УДК 004.4'422 DOI: 10.17587/prin.12.82-88

В. Е. Шампаров, инженер-программист, Victor.E.Shamparov@mcst.ru,
А. Л. Маркин, начальник сектора, Alex.L.Markin@mcst.ru, АО "МЦСТ",
Московский физико-технический институт

Механизм оптимизации Structure Splitting в составе компилятора для микропроцессоров Эльбрус

Предложена новая версия оптимизации Structure Splitting, реализованная в составе компилятора для микропроцессоров с архитектурами Эльбрус и SPARC. Structure Splitting предназначена для улучшения локальности данных с помощью преобразования массивов структур в массивы из структур меньшего размера. Оптимизация была применена к варианту массива структур, вложенного в другую структуру, для которого память может быть перевыделена. При применении разработанного механизма оптимизации скорость исполнения двух тестов из наборов SPEC CPU2000 и SPEC CPU2006 увеличилась на 19 и 12 % соответственно.

Ключевые слова: *компилятор, оптимизация, Structure Splitting, Эльбрус, SPARC*

Введение

Улучшение работы программы с памятью, особенно с кешем, является одной из ключевых задач оптимизирующего компилятора. Для этого в компиляторы включают различные методы, среди которых оптимизации расположения данных в памяти (*data layout optimizations*). Эти механизмы увеличивают степень локальности данных с помощью их перераспределения в памяти. Благодаря таким механизмам уменьшается вероятность промахов кеша и, как следствие, уменьшается время исполнения программы.

Подмножеством механизмов оптимизации расположения данных в памяти являются механизмы реорганизации структур, увеличивающие локальность расположения их полей. Среди таких способов оптимизаций можно отметить Structure Peeling [1] и Structure Splitting, преобразующие массив структур в несколько массивов из структур меньшего размера. В Structure Splitting, в отличие от Structure Peeling, учтена возможность наличия указателей на структуры преобразуемых типов среди изменяемых структур.

В данной работе представлен реализованный авторами механизм оптимизации (далее для краткости — оптимизация) Structure Splitting, обобщенный на случай вложенного в структуру динамически выделенного массива структур, размер которого может изменяться во время исполнения программы. В разд. 1 приведена решаемая созданной оптимизацией проблема. Созданный для решения проблемы алгоритм описан в разд. 2. В разд. 3 приведены полученные результаты. Связанные с темой данной работы публикации описаны в разд. 4. В Заключении подведены итоги проделанной работы и описаны планы дальнейших исследований над оптимизациями реорганизации структур.

1. Решаемая задача

В ходе исследования пакета тестирования SPEC CPU2000 [2] в одной из задач (181.mcf) была обнаружена ситуация, в которой в структуру было вложено несколько массивов структур. Пример на рис. 1 показывает упрощенное описание иерархии структур из задачи 181.mcf. Массивы структур node_t* nodes, arc_t* arcs и arc_t* dummy_arcs

```
typedef struct node
{
    ...
    struct node *pred, *child, *sibling, *sibling_prev;
    ...
    struct arc *basic_arc;
    struct arc *firstout, *firstin;
    ...
} node_t;

typedef struct arc
{
    node_t *tail, *head;
    struct arc *nextout, *nextin;
    int32_t cost;
    ...
} arc_t;

typedef struct network
{
    ...
    node_t *nodes, *stop_nodes;
    arc_t *arcs, *stop_arcs;
    arc_t *dummy_arcs, *stop_dummy;
    ...
} network_t;
```

Рис. 1. Упрощенная иерархия структур из задачи 181.mcf

```
network_t *net;
arc_t *arc;
arc_t *stop;
int new_cost;

arc = net->arcs;
for ( stop = (void*) net->stop_arcs; arc < stop; arc++ )
    arc->cost = new_cost;
```

Рис. 2. Упрощенный пример неэффективного цикла для иерархии структур из задачи 181.mcf

