DOI: 10.15514/ISPRAS-2021-33(3)-4

Data Layout Optimization for the LCC Compiler

[1,2] *V.E. Shamparov, ORCID: 0000-0002-0938-3824 <Victor.E.Shamparov@mcst.ru>*
[2] M.*I. Neiman-zade, ORCID: 0000-0002-4250-9724 <Murad.I.Neiman-zade@mcst.ru>*
[1] *MCST,*
24 Vavilova str., Moscow, 119334, Russia
[2] *Moscow Institute of Physics and Technology,*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia

Abstract. In this research-in-progress report, we propose a novel approach to unified cache usage analysis for implementing data layout optimizations in the LCC compiler for the Elbrus and SPARC architectures. The approach consists of three parts. The first part is generalizing two methods of estimating cache miss amount and choosing more applicable one in the compiler. The second part is finding an applicable solution for the problem of cache miss amount minimization. The third part is implementing this analysis in the compiler and using analysis results for data layout transformations.

Keywords: Compilers; Compiler Optimization; Cache Analysis; Data Layout Transformation

For citation: Shamparov V.E., Neiman-zade M.I. Data Layout Optimization for the LCC Compiler. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 3, 2021, pp. 51-60. DOI: 10.15514/ISPRAS-2021-33(3)-4.

Оптимизации расположения данных для компилятора LCC для архитектуры Эльбрус

[1,2] *В.Е. Шампаров, ORCID: 0000-0002-0938-3824 <Victor.E.Shamparov@mcst.ru>*
[2] *М.И. Нейман-заде, ORCID: 0000-0002-4250-9724 <Murad.I.Neiman-zade@mcst.ru>*
[1] *АО «МЦСТ»*
Россия, 119701, Москва, ул. Вавилова, д. 24
[2] *Московский физико-технический институт,*
Россия, 141701, Московская область, г. Долгопрудный, Институтский пер., 9

Аннотация. В данной статье о проводимом исследовании мы предлагаем новый подход к единому анализу использования кэш-памяти для разработки оптимизаций расположения данных в составе компилятора LCC для архитектур Эльбрус и SPARC. Подход состоит из трёх частей. Первая часть - обобщение двух методов оценки количества кэш-промахов и выбор из них более подходящего для реализации в компиляторе метода. Вторая часть - поиск применимого в компиляторе решения задачи минимизации количества промахов кэша. Третья часть - реализация выбранного метода анализа в компиляторе и использование результатов анализа для оптимизаций расположения данных.

Ключевые слова: компиляторы; оптимизации компилятора; анализ кэша; оптимизации расположения данных.

Для цитирования: Шампаров В.Е., Нейман-заде М.И. Оптимизации расположения данных для компилятора LCC для архитектуры Эльбрус. Труды ИСП РАН, том 33, вып. 3, 2021 г., стр. 51-60 (на английском языке). DOI: 10.15514/ISPRAS–2021–33(3)–4.

1. Introduction

Improving computer resources usage efficiency by a program is one of the main tasks for optimizing compilers. Particularly, improving memory usage is especially important because hardware developers have introduced multi-level intermediate memory, called cache memory, due to the growing performance difference between memory and CPU. Cache memory capabilities must be used efficiently.

Cache memory is structured for using the following program properties effectively *temporal locality* and *spatial locality*. *Temporal locality* means that the program often works with the same data in memory. *Spatial locality* means that the program is likely to work with adjacent data. Thus, to make compiled program use cache memory efficiently, the compiler must improve these two programs' properties.

Nowadays, compilers optimize the programs' temporal locality well by loop optimizations, but optimizing spatial locality is more complicated since it requires choosing the correct data structures for the program. Therefore, optimizing spatial locality is often entrusted to the programmer, although data location optimizations are implemented for some relatively simple cases.

In this article, we describe the ongoing research on cache memory usage for the further development of a high-quality automatic cache usage analysis in the compiler for applying an optimal set of data layout optimizations.

