих циклах — для некоторых локальность может ухудшиться
  - То есть при оптимизации надо оценить все циклы, где массив используется, и выбрать оптимальный именно для этого множества циклов порядок размерностей
  - А где получилось неоптимально, можно заполировать с помощью **Loop Interchange**
- Требует изменить **все** обращения
  - То есть нужен **анализ указателей**, чтобы выяснить, не утекают ли указатели

GCC:	Clang:	LCC: `-fdata-reorg`

IPO Memopt в LCC

Межпроцедурный вариант **Array Transpose** со сведением к одномерному массиву

```
uint32_t **A; // [n2][n1]
A = calloc( n2, … );
…
  A[j] = calloc( n1, … );
…
for (j = 0; j < n1; j++) {
  for (i = 0; i < n2; i++) {
    s += A[i][j];
}}
```

```
uint32_t *A; // [n2][n1]
A = calloc( n2*n1, … );
…
for (j = 0; j < n1; j++) {
  for (i = 0; i < n2; i++) {
    s += A[ j*n2 + i ];
}}
```

Пример **IPO Memopt** для оптимальной работы с памятью в гнезде из 2 циклов (обходе матриц)

GCC:	Clang:	LCC: -fdata-reorg

C++ Russia

elbrus

Что смущает в структуре?

```
struct A_t
{
    uint8_t id;
    uint64_t hash;
    uint16_t mask;
};
```

Field Reordering

Перестановка полей в структуре

```
struct A_t
{
    uint8_t id;
    // padding 7 bytes
    uint64_t hash;
    uint16_t mask;
    // padding 6 bytes
}; // sizeof(A_t) = 24
```

```
struct A_t
{
    uint64_t hash;
    uint16_t mask;
    uint8_t  id;
    // padding 5 bytes
}; // sizeof(A_t) = 16
```

Пример **Field Reordering**

GCC:	Clang:	LCC:	Не релиз

Field Reordering

- Возможная логика расстановки полей
  - Просто сократить размер за счёт уменьшения padding’ов — достаточно жадно расставить поля от больших выравниваний к малым
  - Сгруппировать по локальности использования в горячих циклах — для этого нужно **PGO**
- Требует изменить **все** обращения к структуре
  - То есть нужен **анализ указателей**, чтобы выяснить, не утекают ли указатели
  - Крайне желательно **LTO**, чтобы поймать все обращения

GCC:	Clang:	LCC: Не релиз

Field Reordering

- Фактически нарушает стандарт C++: по стандарту должен сохраняться порядок полей
  - Если в программе есть сравнение адресов полей, то такая программа сломается
  - Сделать в оптимизации отлов таких обращений (чтобы оптимизацию не применять) — требует качественного **анализа указателей**

> *Non-variant non-static data members* of non-zero size (6.7.2) are allocated **so that later members have higher addresses within a class object** (7.6.9). Implementation alignment requirements can cause two adjacent members not to be allocated immediately after each other; so can requirements for space for managing virtual functions (11.7.3) and virtual base classes (11.7.2).

C++23: гл. 11.4.1 note 8

GCC:	Clang:	LCC: Не релиз

Unused Field Elimination

«То, что и написано на упаковке»

```
struct A_t // align = 8
{
    uint64_t hash; // 0 uses
    uint16_t mask;
    uint8_t  id;
}; // sizeof(A_t) = 16
```

```
struct A_t // align = 2
{
    // uint64_t hash;
    uint16_t mask;
    uint8_t  id;
}; // sizeof(A_t) = 4
```

Пример **Unused Field Elimination**

GCC:	Clang:	LCC: Не релиз

Unused Field Elimination

- Всё про **Field Reordering**
- По стандарту можно обращаться к *первому полю* простым изменением типа указателя
  - На C некоторые программисты пытаются изобразить ООП, создавая структуры-дети с первым полем в виде «базовой» структуры
  - Надо отлавливать такие связи, чтобы не удалить лишних де-факто используемых полей

GCC:	Clang:	LCC: Не релиз

Что смущает в обходе?


```
struct A_t { u64 key, val; };
struct B_t {
  A_t *arr; int sz;
};
int find(B_t *b, u64 k) {
  for (i = 0; i < b->sz; i++)
  {
    if (b->arr[i].key == k)
      return i;
  }
}
```

Structure Peeling

Разделение массива структур на массивы по полям

```
struct A_t { u64 key, val; };
struct B_t {
  A_t *arr; int sz;
};
int find(B_t *b, u64 k) {
  for (i = 0; i < b->sz; i++)
  {
    if (b->arr[i].key == k)
      return i;
  } // прокачиваем val
}   // через кэш
```

```
struct B_t {
  u64 *key; u64 *val; int sz;
};
int find(B_t *b, u64 k) {
  for (i = 0; i < b->sz; i++)
  {
    if (b->key[i] == k)
      return i;
  }
}
```