вложены в структуру network_t, а все остальные указатели типов node_t* и arc_t* внутри структур node_t размером 32 байта, arc_t размером 60 байт и network_t являются указателями на элементы трех перечисленных массивов. Хранение данных в виде массива структур может быть неэффективно, если в коде программы встречаются циклы, в которых обращаются только к небольшой части полей элемента массива. Пример подобного цикла с использованием иерархии структур из задачи 181.mcf приведен на рис. 2. При обращении к полю cost (поле имеет сдвиг 16 байт относительно начала структуры arc_t) элемента массива в кеш-строку загружается вся структура размером 32 байта, из которых 28 байт в данном цикле не нужны. Пример кеш-строки на первой итерации приведенного цикла показан на рис. 3. Вследствие загрузки в кеш-память ненужных данных частота промахов кеша выше, чем может быть.

Схожий шаблон иерархии структур был также найден в задаче 429.mcf из пакета SPEC CPU2006. Для улучшения локальности данных в этой иерархии требуется:

1) разделить поля каждой из структур node_t и arc_t на часто и редко используемые и затем объединить часто используемые поля (далее будем называть их горячими) в структуры node_hot и arc_hot, а редко используемые поля (далее будем называть их холодными) — в структуры node_cold и arc_cold;

2) в структуры node_hot и arc_hot добавить по указателю на соответствующие структуры node_cold и arc_cold;

3) создать из каждого массива структур, вложенного в network_t, массив горячих структур и массив холодных структур;

4) все указатели на элементы преобразованных массивов структур преобразовать в указатели на соответствующие элементы соответствующего массива горячих структур.

Преобразованные структуры показаны на рис. 4.

Подобная оптимизация называется Structure Splitting и реализована в компиляторах GCC [3], Open64 [4] и LLVM [5]. Следует однако отметить, что

Рис. 3. Содержимое кеш-строки на первой итерации цикла из рис. 2

```
typedef struct node_hot
{
    ...
    struct node_hot *pred;
    ...
    struct arc_hot *basic_arc;
    ...
    struct node_cold *cold_node_ptr;
} node_hot;
typedef struct node_cold
{
    ...
    struct node_hot *child, *sibling, *sibling_prev;
    ...
    struct arc_hot *firstout, *firstin;
    ...
} node_cold;

typedef struct arc_hot
{
    node_hot *head;
    struct arc_hot *nextin;
    ...
    struct arc_cold* cold_arc_ptr;
} arc_hot;
typedef struct arc_cold
{
    node_hot *tail;
    struct arc_hot *nextout;
    ...
} arc_cold;

typedef struct network
{
    ...
    node_hot *hot_nodes;
    node_hot *stop_nodes;
    arc_hot *hot_arcs;
    arc_hot *stop_arcs;
    arc_hot *hot_dummy_arcs;
    arc_hot *stop_dummy;
    ...
    node_cold *cold_nodes;
    arc_cold *cold_arcs;
    arc_cold *cold_dummy_arcs;
} network_t;
```

Рис. 4. Преобразованная иерархия структур в задаче 181.mcf

в задаче 181.mcf в функции resize_prob обнаружено изменение размера одного из массивов с помощью стандартной функции realloc и алгоритма изменения указателей на элементы внутри этого массива. Упрощенный пример указанного алгоритма приведен на рис. 5. В силу наличия данного кода указанные реализации оптимизации Structure Splitting неприменимы. Поэтому появляется необходимость разработать новый алгоритм преобразования компилируемой программы с учетом применения процедуры realloc.

```
network_t *net;
arc_t *arc;
node_t *node, *stop, *root;
long off;

arc = (arc_t *) realloc( net->arcs, new_size * sizeof(arc_t) );
off = (long)arc - (long)net->arcs;

net->arcs = arc;
node = net->nodes;
for( node++, stop = (void*)net->stop_nodes; node < stop; node++ )
    node->basic_arc = (arc_t *)((long)node->basic_arc + off);
```

Рис. 5. Упрощенный пример алгоритма изменения указателей после `realloc` в задаче 181.mcf

2. Реализация

В рамках работы, результаты которой представлены в настоящей статье, авторами реализована новая версия оптимизации Structure Splitting в оптимизирующем компиляторе lcc для процессорных архитектур Эльбрус и SPARC. Алгоритм подобно описанным в работах [3, 4] состоит из двух этапов:

1) анализ применимости оптимизации — поиск подходящих для оптимизации массивов структур;

2) применение оптимизации.