The article is organized as follows. In section 2, we substantiate the potential effect of optimizing data layout. In section 3, we state the problem. In section 4, we analyze papers on this topic and related ones. In section~5, we propose further research approach. In section 6, we describe current progress. Finally, in section 7, we provide a conclusion.

2. Motivation

It is known that part of program execution time is spent waiting for data from memory. This is especially evident for processors with in-order execution. They have fewer opportunities to mask this wasted time by executing other instructions than processors with out-of-order execution.

To illustrate this problem and determine the potential effect of optimization, we measured the percentage of test execution time from SPEC~CPU benchmark packages that the processor spends waiting for data from memory. This data is shown in Table 1. We used a computer with an Elbrus-4C processor for measurement. It has VLIW ISA, in-order execution and two-level cache memory. Benchmarks were compiled with peak options.

Table 1. Number of benchmark launches from SPEC~CPU packages that use more than 10% of time to wait for data

Set	Part of time	Number of launches	Set	Part of time	Number of launches
1995	10...15%	12	**2000**	10...15%	6
	15...20%	4		15...20%	6
	20...25%	0		20...25%	6
	25...30%	1		25...30%	1
	≥ 30%	0		≥ 30%	6
	Total in set	37		Total in set	44
f2006	10...15%	4	**i2006**	10...15%	3
	15...20%	1		15...20%	1
	20...25%	1		20...25%	3
	25...30%	1		25...30%	3
	≥ 30%	2		≥ 30%	12

	Total in set	20		Total in set	35
f2017	10...15%	1	**i2017**	10...15%	1
	15...20%	2		15...20%	2
	20...25%	3		20...25%	0
	25...30%	0		25...30%	3
	≥ 30%	1		≥ 30%	6
	Total in set	16		Total in set	20
All	10...15%	27			
	15...20%	16			
	20...25%	13			
	25...30%	9			
	≥ 30%	27			
	Total in set	172			

The table shows that more than 10% of the execution time is spent waiting for data from memory in 92 from 172 launches, which is more than a half.

Some of this spent time is due to inefficient use of cache memory. Mainly, these inefficiencies are:

1) loading unnecessary for further work data into the cache, which fact is a violation of spatial locality;
2) conflicts between different data chunks due to hitting the same cache set.

For example, it was found during our previous work that it is possible to reduce the number of cache misses with the help of optimization called Structure Splitting [1]. This optimization improves the spatial locality of the program in some cases. Such CPU pipeline stalls number decrease and consequent execution speeding up are shown in the Table 2.

Table 2. CPU pipeline stalls number decrease and following program execution speeding up

Benchmark	**SPEC CPU package**	**CPU pipeline stalls number decrease**	**Speed-up**	
181.mcf	2000	27%	26%	
429.mcf	2006	19%	13%	

From this example, it can be seen that at least some of the losses due to waiting for data can be removed by data layout transformations improving spatial locality. These transformations require unified analysis for an effective combination.

3. Problem statement

Thus, we need to:

1) Theoretically analyze cache memory usage by programs and develop a method of solving the problem of minimizing time losses based on this theoretical analysis.
2) Based on theoretical results, make applicable automatic analysis in the LCC compiler for the Elbrus and SPARC ISA.
3) Implement in the same compiler a set of data layout transformations, which transform data layout of a program based on the analysis results.

In this case, it is necessary to take into account some restrictions arising from the fact that the implementation is planned in the form of compiler optimizations:

1) Various data structures need to be handled correctly. Particularly, they are:
   a) Arrays, structures and their combinations.

   b) Various data structures that use pointers to other elements internally and allocate memory for new elements via *malloc* and similar memory allocation functions. For example, lists and trees.
4) We need to handle data structures altogether, as their transformations may conflict with each other. Therefore, it is necessary to analytically process not only regular access to memory but also random access.
5) Analysis and transformations must be static (in the compiler) but can be supported with runtime libraries and special profiling, but not memory access trace.
6) Developed analysis and transformations must correctly work in modular build mode.

4. Related work

Several works on related topics have already been written, but each of them does not solve assigned tasks entirely due to different reasons.

