Analysis of the Spatial and Temporal Locality in Data Accesses

Jie Tao, Siegfried Schloissnig, and Wolfgang Karl

Institut für Technische Informatik
Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
{tao, schloissnig, karl}@ira.uka.de

Abstract. Cache optimization becomes increasingly important for achieving high computing performance, especially on current and future chip-multiprocessor (CMP) systems, which usually show a rather higher cache miss ratio than uni-processors. For such optimization, information about the access locality is needed in order to help the user in the tasks of data allocation, data transformation, and code transformation which are often used to enhance the utilization of cached data towards a better cache hit rate.

In this paper we demonstrate an analysis tool capable of detecting the spatial and temporal relationship between memory accesses and providing information, such as access pattern and access stride, which is required for applying some optimization techniques like address grouping, software prefetching, and code transformation. Based on the memory access trace generated by a code instrumentor, the analysis tool uses appropriate algorithms to detect repeated address sequences and the constant distance between accesses to the different elements of a data structure. This allows the users to pack data with spatial locality in the same cache block so that needed data can be loaded into the cache at the same time. In addition, the analysis tool computes the push back distance which shows how a cache miss can be avoided by reusing the data before replacement. This helps to reduce cache misses increasing therefore the temporal reusability of the working set.

1 Introduction

Due to the widen gap between memory and CPU speed caches were introduced into the computer systems for buffering reused data and for providing a low access latency that matches the processor speed. However, many applications still suffer from excessive data accesses in the main memory. This problem is more challenged on current and future chip-multiprocessor systems, since these machines show a higher cache miss ratio, which can be up to four folds of that on uni-processor systems, according to some research reports [7]. As a consequence, optimization with respect to cache locality has been regarded as a critical issue in raising the overall performance of modern processors.

Currently, this kind of optimization is primarily based on two approaches: compiler-level automatic optimization and user-level manual optimization. For the former, the optimization is performed during the compiling time through code transformation

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part II, LNCS 3992, pp. 502–509, 2006.

[10, 8, 9], array padding [1, 14], or both of them [4], while for the later optimization is done directly within the source code via manually rewriting the program code using the same techniques [5, 12]. In contrast to the compiler approach, the user-level optimization is more common due to its straight-forward manner. However, for both of them detailed information about the access pattern of applications is needed.

This information can be achieved using profiling tools, performance counters, simulation systems, or heuristic analysis models; however, we proposed and developed an analysis tool which performs the data analysis directly on memory accesses performed at the runtime during the execution of an application. Hence, this kind of analysis is more accurate and allows the acquisition of some information that can not be obtained using other approaches.

The memory accesses for this tool are provided by a code instrumentor, which inserts additional instructions in the assembly codes. This results in the generation of a memory trace containing all memory references, each of them shows the access address, access type, location in the source file, and the ID of the process issuing this access. Based on this trace, the analysis tool applies appropriate algorithms to detect the regularity among the references and give hits about optimization possibilities.

Overall, our tool makes the following contributions:

- Detecting access group, a group of neighboring accesses with different target address and occurring repeatedly. This helps to allocate these accesses in the way that they are held in the same cache block with a runtime effect of the access to the first data causing the load of other data into cache.
- Detecting access stride, a constant distance between accesses to elements of a data array. This gives the knowledge about which data will be required for further computing and directs the user or the system to prefetch data which is really needed. This also helps programmers to allocate data structures in the manner of presenting the spatial locality.
- Computing cache parameters, including reuse distance, set reuse distance, cache hit/miss, and push back distance. This shows whether an access is a cache hit or a cache miss and in case of misses a push back distance indicates how many steps an access must be pushed back in order to change the miss to a cache hit. This directs the users to use code transformation to enable an earlier issuing of an access before the data is evicted from the cache due to conflicts.
- Mapping between virtual address and data structures in source codes. The analysis tool is capable of providing all information described above in both virtual addresses and data structures. This enables the users to directly locate the optimization object in the source code.

The rest of the paper is organized as follows. Section 2 introduces the algorithms for detecting access patterns. This is followed by a detailed description of the functionality of this analysis tool, especially the implementation details, in Section 3. Section 4 shows some verification results based on applications from standard benchmark suites. The paper concludes in Section 5 with a short summary and some future directions.