2.1. Анализ применимости

Задача этапа 1 состоит в выявлении всех массивов структур и определении того, к каким из них возможно применить оптимизацию. Для возможности выявления всех искомых структур необходимо, чтобы компилятор работал в режиме сборки "вся программа", при котором все модули компилируемой программы объединяются в один.

В работе [6] описан алгоритм анализа применимости на основе профильной информации. Этот алгоритм выбирает структуры для оптимизации на основе счетчика всех обращений ко всем полям структур, суммарных счетчиков обращений к полям конкретных структур и счетчиков обращений к каждому полю. Решение по каждой структуре принимается на основе формулы, включающей все перечисленные счетчики. В ходе проделанной работы указанный алгоритм модернизирован и состоит из трех этапов.

Первый этап заключается в сборе данных обо всех структурах и полях структур в программе:

- для каждой структуры — число обращений к объектам типа этой структуры (назовем это число счетчиком структуры);
- для каждого поля каждой структуры — число обращений к этому полю (назовем это число счетчиком поля).

Полученные данные записываются в ориентированный граф *Struct Nesting Graph* (SNG). Узлы графа SNG — все найденные структуры. Если структура B или указатель на нее является полем f другой структуры A, то между ними проводится ребро $A \xrightarrow{f} B$.

На рис. 6 показан упрощенный пример иерархии структур. В данном примере поля data0 и data1 в структуре BigStruct — массивы структур, а все остальные поля типов InnerStruct0* и InnerStruct1* — указатели на элементы массивов data0 и data1. Пример графа SNG для описанной в этом примере иерархии изображен на рис. 7.

```
typedef struct _InnerStruct0 InnerStruct0;
typedef struct _InnerStruct1 InnerStruct1;

struct _InnerStruct0 {
    int   hot0_0;
    float cold0_0;
    InnerStruct1 *hot0_1;
};
struct _InnerStruct1 {
    int   hot1_0;
    float cold1_0;
    InnerStruct1 *cold1_1;
};

typedef struct _BigStruct {
    int b0;
    InnerStruct0 *data0;
    InnerStruct1 *data1;
    int b1;
} BigStruct;

typedef struct _OtherStruct {
    int o0;
    InnerStruct0 *ptr0;
    int o1;
} OtherStruct;
```

Рис. 6. Пример иерархии структур

Кроме графа, на первом этапе формируются таблицы *Field Counters Table* (FCT) и *Struct Counters Table* (SCT). FCT содержит для каждого поля его счетчик, а SCT для каждой структуры — ее счетчик.

Второй этап заключается в определении подходящих для применения оптимизации массивов структур. Для этого в графе *Struct Nesting Graph* проводится поиск всех ребер, поля на которых являются указателями. Таким образом находятся поля, которые

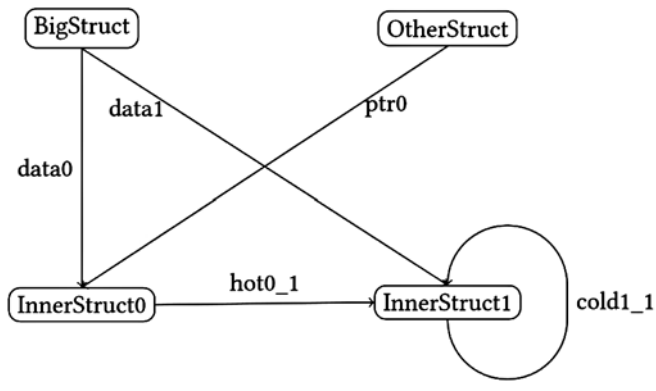

Рис. 7. Граф SNG к рис. 6

могут оказаться вложенными массивами структур. Эти поля записываются в таблицу *Fields for Optimization Table* (FOT).