Chris Lattner proposed automatic Data Structure Analysis to detect data structures whose elements are allocated on the heap in his thesis «Macroscopic Data Structure Analysis and Optimization» [2]. Using the results of this analysis, he proposed a compiler optimization called Automatic Pool Allocation with runtime support, designed to group the elements of such data structures in specific regions of the heap, which improves the spatial and, in some cases, temporal locality of the program. In addition, he offered several optimizations for code already optimized in this way.

Unfortunately, there is no explicit cache memory usage analysis in Lattner's work.

Christopher Haine in his thesis «Kernel optimization by layout restructuring» [3] offered an analyzer, which detects accessing memory regularly simple data structures like structures and arrays and proposes layout transformations using heuristics data. This analysis is separated from the compiler. In addition, this analyzer provides user with information about the complexities of code vectorization. For our purposes, this work is not suitable since there is no explicit cache memory usage analysis.

Mostafa Hagog and Caroline Tice in their article «Cache Aware Data Layout Reorganization Optimization in GCC» [4] proposed several improving spatial locality optimizations of structures and arrays of structures: Structure Peeling, Structure Splitting, and Field Reordering. These optimizations were later implemented in the GCC compiler. Although the authors limited themselves to working with structures, they implemented an analysis handling every structure access, not just regular access. During optimization, particular Field Reference Graphs are built for each analyzed structure for each procedure. Field Reference Graph (FRG) is an analogue of a control-flow graph, where nodes contain operations accessing fields of the analyzed structure and arcs contain information about the amount of data loaded into the cache between nodes. In fact, this is an implicit analysis of cache memory usage. Further, after processing, this information is used in heuristics to apply the specified optimizations and reduce the computational complexity of further algorithms.

This approach can potentially be used for explicit cache memory usage analysis, provided it is generalized for working on all program data in all procedures.

Ghosh et al. [5] and Fraguela et al. [6] suggested more explicit techniques for cache memory usage analysis for regular access cases.

Ghosh et al. [5] proposed to compose and solve systems of linear Diophantine equations to estimate the number of cache misses for each cycle. They implemented this algorithm in the SUIF compiler and implemented the choice of padding size in the Array Padding optimization as an example. However, they did not implement an automatic solution of systems in parametric form - only a particular solution for Array Padding. In addition, this approach was created only for regular memory access.

An alternative approach was suggested by Fraguela et al. [6] for regular memory access. It was improved by Andrade in [7] thesis for some cases of irregular memory access: regular access under condition and access to an array, where the indices are read from another array. This approach is based on estimating the probability of cache misses in each analyzable cycle using Probabilistic Miss Equations (PME) generated from regular access characteristics and cache memory characteristics. To do this, for each processed access in the loop, a partial Probabilistic Miss Equation is built, and then they are combined into a complete equation for the loop or loop nest. This complete equation gives an estimation of cache misses amount. In addition, they did not offer any solution to the problem of minimizing cache misses amount and did not handle random memory access. Thus, the PME approach can potentially be applied for explicit cache memory usage analysis, provided the analysis is generalized for working for all irregular memory access.

Data layout transformations were described in many papers. Particularly, a small catalogue of such transformations was created in the article [8]. Following transformations are listed in this article:

1) Array Padding – adding padding between arrays to reduce number of conflicts between arrays;
2) Array Merging – element-wise arrays merging;
3) Array Transpose – changing dimensions' order of an array by analogy with transposing a matrix.

In addition to these, in the above-mentioned article [4] and thesis [2] some other transformations were described:

1) Structure Peeling – splitting an array of structures element by element into several arrays;
2) Structure Splitting – splitting an array of structures element by element into several arrays and addition of links between the elements corresponding to the initial element;
3) Field Reordering – changing order of fields inside the structure;
4) Automatic Pool Allocation – replacing memory allocation for data structure elements in the heap with memory allocation in a specific pool.

5. Proposal

Firstly, it is proposed to investigate and compare following methods for cache memory usage analysis:

1) the method described in [4] using FRG graphs, generalized for working with all program data in all procedures;
2) the method described in [6, 7] using the Probabilistic Miss Equations, generalized for the case of random access.

