Accurate data distribution into blocks may boost cache performance

Dan Nam Truong
dtruong@irisa.fr

François Bodin
bodin@irisa.fr

André Seznec
seznec@irisa.fr

IRISA/INRIA
Campus de Beaulieu
35042 Rennes, Cedex
France

Abstract

Applications often under-utilize cache space, generating unnecessarily high cache miss ratios.

Data distribution is a software technique which could improve cache miss rates for any types of application. There is a great potential to exploit: data distribution can reduce capacity misses as well as conflict misses.

Keywords: *Cache, Non-scientific programs, Locality, Data layout, Distribution miss, Reference correlation, Miss model, Software optimization.*

1 Introduction

Modern applications tend to under-exploit available cache space. Burger showed that up to 90% of the cache space can be mis-used [BGK95]. Current L1 caches are already quite large, and exhibit low miss ratios when used properly.

Applications usually mis-use cache space because the organisation of the memory hierarchy is not taken into consideration when a data layout is chosen. Compilers layout data based on data management considerations. They organize memory space into heap, stack, or global allocation zones. Programmers organize data based on program specific constraints. Programming languages do not even give them any tools to exploit data locality. Some provide data layout paradigms but only as tools for parallelization.

Caches capture some locality, because compilers organizational layout constraints and programmers' data organization coincide with data usage (for example, stack reuse generates temporal locality). However, as data sets used become larger, these constraints become weaker. Much more locality can be extracted if the data layout is made more precisely.

To recapture spatial locality we propose to distribute data elements in memory. This data layout technique simply consists in changing the relative addresses of data elements to better exploit cache line space. Improved temporal locality comes as a byproduct, and further improves cache performance.

Data distribution is complementary to the other software data layout techniques developped for scientific codes, such as array padding and tiling. Since it impacts directly on data addresses, it can be applied to general purpose, non-scientific codes too.

In the first section, we introduce distribution misses, to show that redistributing data in memory can reduce capacity misses. In the second section we show that it is possible to find which elements should be close together by correlating their reference patterns. The third section shows a profile based scalar data layout heuristic to demonstrate that data distributions can be done automatically. Our conclusion is that data distribution is a necessary step to remove capacity and conflict misses. This study shows that distribution techniques have a big potential and should be developped for all common data types.

2 Distribution misses

Software cache optimization techniques can remove capacity misses. To show this and evaluate the potential for improvment, we split capacity misses into fundamental and distribution misses.

We use Sugumar's definitions for capacity and conflict misses [Sug93]. To compute capacity misses, he uses a perfect cache instead of a fully-associative LRU cache. A perfect cache is a fully-associative cache with an optimal replacement policy. Belady described such an optimal block replacement policy, Min [Bel66]. It uses an oracle to predict which block is the farthest one referenced into the future and replaces it.

With a given data layout, a cache cannot beat the capacity miss ratio. However, at the software level, we can change the relative addresses of data elements in memory. Therefore reference patterns to cache blocks changes, since blocks may hold a different set of data elements: the capacity miss ratio is modified.

There is at least one optimal data distribution in memory for which the capacity miss ratio is the lowest. We call this capacity miss ratio the **fundamental miss ratio**. With a suboptimal layout, there are extra capacity misses. We call them **distribution misses**.

We cannot find an optimal data distribution, but we can lower-bound the fundamental miss ratio: A perfect cache with 1 byte per line generates only fundamental misses. We call $M1$ its miss ratio. The fundamental miss ratio of a cache with L bytes per lines cannot be lower than $M1/L$, because with fewer larger lines, there are fewer replacement choices open.

We approximate the fundamental miss ratio to $M1/L$ and the distribution miss ratio to $M - M1/L$, M is the capacity miss ratio of the cache studied.

With an optimal data distribution and a perfect cache, we can remove all but fundamental misses. Our aim is to remove distribution misses by choosing which data-elements must go together in a block. Furthermore, to approach perfect-cache performance, we also remove conflict misses by distributing blocks in memory to prevent cache interferences.

As an example, we used a 40 million reference trace of go (SpecInt 95). We ran go 10 10, compiled with gcc 2.7.2, -O optimization, and fixplaylevel=20. We tested other programs, and go results are typical of non-scientific applications. The direct-mapped cache generates 6.47% misses, but only 0.59% are fundamental misses, as shown in top rows of table 1. The potential for improvement is therefore over 90%. In the next sections, we show how we can exploit this potential.

3 Correlating data elements

To develop data distribution heuristics, we want to evaluate the spatial locality of a data distribution but without having to measure the miss ratio. This can be done by analysing data-elements reference patterns over the whole program execution.

To improve spatial locality, we regroup data element which are most often referenced together, and separate those often referenced independently. To do this, we introduce the notion of correlation between data elements. With this approach, we find a data layout directly without having to try all layouts.

Table 1: Data distribution reduces capacity (distribution) and conflict misses of the GO benchmark.