Далее для каждого поля F_i в таблице FOT, соединяющего структуры A_i и B_i $\left(A_i \xrightarrow{F_i} B_i\right)$ проводятся следующие действия:

1) определение счетчиков $c_{i,j}$ для каждого поля $f^{i,j}$ структуры B_i;

2) определение максимального счетчика $C_i = \max_j\{c_{i,j}\}$ среди счетчиков полей структуры B_i;

3) определение полей $f_{i,j}$, удовлетворяющих условию $c_{i,j} > \frac{C_i}{N}$ (N — некая константа), такие поля считаются горячими.

Из таблицы FOT исключаются все поля F_i $\left(A_i \xrightarrow{F_i} B_i\right)$, для которых не выполнено хотя бы одно из условий:

- в программе есть вызов функции выделения памяти для поля F_i;
- среди полей структуры B_i есть и горячие, и холодные.

Результатом анализа применимости является таблица FOT, в которой после описанных выше действий остались только те вложенные массивы структур, к которым надо применить оптимизацию Structure Splitting.

2.2. Применение оптимизации

Задача этапа 2 оптимизации Structure Splitting состоит в изменении определенных ранее типов структур, предназначенных для оптимизации, и промежуточного представления программы. Алгоритм применения оптимизации по составу этапов похож на описанные в работах [4, 6]. Он состоит из следующих стадий:

1) создание новой иерархии полей структур;

2) изменение типов переменных компилируемой программы согласно новой иерархии;

3) изменение доступа к переменным в компилируемой программе.

2.2.1. Создание новой иерархии полей структур

В ходе данного этапа для каждой записи в таблице FOT (поля F_i, в графе SNG, являющегося ребром $A_i \xrightarrow{F_i} B_i$), выполняются следующие действия:

1) структура B_i (элемент массива) разрезается на структуру с горячими полями H_i и структуру с холодными полями C_i; в H_i добавляется указатель на C_i; данное действие назовем декомпозицией;

2) в структуре A_i (т. е. охватывающей) тип поля F_i изменяется с указателя на B_i на указатель на H_i; в конец структуры A_i добавляется указатель на C_i, имеющий смысл указателя на начало массива холодных структур;

3) во всех структурах N_j, для которых в графе SNG существуют ребра $N_j \xrightarrow{E_j} B_i$, типы полей E_j изменяются с указателя на B_i на указатель на H_i.

На рис. 8 показан набор структур, представленных на рис. 6, преобразованный в ходе данного этапа.

```
typedef struct _InnerStruct0_Hot InnerStruct0_Hot;
typedef struct _InnerStruct1_Hot InnerStruct1_Hot;
typedef struct _InnerStruct0_Cold InnerStruct0_Cold;
typedef struct _InnerStruct1_Cold InnerStruct1_Cold;

struct _InnerStruct0_Hot {
    int   hot0_0;
    InnerStruct1_Hot *hot0_1;
    InnerStruct0_Cold *cold_0_ptr;
};
struct _InnerStruct0_Cold {
    float cold0_0;
};
struct _InnerStruct1_Hot {
    int   hot1_0;
    InnerStruct0_Cold *cold_1_ptr;
};
struct _InnerStruct1_Cold {
    float cold1_0;
    InnerStruct1_Hot *cold1_1;
};

typedef struct _BigStruct {
    int b0;
    InnerStruct0_Hot *data0_hot;
    InnerStruct1_Hot *data1_hot;
    int b1;
    InnerStruct0_Cold *data0_cold;
    InnerStruct1_Cold *data1_cold;
} BigStruct;

typedef struct _OtherStruct {
    int o0;
    InnerStruct0_Hot *ptr0;
    int o1;
} OtherStruct;
```

Рис. 8. Преобразованный набор структур из рис. 6

2.2.2. Изменение типов переменных компилируемой программы

В ходе данного этапа все типы переменных — указателей на декомпозированные структуры — заменяются на указатели на созданные при декомпозиции горячие структуры.