We propose to choose one method for cache memory usage analysis that is more suitable for implementation in the compiler. The selection criterion is the accuracy of the estimation of cache misses amount. Another selection criterion is analysis time.

Further, we propose to develop an analytical or another compiler-applicable method for solving the problem of minimizing the obtained estimation of the cache misses amount using data layout transformations. This problem is a discrete optimization problem, in which the objective function is the dependence of the cache misses amount on the applied data layout transformations, and a countable set of feasible solutions is the data layout transformations.

Finally, based on the developed analysis method and the method for solving the problem of minimizing the cache misses amount, it is proposed to implement automatic analysis in the compiler that controls a set of data layout transformations. Also, we will need to implement missing transformations.

5.1 Generalizing FRG analysis

This method should be generalized for working on all program data in all procedures and provide an estimation of cache misses amount. To do this, based on the FRG graph for structures, we need to make a generalized graph for structures, arrays, their combinations and other data structures. Such graphs need to be created for each program object. Let us call such graphs Object Reference Graph – ORG. In addition, we need to build a general RGP (Reference Graph in Procedure) graph consisting of all memory accesses in the procedure and including profile information. So any ORG graph in a procedure contains a subset of RGP nodes; therefore, using RGP, one can estimate the probabilities of transitions through various ORG arcs and cache memory usage characteristics between ORG nodes. In addition, RGP is required to analyze conflicts between different data structures.

It is required to determine the probability of a particular cache line being evicted from the cache memory to estimate the probability of a cache miss in each ORG node. Since the probability of preempting a particular cache line depends on the amount of memory loaded into the cache in the general case in a complex way, it is better to store on the arcs of ORG graphs, not the amount of memory loaded into the cache, but the probability of preempting a particular cache line.

To estimate the probabilities, one must know in which memory regions the memory addressed by each pointer is located and the size of these memory regions. To obtain this information, we need to use pointer analysis and a particular version of the profile, which collects data on the size of the allocated memory.

5.2 Generalizing PME analysis

To use this method, we need to generalize it for processing irregular memory access.

For this, we need to:

1) Create a way to calculate cache misses' probability for random access.
2) Generalize PME to those cases of near-regular access where it is possible to estimate cache misses amount more accurately than using a random access model.
3) Combine PME for regular access and ones for random access.
4) Use the developed techniques for estimating cache misses amount for the entire code, not just for loops.

To estimate the probabilities, one must know in which memory regions the memory addressed by each pointer is located and the size of these memory regions. To obtain this information, we need to use pointer analysis and a particular version of the profile, which collects data on the size of the allocated memory.

6. Current progress

In the work [1] we described the particular version of data layout transformation called Structure Splitting, which we had implemented in the LCC compiler for the Elbrus and SPARC architectures. In addition, in this compiler Structure Peeling, Array Transpose, Array Linearization, and Array Padding have already been implemented.

6.1 Cache miss probability for random access

To generalize the PME-based analysis, a method was created for calculating the cache misses probability for random access. It is supposed that the memory region is known for this access, but the address of the region beginning is unknown. PME will be merged with this method.

The method is based on determining cache state transformations for each memory access operation. For this, the operations are traversed sequentially in the basic blocks of the procedure, and the transformations on the code blocks are combined according to the probabilities in the profiled control-flow graph. Any operation of the procedure is traversed once for random access case. For any other case number of single operation traversals must be $O(1)$ due to the analysis applicability requirement.

The cache state notation for the general case of regular and random access has not been determined yet, but the following notation has been chosen for the random access model: matrix $\mathbf{P}$ composed of N vectors $\mathbf{P}_i$ corresponding to N memory regions. Each vector has $S+1$ size, where S is the number of cache lines in the cache. The element of the matrix $\mathbf{P}_{ij}$ is the probability that exactly j lines corresponding to the area i are stored in the cache memory at the moment.

An example of the chosen cache state notation for three memory regions called a_i, where $i = 1..3$, is shown in Table 3. In the shown state it is implied that region a_1 has no lines in cache with 100% probability. Also, probability of a_2 taking all lines of cache is 90% and probability of a_3 taking one line and a_2 taking all other lines is 10%.