2 Algorithms for Pattern Analysis

The main goal of our analysis tool is to provide information needed for applying cache optimization techniques like grouping and software prefetching. For this, we provide both address group and access stride, which is defined as:

Address Group. A repeated sequence of memory accesses with different target addresses and/or of position holders, where a position holder represents any address. The following is a simple example:

Assuming a memory access trace containing references with target address
100 200 300 400 ... 100 200 301 400 ... 100 200 302 400 ... 100 200 303 400 ...
Then two Address Groups exist ("-" is the position holder)
$< 100, 200 >$
$< 100, 200, -, 400 >$

Stride. A constant distance among neighboring accesses to different elements of a data array. A Stride is presented using the form $< start, distance, length >$, where start is the address of the first access, distance is the access stride, and length is the number of accesses with this stride. An example:

Assuming a memory access trace containing the following references
402 102 200 300 900 400 104 899 106 108 898 200 500 897 110
Then three Strides exist
$< 102, 2, 5 > (102, , , , , 104, , 106, 108, , , , , 110)$
$< 200, 100, 4 > (200, 300, , 400, , , , , , , 500)$
$< 900, -1, 4 > (900, , , 899, , , 898, , , 897)$

2.1 Detecting Address Groups

The work of detecting Address Groups is similar to the work of searching for repeated pattern from a DNA, RNA, or protein sequence in Bioinformatics. Because this is a hot research area, several algorithms have been proposed for efficient pattern discovery. Well-known examples are Sequence Alignment [3], Suffix-Tree [2], and Teiresias [13]. The Sequence Alignment algorithm is usually used to order two or more sequences to detect the maximal identity and similarity, while Suffix-Tree detects patterns and their positions from a sequence through building a tree with a single branch for each suffix of the sequence. However, due to the high requirement on memory space and overhead, neither of them is appropriate for detecting Address Groups from a large access trace.

Teiresias is another algorithm for pattern recognition. It first finds small patterns and then reconstructs them into larger ones. For small patterns two parameters are needed to specify both the maximal length of the pattern (L) and the number of letters (N), written in form $< L, N >$. The difference between L and N specifies the number of position holders. For example, the following are patterns with L=5 and N=3:

ABC
A-B-C
A–BC

For pattern discovery, the algorithm first performs a *scan/extension* phase for generating all small patterns of $< L, N >$. This is achieved by first building patterns of length 1 (one letter with a minimal number of occurrence) and then extending them to

patterns of length L with maximal N position holders starting and ending with a letter, i.e. no position holder at the first and the last position of a pattern.

For each detected pattern the Prefixes and Suffixes are then computed. A Prefix is a prefix of a pattern having to be ended with a letter. For instance, the possible Prefixes of the three patterns above are: AB, A-B, and A–B, where each Prefix must contain at least two letters and the last position must be a letter. Similarly, a Suffix is defined as a suffix of a pattern with a letter at the starting position. For example, the Suffixes of the three patterns are BC, B-C, and BC.

In the following, the algorithm performs a *convolution* phase to those pairs of detected small patterns with the feature of the Prefix of one pattern and the Suffix of the other being the same. Such pairs are combined to larger patterns. For example, from pattern DF-A-T and A-TSE pattern DF-A-TSE is generated. In addition, the *convolution* phase is also responsible for finding other patterns, like those overlapped. The following is an example:

pattern 1: abcdf

pattern 2: abcef

overlapped pattern: abc-f

In contrast with Sequence Alignment and Suffix-Tree, Teiresias has lower memory requirement. According to some researchers, Teiresia has also a good runtime behavior. Hence, we deploy this algorithm to detect Address Groups.

2.2 Detecting Strides

For detecting Strides we deploy an algorithm similar to that described in [6]. As shown in Figure 1, the algorithm uses a search window to record the difference between two references. The head line demonstrates the access addresses in a sample access trace, while the differences are presented in the columns corresponding to each address, with the first difference (to the previous access) in line 1, the second difference (to the second previous) in line 2, and so on. The number -297 in line 4 of the last column, for example, shows the difference between address 103 and 400.

Access trace:	200	300	501	923	400	102	881	500	103	.....
1		**100**	201	422	−523	−298	779	−381	−397	
2			301	623	−101	−821	481	398	−778	
3				723	**100**	−399	−42	**100**	1	
4					200	−198	380	−423	−297	
5						−98	581	−1	−820	
6							681	200	−398	
7								300	−197	
8									−97	

Fig. 1. Algorithm for detecting Strides

To discover a Stride, pairs with the same difference value are searched within the window. An example is difference 100 which is highlighted in the figure. The first position lies in line 1 of address 300 (difference between 300 and its direct left neighbor 200) and the second position is line 3 of address 400 (difference between 400 and its

3rd left neighbor 300). Then a stride $< 200, 100, 3 >$ can be concluded that describes an access stride beginning with address 200, of length 100, and repeating for three accesses. In the following a further difference 100 in line 3 of address 500 is observed. While this is the difference to address 400, the last address in the detected Stride, the Stride can be extended to $< 200, 100, 4 >$.

3 Implementation

For developing this analysis tool, we first need a memory trace that records all references performed at the runtime. We rely on a code instrumentor to generate the trace. This is followed by the implementation of the chosen algorithms for both Address Groups and Strides. In the next step we implement algorithms to determine whether an access is a hit or a miss and to give hints about how to transform a cache miss to a cache hit.

Memory Trace. The memory trace is generated by a code instrumentor, called Doctor. Doctor is originally developed as a part of Augmint [11], a multiprocessor simulation toolkit for Intel x86 architectures, and is used to augment assembly codes with instrumentation instructions that generate memory access events. For every memory reference, Doctor inserts code to pass its address, pid, and type to the simulation subsystem of Augmint. For this work, we slightly modified Doctor allowing to generate a trace file which stores all memory accesses performed by the application at runtime.

Teiresias and the Algorithm for Strides. The implementation of Teiresias follows actually the description of this algorithm in Section 2. First in the *scan/extension* phase small Address Groups $< L, N >$ are detected. The number of addresses and the length of an Address Group can be specified through command line parameters. Due to the large size of the memory trace, we slightly modified the functionality of this phase in order to reduce the execution time. In addition, we use a parameter K to specify the minimal occurrence of an address sequence and this reduces the number of small groups and thereby the time for generating larger groups in the *convolution* phase. As mentioned in Section 2, small patterns are combined in this phase and additionally overlapped patterns are detected and generated.

The implementation of the algorithm for Strides also follows the description in Section 2. First the search window is initialized and then for each reference the differences to all former accesses are calculated and stored in the window. It is clear that the size of the window is restricted and in case that the window is full the differences related to further accesses cover the columns in the window from left to right. Strides are generated and extended during the building and update of the search window.

Cache Parameters. In order to know if an access is a cache hit or a cache miss, our analysis tool models the architecture of the specified cache. The configuration information, such as cache size, block size, and the associativity, is delivered to the tool via command line parameters. According to the cache organization, reuse distance and set reuse distance are computed, where the former is the number of different addresses between two accesses to the observed address and the latter has the same meaning but considers only addresses lying in the same set. Using both cache parameters it is

possible to determine whether an access lands in cache or not. It is also possible to compute the Push Back Distance, which shows how many steps a miss access must be shifted in order to achieve a hit. Due to the high overhead, we do not compute the Push Back Distance for all variables in the program. Rather only user-specified variables are handled. For hit/miss estimation, however, all variables are covered in order to allow a combined analysis of the access behavior and the detection of optimization strategies.

Output. At the end of the analysis, our tool provides for each process three XML files, one for Address Groups, one for Strides, and the other for Cache Parameters. Each record in the Address Group file shows the addresses in the group and number of occurrence of this group. The Stride file gives all detected regular access stride together with the start address and the number of elements that hold this stride. In the Parameter file, the hit/miss and Push Back Distance, together with access address, variable, and source information, are demonstrated.

4 Verification with Standard Applications

We use several shared memory applications from the SPLASH-II benchmark suite [15] and a few small easy-to-understand codes to test this analysis tool for examining its feasibility in detecting access patterns and optimization possibilities.

Address Group. We first use the SPLASH-II applications to examine the ability of the tool for detecting frequently occurring Address Groups. Table 1 depicts the results with FFT (Fast Fourier Transformation), Barnes-Hut (solution of n-body problem), and Radix (integer radix sort).

Table 1. Results of Address Group detection with SPLASH-II applications

	Parameters			Number of patterns	
	L	W	K	process 1	process 2
FFT	3	5	500	46	31
	5	8	500	30	18
	5	15	5000	4	4
Barnes-Hut	3	5	500	111	587
	5	8	500	60	366
	5	15	5000	3	3
Radix	3	5	500	1492	1484
	5	15	5000	1255	48
	5	15	10000	21	22

It can be observed, and it is also clear, that for each application less Address Groups are detected with the increase of the number of addresses in the group (L), the group length (W), and the number of minimal occurrence (K). However, some applications still show a good behavior with large groups. For example, Radix holds more than 20 Address Groups of length 15 that repeat more than 10000 times. This indicates how well a program can be optimized with the grouping strategy. Applications, which hold

many smaller Address Groups, would benefit from caches with small cache blocks, while applications with many longer Address Groups benefit more from caches of large blocks.

It is also interested to see that some applications show an identical behavior among processes. For Barnes-Hut, for example, we have examined the corresponding Address Group of the same ID in the detected Groups of all processes and found that they map to each other and have the same structure: position holders at the same position and the same frequency of occurrence. We also detected that these mapping Groups target the same code region. This means that for these applications address grouping can be generally done without special handling for each process.

Access Stride. For verifying the profit capable of being achieved by the information about access strides we use a small code containing mainly the following loop:

$for(n1 = 0; n1 < DIM; n1++)$
  $for(n2 = 0; n2 < DIM; n2++)$
    $arr[n2 * DIM + n1] = 1;$

The result of analysis shows that all accesses to the array introduce a cache miss. In the XML file for detected Strides we found a set of records like:

.....
$< stride - id = 18\ start = 536883260\ step = 128\ length = 32 >$
$< stride - id = 18\ start = 536883188\ step = 128\ length = 32 >$
.....

We conclude that the misses are caused by the large stride between accesses to different elements of the array. With this stride the elements, which are loaded into cache together with the required element, can not be used before the cache block is evicted from cache due to conflict. With this observation we exchange the loops of n1 and n2 and this time only one Stride is reported:

$< stride - id = 18\ start = 536883140\ step = 4\ length = 1024 >$

The result shows significantly less cache misses with one miss every eight accesses. This is caused by the improvement in spatial locality, where data close to the accessed one is required in the next step of computing.

In summary, the experimental results demonstrate that our analysis tool can provide information which is needed and even necessary for conducting optimizations with respect to cache locality.

5 Conclusions

In this paper we introduce an analysis tool capable of providing information, like sequence of repeated access addresses and stride between accesses to elements of a data structure. It also exhibits the hit/miss behavior of each access and gives hints about how to transform a cache miss to a cache hit.

Currently we are developing a programming environment that both visualizes the information provided by the analysis tool and establishes a platform for analyzing, optimizing, compiling, and executing the applications. This environment also shows the influence of any optimization with the program codes. After this work we will starts with the optimization process first using benchmark applications and then realistic codes.

References

1. David F. Bacon, Jyh-Herng Chow, Dz ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness. In *Proceedings of CASCON'94 – Integrated Solutions*, pages 270–282, October 1994.
2. A. Chattaraj and L. Parida. An Inexact-suffix-tree-based Algorithm for Detecting Extensible Patterns. *Theoretical Computer Science*, 335(1):3–14, 2005.
3. A. Delcher, S. Kasil, R. Fleischmann, O White J. Peterson, and S. Salzberg. Alignment of Whole Genomes. *Nucelic Acids Research*, 27(11):2369–2376, 1999.
4. C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. *ACM SIGPLAN Notices*, 34(5):229–241, May 1999.
5. C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis*, 10:21–40, 2000.
6. T. Mohan et. al. Identifying and Exploiting Spatial Regularity in Data Memory References. In *Supercomputing 2003*, Nov. 2003.
7. S. Fung. Improving Cache Locality for Thread-Level Speculation. Master's thesis, University of Toronto, 2005.
8. Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. *ACM SIGPLAN Notices*, 33(11):228–239, November 1998.
9. Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Automated Cache Optimizations using CME Driven Diagnosis. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 316–326, May 2000.
10. N. Megiddo and V. Sarkar. Optimal Weighted Loop Fusion for Parallel Programs. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 282–291, New York, June 1997.
11. A-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 architectures. In *Proceedings of 1996 International Conference on Computer Design*, October 1996.
12. J. Park, M. Penner, and V. Prasanna. Optimizing Graph Algorithms for Improved Cache Performance. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pages 32–33, April 2002.
13. I. Rigoutsos and A. Floratos. Combinatorial Pattern Discovery in Biological Sequences: the TEIRESIAS Algorithm. *Bioinformatics*, 14(1):55–67, January 1998.
14. G. Rivera and C. W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 1998.
15. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.