2.2.3. Изменение доступа к переменным в компилируемой программе

В ходе данного этапа выполняется преобразование промежуточного представления программы с изменением всех обращений к декомпозированным массивам и их элементам. Обращения бывают нескольких категорий. Для каждой категории ниже приведен пример оригинального и преобразованного обращения. В представленных примерах:

- data — указатель на начало массива до преобразования иерархии структур; data_hot и data_cold — указатели на начала горячего и холодного массивов после преобразования иерархии структур (для краткости будем называть эти указатели горячим и холодным соответственно);
- ptr — некий указатель на элемент массива до преобразования переменных программы; ptr_hot —

тот же указатель после преобразования переменных программы;

- field_hot и field_cold — горячее и холодное поля элемента массива, к которому осуществляется доступ;
- data_elem_cold — созданный внутри горячей структуры в ходе преобразования иерархии структур указатель на соответствующую холодную структуру;
- s — число элементов в массиве; s_new — число элементов в массиве после вызова realloc;
- ArrElem — структура элемента массива; ArrElemHot и ArrElemCold — структуры элементов горячего и холодного массивов, созданные из ArrElem в ходе преобразования иерархии структур.

Отметим категории обращений и соответствующие им преобразования.

1. Работа с полем элемента массива:

1.1) доступ к горячему полю элемента массива с помощью указателя на начало массива и индекса data[index].field_hot; преобразование состоит в изменении указателя на начало массива: data_hot[index].field_hot;

1.2) доступ к горячему полю элемента массива с помощью указателя на элемент массива ptr->field_hot; преобразование состоит в изменении указателя на элемент массива: ptr_hot;

1.3) доступ к холодному полю элемента массива с помощью указателя на начало массива и индекса data[index].field_cold; преобразование состоит в изменении указателя на начало массива: data_cold[index].field_cold;

1.4) доступ к холодному полю элемента массива с помощью указателя на элемент массива ptr->field_cold; преобразование состоит в изменении указателя на элемент массива и получении указателя на холодный элемент массива из указателя на горячий элемент массива: ptr->data_elem_cold->field_cold.

2. Работа с самим массивом:

2.1) выделение или изменение размера выделенной для массива памяти: data = (ArrElem*) malloc(s*sizeof(ArrElem)); преобразование состоит в изменении размера выделяемой памяти для двух созданных массивов data_hot и data_cold, создании операции вычисления указателя на начало холодного массива и создании цикла с вычислением указателей на холодные элементы массива для записи в горячие; для realloc добавляется перемещение данных холодного массива по новому адресу; пример преобразованного выделения памяти показан на рис. 9 и 10; для сохранения корректности указателей во время алгоритма после realloc, описанного на рис. 5, вместо раздельного выделения памяти для обоих созданных массивов требуется выделить для них одну область памяти и разметить одну часть для горячего массива, а другую — для холодного;

```
size_t new_sizeof = sizeof(ArrElemHot) + sizeof(ArrElemCold);
data_hot = (ArrElemHot*) malloc(s * new_sizeof);
data_cold = (ArrElemCold*)( data_hot + s );
for ( i = 0; i < s; i++ ) {
    data_hot[i]->data_elem_cold = &data_cold[i];
}
```

Рис. 9. Преобразованное выделение памяти для массива

```
size_t new_sizeof = sizeof(ArrElemHot) + sizeof(ArrElemCold);
data_hot = (ArrElemHot*) realloc(data_hot, s_new * new_sizeof);
ArrElemCold* old_data_cold = (ArrElemCold*)( data_hot + s );
data_cold = (ArrElemCold*)( data_hot + s_new );
memmove( data_cold, old_data_cold, s*sizeof(ArrElemCold) );
for ( i = 0; i < s_new; i++ ) {
    data_hot[i]->data_elem_cold = &data_cold[i];
}
```