Table 3. Chosen cache state notation example $\boldsymbol{P_{ij}}$ for three memory regions called a_i, $i = 1..3$

$\boldsymbol{j}$	$\mathbf{a_1}$	$\mathbf{a_2}$	$\mathbf{a_3}$
$\boldsymbol{S}$	0%	90%	0%
$\boldsymbol{S-1}$	0%	10%	0%
…	…	…	…
1	0%	0%	10%
0	0%	100%	90%

Let us introduce for each operation or code section c an operator for changing the state T^{c}. If there was state P^b before executing c, then state P^a after executing c is: $\mathrm{P}^a = \mathrm{T}^{\mathrm{c}}\mathrm{P}^b$. We require the following properties for the operator:

1) For a code section c, consisting of K consecutive code sections or operations $\mathrm{c}_1,\dots,\mathrm{c}_K$, the operator is a composition of operators for parts of the section: $\mathrm{T}^{\mathrm{c}} = \mathrm{T}^{\mathrm{c}_K} \dots \mathrm{T}^{\mathrm{c}_1}$.
2) For a code section consisting of K alternative code sections or operations $c_1,\dots,c_K$ with probabilities of passing through them $p_1,\dots,p_K$ (for example, `if` block and `else` block), with $\sum_{j=1}^{K} p_j = 1$, the operator is a linear combination of operators for parts of the section: $\mathrm{T}^{\mathrm{c}} = \sum_{j=1}^{K} p_j \mathrm{T}^{\mathrm{c}_j}$.
3) Similarly, if during the execution of one operation op of the K different state changes $\mathrm{T}_1^{\mathrm{op}} \dots \mathrm{T}_K^{\mathrm{op}}$ may occur with probabilities $p_1,\dots,p_K$, and $\sum_{j=1}^{K} p_j = 1$, the operator is a linear combination of their operators: $\mathrm{T}^{op} = \sum_{j=1}^{K} p_j \mathrm{T}_j^{\mathrm{op}}$.

For the chosen matrix cache state notation, we also introduce an element-wise product $\circ$ of the operator and coefficients.

Let us consider one memory access operation. It can cause three different outcomes:

1) Cache hit. In this case, cache state in the selected notation is not changed.
2) Cache miss with a conflict in the memory region. In this case, cache state in the selected notation does not change since it only stores the probabilities of having a certain amount.
3) Cache miss with a conflict with another memory region. A new line is loaded into the cache for the memory region the operation is working with. For one of the other memory regions, the line is evicted from the cache.

Thus, change in cache state for a single operation for a specific memory region can consist only in loading a new cache line for memory region, deleting cache line from the cache for memory region, or no changes for memory region. For such changes we introduce operators for the movement of cache state in selected notation:

1) M^+ – moves the matrix values up by 1: if $\mathrm{P}^a = \mathrm{M}^+\mathrm{P}^b$, then

$$\forall i \in 1 \dots N \mapsto \begin{cases} \mathrm{P}_{ij}^a = \mathrm{P}_{i(j-1)}^b, j = 0, S-1 \\ \mathrm{P}_{iS}^a = \mathrm{P}_{iS}^b + \mathrm{P}_{i(S-1)}^b \\ \mathrm{P}_{i0}^a = 0 \end{cases}$$

2) M^- – moves the matrix values down by 1: if $\mathrm{P}^a = \mathrm{M}^-\mathrm{P}^b$, then

$$\forall i \in 1 \dots N \mapsto \begin{cases} \mathrm{P}_{ij}^a = \mathrm{P}_{i(j+1)}^b, j = 0, S-1 \\ \mathrm{P}_{i0}^a = \mathrm{P}_{i0}^b + \mathrm{P}_{i1}^b \\ \mathrm{P}_{iS}^a = 0 \end{cases}$$

3) M^0 – does not move matrix values.