Cache	DM	2-SA	4-SA
Fundamental	0.59%	0.59%	0.59%
Distribution	1.18%	1.18%	1.18%
Capacity	1.77%	1.77%	1.77%
Conflict	4.70%	2.12%	1.02%
Total **original**	**6.47%**	**3.89%**	**2.79%**
Fundamental	0.59%	0.59%	0.59%
Distribution	0.37%	0.37%	0.37%
Capacity	0.96%	0.96%	0.96%
Conflict	0.43%	0.38%	0.38%
Total **improved**	**1.39%**	**1.34%**	**1.34%**

Elements which have references far appart will be less likely to beneficiate from spatial locality. Therefore we will only consider close references.

Definition 1 *We define a* **time reference window** *W as a time interval of size $||W||$. Two references are said to be* **close** *in time if the time interval between them is smaller than $||W||$.*

The activity of a data element determines when it is heavily referenced and should be kept in the cache, and when it is not referenced and should be out.

Definition 2 *A portion X of memory is* **active** *at a time t, if t is between two references*[1] *to X that are not separated by a time interval greater than $||W||$.*
$A^W(X,t) = 1$ *if* $\exists(t', t") \mid (t \in [t', t"]) \wedge (Ref(X,t') = 1, Ref(X,t") = 1) \wedge (|t" - t'| \leq ||W||)$, *and* 0 *otherwise.*

We must regroup elements which provide the highest spatial locality. These must be most often active together, and rarely active separately. We capture this in a global, time independant correlation metric. Correlation also measures temporal locality, it increases with reference frequency.

Definition 3 *The* **correlation** *$C^W()$ of two data-elements is the count of all references done while both are active minus all the references to each while the other element is inactive.*
$C^W(X,Y) = \sum_{t=0}^{t_{end}}((A^W(X,t) \times Ref(Y,t)) + (Ref(X,t) \times A^W(Y,t)) - ((1 - A^W(X,t)) \times Ref(Y,t)) - (Ref(X,t) \times (1 - A^W(Y,t)))$

An improved data distribution can be chosen by making sure that elements with highest correlations

[1]We note $Ref(X,t)$ the predicate that is 1 if X is referenced at time t and 0 otherwise.

fit together in the cache. Reference patterns can be obtained by using a profile, source code analysis or just programmer knowledge of program behaviour.

4 A scalar data distribution heuristic

To show the feasability of data distribution heuristics, we developped and experimented an automatic, profile based scalar data distribution heuristic.

Using correlation, the heuristic is straightforward:

- Compute and store A^W for each data-element during profiling.
- Redistribute data-elements with the highest C^W into the same blocks. We repeatedly make a block, and fill it with the best correlated elements untill all elements are assigned to a block.
- Distribute blocks with highest C^W into distinct cache lines. We repeatedly seek the block correlating most with all blocks already assigned to a line, and assign it to the line holding blocks correlating the least with it.
- Blocks are laid out in memory at any address mapping to the correct line.

Since our heuristic handles only scalars, we performed a layout optimization using a trace as input to ignore all data structuring constraints. All memory references are seen as references to scalars.

With such an approximation, we cannot fairly compare the original layout and the improved layout, but we can see that the compiler layout is far from optimal. We could improve it by 79%, from 6.47% misses downto 1.39%, as shown table 1. Data distribution removes 46% of the capacity misses. It also removes 91% of the direct-mapped cache conflict misses. As a result, the 1.39% improved miss ratio of the direct mapped cache is even lower than the original layout's 1.77% capacity miss ratio. That means that compilers under exploit data locality: even with a fully-associative cache, the original compiler-generated layout cannot outperform the improved data-distribution layout with a simple direct-mapped cache.

5 Conclusion

Data distribution consists in changing the address of data in memory to improve spatial locality. Since it modifies directly the address of data in memory, it can be used on complex data structures other than arrays.

We have split capacity misses into fundamental and distribution misses. Distribution misses can be eliminated with an optimal data distribution. For caches relying on address mapping, data distribution can also remove conflict misses. Only few misses are actually fundamental misses, so there is a great potential available to reduce the cache miss ratios. For the go benchmark, up to 90% of the misses can be removed.

Because data distribution may seem untractable, we provide theoritical tools to make distribution heuristics. We define the correlation between data elements. Data distribution must allow correlated elements to coexist in the cache. We tried a sample heuristic and used it to improve go miss ratio by 79% for a direct-mapped cache.

We use profiling to capture reference patterns, but other approaches like source code analysis can be used. Our heuristic works only for scalars, and data distribution should be adapted to all common data types. Readers can find more details in [TBS96].

References

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM System Journal*, page 24, 1966.

[BGK95] D. C. Burger, J. R. Goodman, and A. Kagi. The declining effectiveness of dynamic caching for general purpose multiprocessor. Technical report, University of Winsconsin, 1261, 1995.

[Sug93] R. A. Sugumar. *Multi Configuration Simulation Algorithms for the Evaluation of Computer Architecture Designs.* PhD thesis, University of Michigan, 1993.

[TBS96] D. Truong, F. Bodin, and A. Seznec. Accurate data distribution into blocks may boost cache performance. PI 1000, IRISA, 1996. ftp://ftp.irisa.fr/techreports/1996/PI-1000.ps.gz or http://www.irisa.fr/caps.