Рис. 10. Преобразованное перевыделение памяти для массива

2.2) освобождение памяти для массива: free(data); преобразование состоит в изменении подаваемого указателя free на указатель на начало горячего массива: free(data_hot); так как для обоих массивов выделяется одна область памяти, достаточно вызвать функцию освобождения памяти для начала этой области, в данном случае начало области совпадает с началом горячего массива;

2.3) сравнение указателя на начало массива с нулем: data == 0; преобразование состоит в изменении сравниваемого указателя на горячий: data_hot == 0;

2.4) присвоение нуля указателю на начало массива: data = 0; преобразование состоит в присвоении нуля и горячему, и холодному указателям на начала массивов: data_hot = 0; data_cold = 0.

3. Результаты измерений

Разработана оптимизация Structure Splitting. Измерения были проведены на компьютере с процессором Эльбрус-4С частотой 750 МГц с архитектурой системы команд типа VLIW [7].

Реализованную оптимизацию удалось применить к нескольким задачам.

1. Задачи 181.mcf (SPEC CPU2000) и 429.mcf (SPEC CPU2006), представляющие собой реализации симплекс-метода оптимизации расписания движения транспорта, созданные А. Лёбелем (институт Цузе) для оптимизации транспортных систем Берлина и иных городов [8].

2. GeoNoise&Water — авторская реализация алгоритма генерации случайных карт "diamond-square" и наивного алгоритма моделирования водной эрозии на сгенерированной карте.

При применении реализованной оптимизации к задаче 181.mcf удалось получить ускорение выполнения на 26 %. Ускорение произошло вследствие уменьшения числа блокировок конвейера процессора для ожидания загрузки данных из памяти на 27 %.

Задача 429.mcf после применения оптимизации ускорилась на 14 %. Ускорение произошло в силу уменьшения числа блокировок конвейера процессора для ожидания загрузки данных из памяти на 19 %.

Задача GeoNoise&Water после применения оптимизации ускорилась на 4 %. Ускорение произошло вследствие уменьшения числа блокировок конвейера процессора для ожидания загрузки данных из памяти на 54 %.

4. Работы по смежной тематике

В работе [3] описан иной алгоритм оптимизации Structure Splitting, выполненной для компилятора GCC.

Отличия целей разработки указанной оптимизации от целей данной работы состоят в следующем:

- перед версией для GCC не стоит задачи обеспечения работы при изменении размера массива, поэтому в ходе оптимизации обрабатываются только стандартные функции выделения и освобождения памяти для массива;
- не обрабатываются случаи, когда указатели на элементы массива структур могут лежать в других структурах.

Описанный в работе [3] алгоритм состоит из двух этапов:

1) определение необходимости применения оптимизации;

2) применение оптимизации.

Определение необходимости проведения оптимизации проводится по составленным для каждой процедуры графам *Field Reference Graph* (FRG) и *Close Proximity Graph* (CPG). FRG — ориентированный граф, узлами которого являются операции доступа к полям исследуемой структуры, а дуги определяют порядок доступа к полям. Каждой дуге соответствует счетчик, взятый из профилированного графа потока управления процедуры, и дистанция — число доступов к памяти между доступом к полям. Второй граф является упрощенным графом FRG, в котором рассматриваются только дуги с дистанцией, меньшей некой определенной заранее. Также в CPG по сравнению с FRG множественные дуги между узлами сведены к единичным. На основе анализа графа CPG выносится решение о необходимости применения оптимизации.

Применение Structure Splitting в версии GCC состоит из следующих этапов:

1) создание новой иерархии полей структур;

2) изменение типов переменных компилируемой программы согласно новой иерархии;

3) изменение доступа к полям согласно новой иерархии;

4) изменение обращений к функциям выделения и освобождения памяти;

5) изменение доступа к элементам преобразованного массива согласно новой иерархии.