Writing down cache state change operator T^{op} for operation, working with the memory region i, we get:

$$\mathrm{T}^{\mathrm{op}} = \rho_{i+} \circ \mathrm{M}^+ + \rho_{i0} \circ \mathrm{M}^0 + \rho_{i-} \circ \mathrm{M}^-$$

where:

1) ρ_{i+} – matrix of coefficients for loading a new line of i into the cache; this matrix consists of a nonzero column for the i -th vector, other coefficients are equal to zero;
2) ρ_{i0} – matrix of coefficients for saving cache state as it is;
3) ρ_{i-} – matrix of coefficients for evicting a line from the cache when loading a new line of the i area into the cache; this matrix consists of nonzero columns for all vectors except the i -th.

An example of applying operator T^{op} to cache state example above is shown in Table 4. Operation op accesses memory region a_1 so one line of a_1 is loaded into cache and one line of a_2 or a_3 is evicted from the cache.

Table 3. Result of applying operator T^{op} to cache state from Table 3 when op works with memory region a_1 ($i = 1$)

$\boldsymbol{j}$	$\mathbf{a_1}$	$\mathbf{a_2}$	$\mathbf{a_3}$
$\boldsymbol{S}$	0%	0%	0%
$\boldsymbol{S-1}$	0%	90%+10%$\cdot\frac{1}{S}$	0%
$\boldsymbol{S-2}$	0%	10%$\cdot\frac{S-1}{S}$	0%
…	…	…	…
1	100%	0%	10%$\cdot\frac{S-1}{S}$
0	0%	0%	90%+10%$\cdot\frac{1}{S}$

7. Conclusion

Publications analysis showed that there is no unified solution to the problem of improving cache usage of compiled programs. In this paper, we propose a research approach, which can lead to a solution to this problem in compilers.

Список литературы / References

[1] В.Е. Шампаров, А.Л. Маркин. Механизм оптимизации Structure Splitting в составе компилятора для микропроцессоров Эльбрус. Программная инженерия, том 12, no. 2, 2021 г., стр. 82-88 / V. E. Shamparov and A. L. Markin. Structure splitting for elbrus processor compiler. Software Engineering, vol. 12, no. 2, 2021, pp. 82-88 (in Russian).

[2] C. Lattner. Macroscopic Data Structure Analysis and Optimization. Ph.D. dissertation, Computer Science Dept., University of Illinois at Urbana-Champaign, 2005, 225 p.

[3] C. Haine. Estimation d'efficacité et restructuration automatisées de noyaux de calcul. (Kernel optimization by layout restructuring). Ph.D. dissertation, University of Bordeaux, France, 2017, 114 p.

[4] M. Hagog and C. Tice. Cache aware data layout reorganization optimization in gcc. In Proc. of the GCC Developers' Summit, 2005, pp. 69-92.

[5] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. ACM Transactions on Programming Languages and Systems, vol. 21, no. 4, 1999, pp. 703-746.

[6] B.B. Fraguela, R. Doallo, and E.L. Zapata. Probabilistic miss equations: Evaluating memory hierarchy performance. IEEE Transactions on Computers, vol. 52, no. 3, 2003, pp. 321-336

[7] D. Andrade. Systematic analysis of the cache behavior of irregular codes. Ph.D. dissertation, Department of Electronics and Systems, University of A Coruña, Spain, 2007, 165 p.

[8] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. Lecture Notes in Computer Science, vol, 2625, 2003, pp. 213-232.

Информация об авторах / Information about authors

Виктор Евгеньевич ШАМПАРОВ, аспирант МФТИ, программист АО «МЦСТ». Научные интересы: компиляторы, оптимизация кода, VLIW-архитектура.

Viktor SHAMPAROV, PhD student at MIPT, software engineer at MCST. Research interests: compilers, code optimization, VLIW architecture.

Мурад Искендер-оглы НЕЙМАН-ЗАДЕ, к.ф.-м.н., доцент. Научные интересы: компиляторы, оптимизация кода, VLIW-архитектура.

Murad NEIMAN-ZADE, PhD in mathematics, associated professor. Research interests: compilers, code optimization, VLIW architecture.