Пример **Structure Peeling**

GCC: ?	Clang:	LCC: -fdata-reorg

Structure Peeling

- Позволяет не прокачивать через кэш все не нужные поля
  - Но приводит к ухудшению локальности в тех циклах, где используется много полей
  - Бессмысленна без **PGO** (в LCC работает только с ним)
- Требует изменить **все** обращения к структуре
  - То есть нужен **анализ указателей**, чтобы выяснить, не утекают ли указатели
  - Крайне желательно **LTO**
- На задаче libquantum (SPEC CPU) ускорение на **52%** (замеры на Эльбрусе)

GCC: ?	**Clang:**	**LCC:** `-fdata-reorg`

Structure Splitting

Разделение массива структур на несколько массивов меньших структур

```
struct A_t {
  u64 key, val;
  A_t *next;    <- !!!
};
struct B_t {
  A_t *arr; int sz;
};

// Цикл тот же
```

```
struct A_hot_t {
  u64 key;
  A_cold_t *__pair;
};
struct A_cold_t {
  u64 val;
  A_hot_t *next;
};
struct B_t {
  A_hot_t *arr; int sz;
};
```

Пример **Structure Splitting**

GCC:	Clang:	LCC: -fdata-reorg

Structure Splitting

Здесь должна быть копипаста со слайда **Structure Peeling**, но вы и без неё помните?

:)

- На задачах mcf (SPEC CPU) ускорение на **12-25%** (замеры на Эльбрусе)

GCC:	**Clang:**	**LCC:** `-fdata-reorg`

C++ Russia

Где есть перечисленные оптимизации?

elbrus

	GCC	Clang	LCC
Prefetching	+	+	+
Loop Interchange	+	`-floop-interchange`	+
Loop Fusion	+	+	+
Array Padding			+
Array Transpose			`-fdata-reorg`
Unused Field Elimination			Не релиз
Field Reordering	Удалено за ненадёжность, попытки возвращения		Не релиз
Structure Peeling			`-fdata-reorg`
Structure Splitting			

GCC: следы в публикациях

- Как некоторое время работали оптимизации расположения данных в GCC:
  - Olga Golovanevsky and Ayal Zaks. **Struct-reorg: current status and future perspectives**. In *Proceedings of the GCC Developers' Summit*, 2007.
- Попытка возвращения Structure Peeling в GCC (2022) — следов в самом GCC найти не удалось:
  - Huang, Liangming & Jiang, Jun & Gao, Xiuwu. (2022). **Structure Peeling Based on Link Time Optimization in the GCC compiler**. 317-322. 10.1109/ISPDS56360.2022.9874019.

Выводы

- Располагать данные лучше руками
  - Не во всех компиляторах есть соответствующие оптимизации
  - Компилятор слишком зажат вопросами надёжности и эффективности
- Цикловым оптимизациям могут мешать зависимости или малое число итераций
  - Можно следить за этим с помощью **opt-report**
- Желательно использовать **LTO** и **PGO**

Что почитать и посмотреть

- Muchnick «**Advanced Compiler Design and Implementation**»
- Kowarschik, M., Weiß, C. (2003). **An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms**. *In: Meyer, U., Sanders, P., Sibeyn, J. (eds) Algorithms for Memory Hierarchies. Lecture Notes in Computer Science, vol 2625. Springer, Berlin, Heidelberg.*

Вопросы?

C++ Russia

Эльбрус: Array Prefetch Buffer

elbrus

Буфер предподкачки массивов:

- В коде — специальные инструкции загрузки
- В специальных секциях — асинхронная программа для буфера

Пример инструкции подкачки

```
movad,0 area = 2, ind = 8, am = 0, be = 0, %db[3]
```

Пример асинхронной программы

```
M_1cc90:
    apb ct=1, si=0, dcd=0, fmt=3, mrng=4, d=1,
        incr=1, ind=0, asz=5, abs=0, disp=0x0
    apb si=0, dcd=0, fmt=3, mrng=4, d=1,
        incr=1, ind=0, asz=5, abs=0, disp=0xa0
```

… а лучше посмотреть на сайте МЦСТ:
https://www.mcst.ru/doc/elbrus_prog/html/chapter6.html#reference-prefetch

C++ Russia

elbrus

Array Merging

Слияние массивов

```
uint32_t A[2048], B[2048];
uint32_t C[2048], D[2048];
uint32_t E[2048], F[2048];

for (i = 0; i < n; i++) {
  s += A[i]*B[i] + C[i]*D[i];
  s += C[i]/D[i] - E[i]/F[i];
}
```

```
uint32_t M[2048][6];

for (i = 0; i < n; i++) {
  s += M[i][0]*M[i][1]
     + M[i][2]*M[i][3];
  s += M[i][2]/M[i][3]
     - M[i][4]/M[i][5];
}
```

Пример **Array Merging** для того же цикла с конфликтами

GCC:	Clang:	LCC:

Array Merging

- В общем случае предназначена объединять данные, которые используются вместе
- Требует изменить **все** обращения
  - То есть нужен **анализ указателей**, чтобы выяснить, не утекают ли указатели
- Вариации:
  - Слияние элементов в массив новой нижней размерности, если тип элементов один и тот же
  - Слияние элементов в структуру, становящуюся элементом преобразованного единого массива