Способ оптимизации Structure Splitting в указанном выше варианте применен в компиляторе GCC версии 4.3 [8]. В версии 4.8 эта оптимизация была убрана на доработку из-за вопросов, связанных с надежностью.

В работе [6] предлагается другой способ анализа применимости оптимизации для программ на языке Java. Он состоит в подсчете полных счетчиков обращений к полям и структурам и определении горячих полей на основе:

- S — счетчика обращений к структуре элемента массива;
- N — числа полей в элементе массива;
- s_i — счетчиков обращений к полю i элемента массива.

Горячими считаются поля, для которых $s_i > \frac{S}{2N}$.

Немного отличается от описанного в статье [6] анализ применимости, описанный в работе [4]. В нем применены профили каждого узла графа потока управления программы.

Предложенный алгоритм анализа состоит из следующих этапов для каждой структуры.

1. Для каждого горячего (т. е. с большим счетчиком выполнения) узла выяснение, к каким полям исследуемой структуры проводятся обращения и сколько раз.

2. Разделение полей исследуемой структуры на группу горячих и группу холодных согласно определенным на предыдущем этапе данным.

Применение оптимизации, описанное в работе [4], не отличается по принципу работы от описанного в работе [2].

Оптимизация Structure Splitting в варианте, описанном в работе [4], применена в компиляторе Open64.

В работе [9] описан полностью иной подход к Structure Splitting, реализованный с помощью отдельной программы, названной *On-the-fly Structure Splitting* (OSS). Подход состоит в профилировании исполняемого файла с перехватом обращений программы для выделения и освобождения памяти, в анализе и внесении изменений в исполняемый файл.

Заключение

Разработана и отлаживается новая версия оптимизации Structure Splitting, обобщенная на случай изменения размера массива структур с применением алгоритма сохранения корректности указателей на элементы измененного массива.

Достигнуто ускорение работы теста 181.mcf на 19 %. Задача 429.mcf ускорилась на 12 %, хотя при ручном применении оптимизации удалось достигнуть ускорения данной задачи на 20 %.

Дальнейшие направления работы заключаются в подборе констант для определения горячих и холодных полей, чтобы сделать ускорение кода максимальным и исключить случаи замедления кода вследствие применения оптимизации.

Список литературы

1. **Шампаров В. Е., Маркин А. Л.** Преобразование компилятором массива структур в несколько массивов // Труды 61-й Всероссийской научной конференции МФТИ. Радиотехника и компьютерные технологии. МФТИ. — 2018. — С. 9—10.

2. **SPEC CPU.** URL. https://www.spec.org/benchmarks.html#cpu

3. **Hagog M., Tice C.** Cache Aware Data Layout Reorganization Optimization in GCC // Proceedings of the GCC Developers' Summit. Ottawa, Ontario. — 2005. — P. 69—92.

4. **Chakrabarti G., Chow F.** Structure layout optimizations in the Open64 compiler: Design, implementation and measurements // Open64 workshop held in conjunction with the international symposium on code generation and optimization, 2008. URL: https://www.capsl.udel.edu//conferences/open64/2008/Papers/111.pdf

5. **Prashantha N. R., Vikram T. V., Vaivaswatha N.** Implementing Data Layout Optimizations in the LLVM Framework. 2014. URL: http://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf

6. **Chilimbi T., Davidson B., Larus J. R.** Cache-Conscious Structure Definition // Proceedings of the ACM SIGPLAN'99 Con-

ference on Programming Language Design and Implementation. ACM. — 1999. — P. 13—24.
7. **Ким А. К., Перекатов В. И., Ермаков С. Г.** Микропроцессоры и вычислительные комплексы семейства "Эльбрус". — СПб.: Питер, 2013. — 272 с.
8. **Loebel A.** Optimal Vehicle Scheduling in Public Transit: Ph.D. thesis. — Berlin, 1997. — 185 p.
9. **Golovanevsky O., Zaks A.** Struct-reorg: current status and future perspectives // Proceedings of the GCC Developers' Summit. Ottawa. — Ontario, 2007. — P. 47—56.
10. **Wang Zh., Wu Ch., Yew P.-Ch., Li J., Xu D.** On-the-Fly Structure Splitting for Heap Objects ACM Trans. Archit. Code Optim. — 2012. — Vol. 8, No. 4. — Article 26. URL: https://doi.org/10.1145/2086696.2086705

Structure Splitting for Elbrus Processor Compiler

V. E. Shamparov, Victor.E.Shamparov@mcst.ru, **A. L. Markin,** Alex.L. Markin@mcst.ru, MCST, Moscow, 117105, Russian Federation, Moscow Institute of Physics and Technology, Moscow Region, 141701, Russian Federation

Corresponding author:

Shamparov Viktor E., Software Engineer, MCST, Moscow, 117105, Russian Federation
E-mail: Victor.E. Shamparov@mcst.ru

Received on October 13, 2020
Accepted on November 21, 2020

This report presents a new version of Structure Splitting optimization, implemented for the compiler for Elbrus and SPARC processors. Structure Splitting tries to improve data locality by splitting arrays of structures into arrays of smaller structures. This solution helps to decrease probability of cache misses, which leads to execution time decrease. The optimization was generalized for the case of array of structures nested in another structure and possibility of its reallocation. Execution speed of two tests from SPEC CPU2000 and SPEC CPU2006 increased by 19 and 12 %.

Keywords: *compiler, optimization, Structure Splitting, Elbrus, SPARC*

For citation:

Shamparov V. E., Markin A. L. Structure Splitting for Elbrus Processor Compiler, *Programmnaya Ingeneria,* 2021, vol. 12, no. 2, pp. 82—88

DOI: 10.17587/prin.12.82-88

References

1. **Shamparov V. E., Markin A. L.** Compiler transformation of array of structures into arrays, *Trudy 61-j Vserossijskoj nauchnoj konferencii MFTI. Radiotekhnika i komp'yuternye tekhnologii,* 2018, pp. 9—10 (in Russian).
2. **SPEC** CPU, available at: https://www.spec.org/benchmarks.html#cpu
3. **Hagog M., Tice C.** Cache Aware Data Layout Reorganization Optimization in GCC, *Proceedings of the GCC Developers' Summit*, Ottawa, Ontario, 2005, pp. 69—92.
4. **Chakrabarti G., Chow F.** Structure layout optimizations in the Open64 compiler: Design, implementation and measurements, *Open64 workshop held in conjunction with the international symposium on code generation and optimization,* 2008, available at: https://www.capsl.udel.edu//conferences/open64/2008/Papers/111.pdf
5. **Prashantha N. R., Vikram T. V., Vaivaswatha N.** Implementing Data Layout Optimizations in the LLVM Framework in the LLVM Framework, 2014, available at: http://llvm.org/devmtg/2014-10/Slides/Prashanth-DLO.pdf
6. **Chilimbi T., Davidson B., Larus J. R.** Cache-Conscious Structure Definition, *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, ACM, 1999, pp. 13—24.
7. **Kim A. K., Perekatov V. I., Ermakov S. G.** *Mikroprocessory i vychislitel'nye kompleksy semejstva "El'brus"*, Saint-Petersburg, Piter, 2013, 272 p. (in Russian).
8. **Loebel A.** Optimal Vehicle Scheduling in Public Transit: Ph.D. thesis. Berlin, 1997, 185 p.
9. **Golovanevsky O., Zaks A.** Struct-reorg: current status and future perspectives, *Proceedings of the GCC Developers' Summit*, 2007, pp. 47—56.
10. **Wang Zh., Wu Ch., Yew P.-Ch., Li J., Xu D.** On-the-Fly Structure Splitting for Heap Objects, *ACM Trans. Archit. Code Optim.*, 2012, vol. 8, no. 4, article 26.