Guided Prefetching Based on Runtime Access Patterns*

Jie Tao[1], Georges Kneip[2], and Wolfgang Karl[2]

[1] Steinbuch Center for Computing
Forschungszentrum Karlsruhe
Karlsruhe Institute of Technology, Germany
jie.tao@iwr.fzk.de

[2] Institut für Technische Informatik
Universität Karlsruhe (TH)
Karlsruhe Institute of Technology, Germany

Abstract. Cache prefetching is a basic technique for removing cache misses and the resulting access penalty. This work proposes a kind of guided prefetching which uses the access pattern of an application to prohibit from loading data which are not required. The access pattern is achieved with a data analyzer capable of discovering the affinity and regularity of data accesses. Initial results depict a performance improvement of up to 20%.

1 Introduction

During the last years, the gap between processor and memory speed has considerably widened. This indicates a significant performance degradation for applications with a large number of cache misses, because an access to the main memory takes hundreds of cycles while an access in the cache needs only several ones. According to [8], the SPEC2000 benchmarks running on a modern, high-performance microprocessors spend over half of the time stalling for loads that miss in the last level cache.

Cache misses can be caused by several reasons, for example, accessing a memory word at the first time, cache size smaller than that of the working set, or data evicted from the cache due to mapping conflict but used again. The resulted cache miss is correspondingly called compulsory miss, capacity miss, and conflict miss.

The most efficient way to reduce compulsory miss is prefetching [2]. This technique attempts to load data into the cache before it is requested. A key issue with this technique is to avoid prefetching data that is not required in a short time. Such inefficient prefetching may cause more cache misses because the prefetched data can directly, or indirectly, evict frequently reused data out of the cache. Moreover, prefetching consumes memory bandwidth. This additional

* This work was conducted as Dr. Tao was a staff member of the Institut für Technische Informatik.

memory traffic may degrade performance of multiprocessor systems, especially the emerging multi-core processors, where the main memory is shared across all on-chip processors.

In this work, we use a data analyzer to detect the affinity and regularity of data accesses and then apply this information to guide prefetching. The base of this analyzer is a memory reference trace achieved by instrumenting the assembly code during the compiling process. Based on this trace, the analyzer runs optimized analysis algorithms to discover repeated access chains and access strides. The former is a group of accesses that repeatedly occur together but target on different memory locations. The latter is a constant distance between accesses to successive elements of a data array. It is clear that both access chain and stride depict which data is requested.

These findings can be used to guide hardware prefetching. However, within this work we use them to perform software prefetching for the reason of easy implementation. We implemented a source-to-source compiler that takes an original program as input, and creates a new version of the same code but with prefetching instructions inserted. These instructions are formulated based on the output of the data analyzer, which shows both the access pattern and their occurrence in the source code. We studied this approach with several sample programs. Initial results depict a performance improvement of up to 20%.

The remainder of this paper is organized as following. Section 2 briefly describes existing research work in the field of cache prefetching. In Section 3 the analysis tool, together with the instrumentor for access trace, is introduced. This is followed by a detailed description of the precompiler in Section 4. In Section 5 first experimental results are presented. The paper concludes in Section 6 with a short summary and several future directions.

2 Related Work

Earlier techniques for cache prefetching are simple, where a prefetch is always issued to the next cache block [13] or several consecutive blocks [7]. This approach is still used by processor-associated prefetching, e.g. Pentium 4, for a straightforward implementation. However, due to its inefficiency current research work focuses on prefetching with access pattern like stride and linked memory references. Hardware-based prefetches often use a specific hardware component to trace repeated execution of a particular memory instruction and detect thereby reference strides, while software approaches usually rely on a compiler to analyze array references in program loops.

Baer and Chen [1] deployed a hardware function unit with the basic idea of keeping track of access patterns in a Reference Prediction Table (RPT). This RPT stores the last known strides. Prefetches are issued when the strides between the last three memory addresses of a memory reference instruction are the same.

Another hardware-based prefetching applies the Markov predictor [6] that remembers past sequences of cache misses. When a miss is found which matches a miss in the sequence, prefetches for the subsequent misses in the sequence are issued. The advantage of this approach is to be able to prefetch any sequence of memory references as long as it has been observed once.

The converse approach is software-based prefetching, in which access patterns are usually discovered by compilers. Luk and Mowry [9] use a compiler to detect linked memory references. Since addresses are not known at compile time, the compiler observes data structures containing an element that points to some other or the same data structure. Inagaki et al. [4] also uses profiling to guide compilers to perform stride prefetching. They proposed a profiling algorithm capable of detecting both inter- and intra-iteration stride patterns. This is a kind of partial interpretation technique and only dynamic compiler can perform such profiling. The profile information is gathered during the compiling process. The algorithm is evaluated with a Java just-in-time compiler and experimental results show an up to 25.1% speedup with standard benchmarks.

In summary, for detecting access patterns hardware approaches can take advantage of the runtime information but do not know the references in the future. In addition, they need specific support of hardware components and can hence not be commonly applied. Software prefetching is general and more accurate, however, the accuracy can only be achieved by compiler techniques in combination with runtime profiling.

We deploy a straightforward approach to achieve the access patterns. This is a separate analysis tool which has not to be integrated into modern sophisticated compilers. Since the analysis is based on memory references performed at the runtime, the accuracy of the access pattern can be guaranteed. More importantly, the tool can find all strides associated either with arrays or linked memory references. Additionally, this analysis tool delivers access sequences that are a number of references targeting different addresses but frequently issued successively. Clearly, this information also specifies the prefetch targets.

3 Pattern Acquisition

As mentioned, the information for our guided prefetching is directly acquired from the runtime references. For this, we developed a pattern analysis tool which is based on a code instrumentor.

3.1 Code Instrumentation for Access Trace

The instrumentor is an altered version of an existing one called *Doctor*. *Doctor* is originally developed as a part of Augmint [11], a multiprocessor simulation toolkit for Intel x86 architectures. It is designed to augment assembly codes with instrumentation instructions that generate memory access events. For every memory reference, *Doctor* inserts code to pass the accessed address, its size, and the issuing process to the simulation subsystem of Augmint.

We modified *Doctor* in order to achieve an individual instrumentor independent of the simulation environment. The main contribution is to remove from *Doctor* the function calls to Augmint. For example, *Doctor* relies on Augmint to acquire the process/thread identifier of the observed memory access. Within the modified version, we use the *pthread* library to obtain this parameter. Hence, the new instrumentor is capable of generating memory access traces for multi-threading programs based on the *pthread* library, like the OpenMP applications.

This instrumentation results in the recording of each memory reference at runtime. Besides the access address and the thread ID, we also store the position in the source code for each access. This information is needed to map the accesses to the source code and further to correctly insert prefetching instructions in it. For a realistic application, the trace file could be rather large. It is possible to use specific strategy, such as lossy tracing [10], to reduce the size of the trace file, however, for the accuracy of the analysis results, needed for the guided prefetching, a full trace is essential. In this case, we store the access records in a binary format and additionally develop a tool for transforming the binaries to the ASCII form if needed. This scheme also reduces the runtime overhead introduced by the instrumentation.

3.2 Affinity Analysis

For acquiring access patterns we developed an analysis tool [14] which currently detects both access chain and access stride. The former is a group of accesses that repeatedly occur together but target on different memory locations. An example is the successive access to the same component of different instances of a *struct*. It is clear that this information directly shows the next requested data which is actually the prefetching target. The latter is the stride between accesses to the elements of an array. Similarly, this information also indicates which data is next needed.

For detecting address chains the analysis tool deploys Teiresias [12], an algorithm often used for pattern recognition in Bioinformatics. For pattern discovery, the algorithm first performs a *scan/extension* phase for generating small patterns of pre-defined length and occurrence. It then combines those small patterns, with the feature that the prefix of one pattern is the suffix of the other, into larger ones. For example, from pattern DFCAPT and APTSE pattern DFCAPTSE is generated.

For detecting access strides the analyzer applies an algorithm similar to that described in [5]. This algorithm uses a search window to record the difference between each two references. References with the same difference are combined to form an access stride in the form of $< start\_address, stride, repeating\_times >$. For example, a stride $< 200, 100, 50 >$ indicates that starting with the address 200, memory locations with an address difference of 100, e.g. 300, 400, 500 etc., are also requested, and this pattern repeats for 50 times.

The detected access patterns are stored in an XML file. Together with each access chain or stride, the position in the source code is also recorded. This allows us to track the patterns in the program source and hence to find the location for inserting prefetching codes.

4 Automatic Prefetching

With the access pattern in hand, we now need a precompiler to generate an optimized source program with prefetching inserted. This is actually the task of a parser. We could directly deploy an existing parser of any compilers and build the prefetching on top of it, however, theses parsers are usually of the complexity that is not necessary for this work. In this case, we developed a simple parser specifically for the purpose of prefetching.

The key technique with the parser is to determine what and where to prefetch. For the former the access chains and strides clearly give the answer. However, the access patterns are provided in virtual addresses, while the parser works with variables. Therefore, an additional component was implemented for the goal of transforming the addresses in the pattern to the corresponding data structures in the program.

Previously, we relied on specific macros to register variables and generate a mapping table at the runtime, but currently we obtain the mapping table automatically by extracting the static data structure from the debugging information and instrumenting the *malloc* calls for dynamic ones.

Based on the variable names, the parser can build the prefetching instructions like $prefetch(p)$, where p is the variable to prefetch. Nevertheless, for arrays and linked references such as the *struct* data structure in C/C++ programs, it must additionally examine the corresponding code line to acquire other information about e.g. dimensions of an array.

The other important issue with prefetching is to decide the location where the prefetching instruction is inserted. This location directly determines the prefetching efficiency. For example, if the data is loaded into the cache too earlier, it can be evicted from the cache before it is used. On the other hand, if the prefetching is not issued early enough, the data is potentially not in cache as the processor requests it.

However, it is a quite tedious work to accurately compute the prefetching location. For simplicity, in this initial work we put the prefetching instruction several iterations before the associated. Users are asked to specify a prefetch distance, i.e. the expected number of iterations.

Knowing where to prefetch, the parser now has to identify the individual code line in the program. For this, the source code is scanned and key words that represent the start or end of a code line, like *if*, *for*, and *;*, are searched. Comments and directives for parallelization are removed before the scanning process for efficiency, but inserted into the resulted program again after this process.

Overall, based on the parser and other components, an optimized version of an application is generated with each prefetching instruction corresponding to a detected access pattern. This new version can be traditionally compiled and executed on the target architecture supporting software prefetching.

5 Initial Experimental Results

We used several applications to examine the effectiveness of this approach. In the following, we show two representative results, one achieved with the matrix multiplication code and the other with a realistic application.

The matrix multiplication program is used to examine how the prefetch distance influence the overall performance of an application. As mentioned in the previous section, we simply place the prefetching instruction several iterations, specified by the user, before the data is requested.

For this small code, our pattern analyzer found two strides, but no access chains. Theses strides correspond to both input matrices, A and B, one with an access stride of 1 (A) and the other of n (B) where n is the length of a matrix row.

Both matrices are prefetched based on the observed strides. We run the program on a Pentium 4 machine using different prefetch distances and the execution time is measured.

Figure 1 shows the experimental results, where the x-axis presents the various prefetch distance, while the y-axis depicts the improvement which is calculated with the execution time of the original code divided by that needed for running the code version with prefetching.

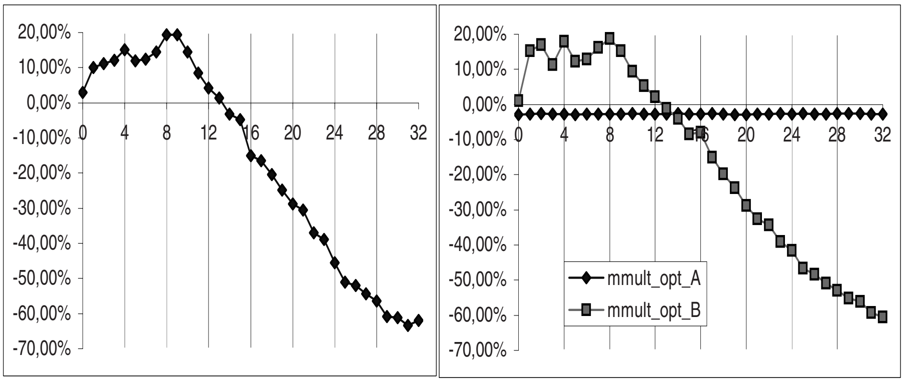

Fig. 1. Performance change with the prefetch distance (left: prefetching both A and B, right: prefetching A or B)

Observing the left diagram in the figure, it can be seen that the performance improvement arises up to a prefetch distance of 8 where the maximum of 20% is reached, and then decreases. This indicates that with a prefetch distance of smaller than 8 the data perhaps have not been loaded to the cache. For a larger prefetch distance, however, some prefetched data have been potentially evicted from the cache as the processor requests them. Hence, the best prefetch distance for this code is 8.

For a more detailed insight into the prefetching impact, we additionally measured the execution time with prefetching only one matrix, either A or B. The result is depicted in the right diagram of Figure 1.

As expected, matrix B performs well with prefetching and is therefore the contributor of the performance achieved with the overall program. However, prefetching A shows a performance lost of about 3%. It can also be observed that the prefetch distance does not influence the prefetching efficiency. This is caused by the fact that Pentium 4 performs hardware prefetching that preloads the successive data block of the accessed one. In this case, the requested data are prefetched by the hardware, indicating that our software prefetching is not necessary, but introduces overhead. For matrix B with a large access stride, nevertheless, the hardware prefetching is not efficient. In this case, our guided prefetching shows its power.

Now we observe a realistic application to evaluate the feasibility of this approach. The chosen application implements a 2D Discrete Wavelet Transform (DWT) algorithm, which is usually applied for image and video compression. This algorithm mainly contains two 1D DWT in both directions: horizontal filtering processes the rows followed by vertical filtering processing the columns. Our pattern analyzer detected strides with the input and out images in both filtering functions. The application is executed on an AMD and a XEON processor individually. Figure 2 depicts the experimental results, where the x-axis presents the image size while the y-axis shows the improvement in execution time to the original code.

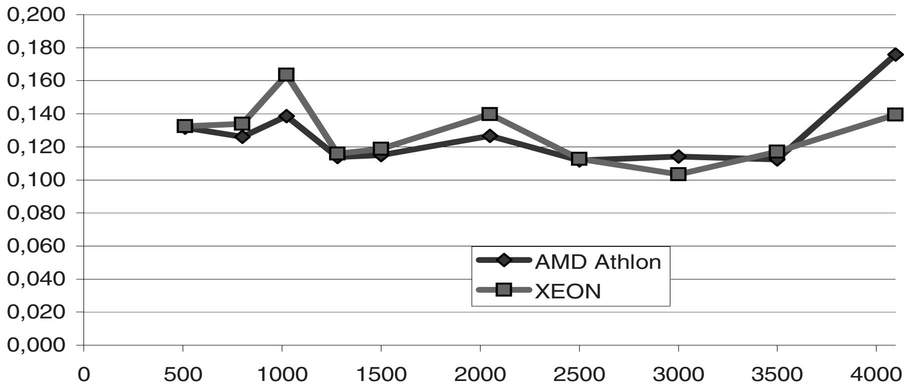

Fig. 2. Performance of the DWT program on different architectures

As can be seen, we achieved peak performance with each image size of a power of two. This is because the DWT algorithm has data locality problem with theses image sizes [3]. Hence, the optimization, specifically for improving the cache performance, shows its effectiveness. Overall, we achieved a performance gain of up to 18% with this application.

6 Conclusions

In this paper, we proposed a kind of guided prefetching for tackling cache problems. The prefetching is based on access patterns acquired by a self-developed analysis tool. Initial experimental results show a speedup of up to 20% in execution time.

However, in this prototypical implementation we have not deployed heuristics for computing the accurate prefetching position. This will be done in the next step of this research work. In addition, we intend to discover other access patterns that could be used for choosing the prefetch targets.

References

1. Baer, J.-L., Chen, T.-F.: Effective Hardware-Based Data Prefetching for High-Performance Processors. IEEE Transactions on Computers 44(5), 609–623 (1995)
2. Berg, S.G.: Cache prefetching. Technical Report UW-CSE 02-02-04, Department of Computer Science & Engineering, University of Washington (February 2004)
3. Chaver, D., Tenllado, C., Pinuel, L., Prieto, M., Tirado, F.: 2-D Wavelet Transform Enhancement on General-Purpose Microprocessors: Memory Hierarchy and SIMD Parallelism Exploitation. In: Proc. Int. Conf. on the High Performance Computing (December 2002)
4. Inagaki, T., et al.: Stride Prefetching by Dynamically Inspecting Objects. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 269–277 (June 2003)
5. Mohan, T., et al.: Identifying and Exploiting Spatial Regularity in Data Memory References. In: Supercomputing 2003 (November 2003)
6. Joseph, D., Grunwald, D.: Prefetching using markov predictors. IEEE Transactions on Computers 48(2), 121–133 (1999)
7. Jouppi, N.P.: Improving Direc-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In: Proceedings of the 17th Annual International Symposium on Computer Architectures, May 1990, pp. 364–373 (1990)
8. Lin, W., Reinhardt, S., Burger, D.: Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In: Proceedings of the 7th International symposium on High-Performance Computer Architecture, January 2001, pp. 301–312 (2001)
9. Luk, C., Mowry, T.C.: Compiler-Based Prefetching for Recursive Data Structures. In: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996, pp. 222–233 (1996)
10. Marathe, J., Mueller, F., de Supinski, B.: A Hybrid Hardware/Software Approach to Efficiently Determine Cache Coherence Bottlenecks. In: Proceedings of the International Conference on Supercomputing, June 2005, pp. 21–30 (2005)
11. Nguyen, A.-T., Michael, M., Sharma, A., Torrellas, J.: The augmint multiprocessor simulation toolkit for intel x86 architectures. In: Proceedings of 1996 International Conference on Computer Design (October 1996)
12. Rigoutsos, I., Floratos, A.: Combinatorial Pattern Discovery in Biological Sequences: the TEIRESIAS Algorithm. Bioinformatics 14(1), 55–67 (1998)
13. Smith, J.E.: Decoubled Access/Execute Computer Architectures. In: Proceedings of the 9th Annual International Symposium on Computer Architectures, July 1982, pp. 112–119 (1982)
14. Tao, J., Schloissnig, S., Karl, W.: Analysis of the Spatial and Temporal Locality in Data Accesses. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 502–509. Springer, Heidelberg (2006)