GCC:	Clang:	LCC:

Loop Blocking/Tiling

- Циклы на все размерности делятся на циклы по длине блока
- **Loop Interchange** циклов по длине блока во вложенные

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    s += A[i][j] * B[i][j];
  }
}
```

→

```
for (i = 0; i < n; i += T) {
 for (i1 = i; i1 < i+T; i1++) {
  for (j = 0; j < n; j += T) {
   for (j1 = j; j1 < j+T; j1++) {
    s += A[i1][j1] * B[i1][j1];
  }}
}}
```

Пример **Loop Tiling** (блоки размером $T \times T$) в гнезде из 2 циклов (обходе матрицы) при условии, что n кратно T

GCC:	Clang:	LCC:

Loop Blocking/Tiling

- Циклы на все размерности делятся на циклы по длине блока
- **Loop Interchange** циклов по длине блока во вложенные

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
    s += A[i][j] * B[i][j];
  }
}
```

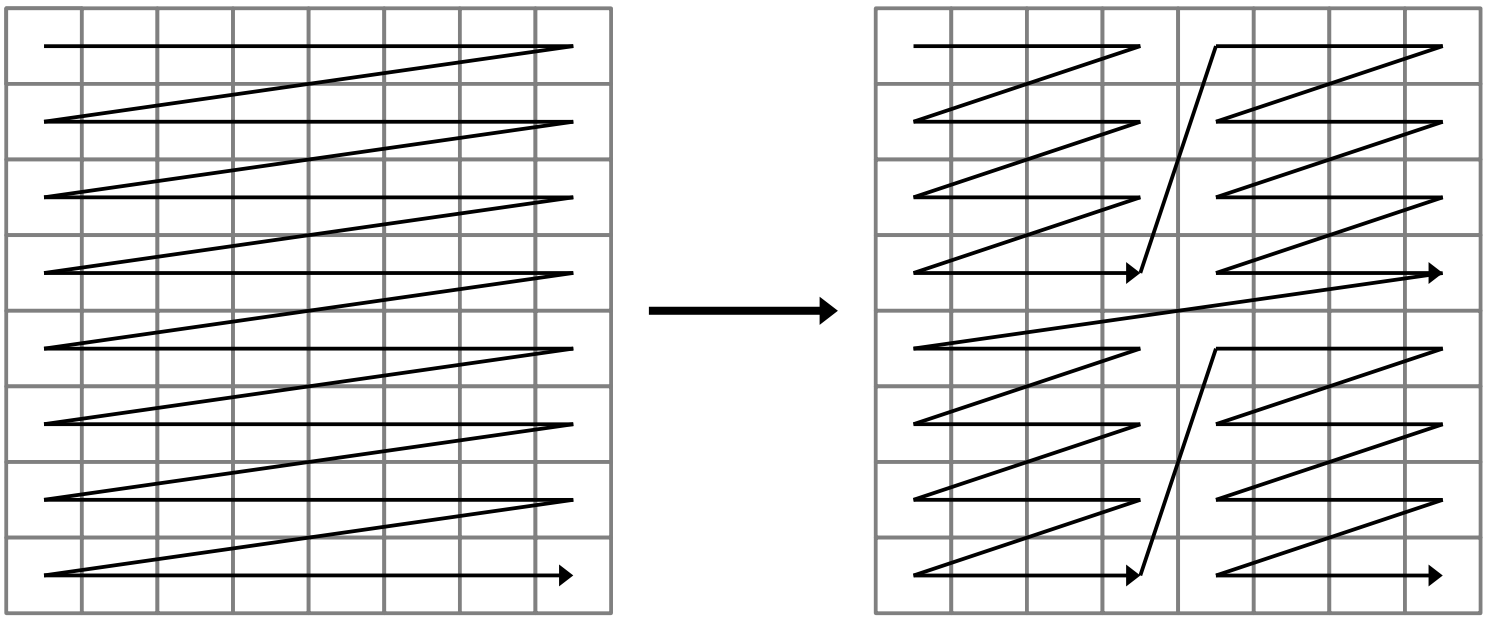
→

```
for (i = 0; i < n; i += T) {
 for (j = 0; j < n; j += T) {
  for (i1 = i; i1 < i+T; i1++) {
   for (j1 = j; j1 < j+T; j1++) {
    s += A[i1][j1] * B[i1][j1];
  }}
}}
```

Пример **Loop Tiling** (блоки размером $T \times T$) в гнезде из 2 циклов (обходе матрицы) при условии, что n кратно T

GCC:	Clang:	LCC:

Loop Blocking/Tiling



Пример направления обхода матрицы A до и после **Loop Tiling**

GCC:	Clang:	LCC:

Loop Blocking/Tiling

- Легальна, когда легален **Loop Interchange** созданных блочных цикл во вложенные
- Может улучшить **локальность**
- Размеры блока можно подбирать из разных соображений
  - Например, из расчёта помещения всех нужных данных блока в кэш нужного уровня
  - Или из расчёта оптимальной векторизации вычислений в блоке

GCC:	Clang:	LCC: