2014 Second International Symposium on Computing and Networking

Online Memory Access Pattern Analysis on an Application Profiling Tool

Yuki Matsubara*†, Yukinori Sato*†

*Research Center for Advanced Computing Infrastructure, JAIST, Japan.
†JST CREST, Japan
{yuuki-mt, yukinori}@jaist.ac.jp

***Abstract*—As memory subsystems have become complex in the state of the art system architectures, application program codes required to be optimized targeting to their deeper memory hierarchy for rewarding their performance. To support such optimizations, we are developing a memory access pattern analysis tool. In this paper, we present the methodology how we detect memory access patterns on-the-fly on an execution-driven application analysis tool called Exana. First, we implement an offline trace file based method using a Python script code and verify its functionalities. Then, in order to improve its analysis speed, the code is ported to C++ language programs and integrated in the Exana. We evaluate the time and memory usage for the analysis of each implementation. From the results, we confirmed our online implementation can process faster than the offline trace file based method.**

***Keywords*—Memory access pattern; Online profiling;**

I. Introduction

In recent years, due to the fact that improvements of memory access speed are not caught up with those of CPU, memory accesses often become a serious bottleneck during the execution of applications. Such a bottleneck is often called memory wall problem [1] and widely seen from the super-computing field to the mobile computing system. Especially, in the HPC field, the large amount of memory access is performed to process massive data. There are possibilities to improve memory access performance using the locality within hierarchical caches by loop blocking, data prefetching or adjusting memory alignment for efficient memory layout. Such optimizations for memory access is important to overcome the memory bottleneck. However, it is difficult for programmers to analyze and understand memory access patterns during the execution of their applications because memory access information for large applications easily become huge volume.

In order to support memory access optimizations, we focus on memory access pattern analysis from the actual memory access information obtained from the execution. If we could identify any regularities form memory access behaviors, the massive memory access information obtained at runtime will be detected as patterns and we can keep track of it throughout the execution. Because mostly the access patterns to the data in memory tend to be regular especially for applications in the HPC domain, we have much room to handle memory access information during whole the execution. While the most of access behaviors are regular, the rest of them show very complex and irregular behaviors. Therefore, we need to extract both of them.

In this paper, we present a methodology how we detect memory access patterns on-the-fly during the actual execution of application programs. Here, we obtain memory traces using our execution-driven application analysis tool called *Exana* [2]. Based on the obtained memory traces, we first design the MemPat, that is an algorithm for Memory Access Pattern Analysis, and implement it as an offline trace file based method using a Python script code. After we verify its functionalities, we find that we need to perform on-line analysis to increase the scalability for the actual problem size and the speed for analysis. Then, the code is ported to C++ language programs and integrated in the Exana and its speed for the analysis is evaluated.

II. Memory access pattern format

In this section, we present a memory access trace and the proposed format for memory access patterns used in this paper. A memory access trace is a sequence of accessed memory addresses and their related information. Here, we assume it includes each location of the memory access instructions in the executed binary file, the size of the memory access, and its functionality (read or write) in addition to the memory address accessed by the instruction.

For simplicity, we call an instruction with at least one memory access operations a memory access instruction in a later section. Our analysis is performed for each of memory access instructions and all memory accesses are organized into memory access patterns for each memory instruction.

Next, we present our proposed format for detecting memory access patterns used in this paper. In order to analyze memory accesses for each of memory access instructions, we classify it into the following four base patterns: Fix, Sequential, Stride, Sequential Stride.

The Fix represents a single memory access element that always refer the same address. The Sequential represents a single element of basic sequence of memory accesses without any strides or offsets. These two patterns are represented by $[D_{Size} \times N_{apr}]$, where D_{Size} is the size of each memory access and N_{apr} is the number of appearances of the memory accesses for each instruction. In the case that N_{apr} is greater than 1, it becomes a Sequential pattern; otherwise it becomes a Fix pattern.

978-1-4799-4152-0/14 $31.00 © 2014 IEEE
DOI 10.1109/CANDAR.2014.86

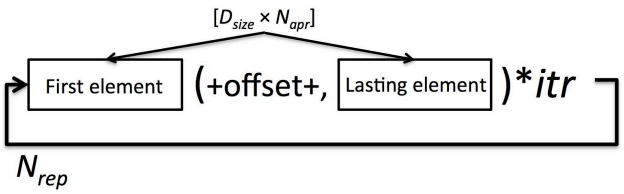

Fig. 1: A memory access pattern format for the Stride or Sequential Stride access.

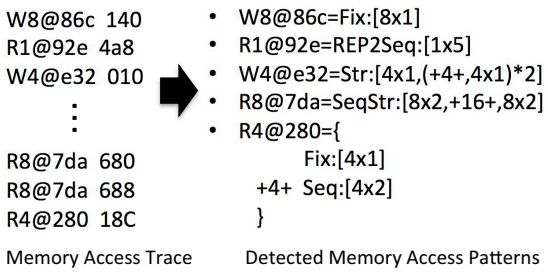

Fig. 2: An example of memory access traces and memory access patterns.

A Stride or a Sequential Stride pattern is constructed by accessed data elements and the offset between them using the format described. In Fig. 1, it is constructed by the first element and the lasting part "()" which includes offsets between elements "+" and "+" and lasting elements. Each of elements in that first and lasting part is composed of a Fix or a Sequential pattern. The number of iterations of the lasting part is indicated by *itr*. In this representation, multiple lasting parts are reduced to one part with the number of iterations. In the Stride or the Sequential Stride, when the total number of the elements is *n*, the number of offsets between elements actually appeared in the unrolled representation of elements becomes *n-1*. In order to detect simple cyclic patterns, we also attempt to use the representation the Repeat "N_{rep}". In the case that the same memory access pattern with the same address is detected continuously, we treat it as Repeat.

In addition to these 4 base patterns, we represent memory access patterns using complex patterns. A complex pattern is a combination of the 4 base patterns and their offsets, and used in the case that the memory access pattern of a memory instruction can't be represented just using four base patterns. Fig. 2 shows an example of obtained access patterns.

III. The Algorithm for Memory access pattern analysis

In this section, we present the MemPat, an algorithm for detecting memory access patterns from a chunk of memory trace. In order to be applied to both of the online and the offline pattern analysis, our algorithm attempts to sequentially seek a line of memory trace data only once on-the-fly. Our on-the-fly algorithm is composed of the following 3 phases; 1) the construction of the intermediate access patterns, 2) the update of lasting part of the intermediate pattern, and 3) the post processing for intermediate patterns.

At the beginning phase, the MemPat constructs the intermediate access pattern. The memory access trace is read sequentially from the top. After that, the MemPat detects a first element and an offset for constructing the intermediate pattern. At this time, the lasting element is set the same value with the first element because it is defined to become the same elements in our format. And then, *itr* count is set 0.

As the second phase, the update of lasting part in the intermediate pattern. When the MemPat detects an offset in the memory trace, it compares the offsets appeared before stored in the intermediate pattern. The lasting element is also compared in the same manner. If the offset and the lasting element match completely, *itr* count is counted up. On the other hand, if each do not match, the intermediate pattern is detected as a committed pattern. Then, the MemPat moves to detection of next memory access patterns. In this case, the final pattern becomes a complex pattern, which includes several committed patterns. And also, when the new committed pattern is detected, the MemPat compares it with previous one. If the start addresses and patterns of these in committed patterns are matched each other, these are treated as Repeat and "N_{rep}" is counted up.

For the final phase, we need to perform the post processing for intermediate patterns. If intermediate patterns are not committed, the MemPat detects deals with them as committed patterns.

IV. Implementations using the Exana

In our research group, we have been developing the *Exana* [2], which is an application profiling infrastructure for systems with deeply hierarchal memory subsystem. Using an already compiled executable binary code and if necessary its input data set as its input, we can profile various application behaviors. Since the Exana performs runtime analysis on-the-fly by the dynamic binary translation, it is also able to generate memory access trace and analyze it based on the proposed algorithm.

We implement the MemPat algorithm by the following two methods: the offline version using the generated memory trace offline as an input, and the online version written in C++ language and directly integrated in the Exana. First, we describe the details of the implementation by Python. The offline MemPat read the memory access trace sequentially from the top and the memory access information is fed to intermediate patterns corresponding to each of memory access instructions as discussed in the previous section. Since the offline MemPat is written in an interpreted language, it is nice for rapid application development and prototyping for verifying its functionalities. Although the offline MemPat is good for prototyping, we found from the evaluation discussed in the later section that it needs to be much more efficient implementation for analyzing production-level HPC applications.

The online MemPat implementation aims at analyzing production-level HPC applications on-the-fly. In order to improve its analysis speed, the MemPat component is ported to C++ language. At the same time, the MemPat algorithm is implemented as a part of the Exana and the runtime memory

TABLE I: The required time and the memory usage for each implementation

	Processing time [s]	Memory usage [MB]
Native	0.19	30.15
Exana (traceGen)	303.07	364.12
Offline MemPat	1139.59	166.22
Online MemPat	78.95	416.28

access information is directly fed to the MemPat component without generating memory trace file. Therefore, it can avoid creating a massive size of trace file on the Exana and reading it after it is generated.

V. Performance evaluation

A. Methodology

In this section, we evaluate our implementations of the MemPat algorithm in terms of the processing time and the memory usage. Here, we use the Himeno benchmark as an input program. The Himeno benchmark is a kernel program that solves the Poisson's equation by the Jacobi method [3]. Here, we use the static version of Himeno benchmark implemented in the C language. In the original Himeno Benchmark, scores of MFLOPS are measured based on the fact how much the main loop is executed during one minute. In this evaluation, we modify the number of iterations of main loops defined by the variable `nn` in the source code in order to adjust simulation time. Also, we use the GCC 4.4.7 compiler tool set with '-O3' option to generate the executable binary code for the application program.

Here, we show our evaluation environment. We use CentOS 6.4 on a typical x86 Linux server for the evaluation, which is equipped with Intel Xeon E5-2680 and 128GB memory. Also, we use the Exana tool set [2][4] that is implemented on the Pin tool set [5].

B. Results

Here, we evaluate both of the offline and the online MemPat implementations. First, we evaluate the offline and online MemPat in terms of the memory usage and the processing time of each program.

In Table I, we show the processing time for each execution. Here, we use the simulation size S and fix the variable `nn` to 35. We note that since the offline MemPat uses a trace file generated by the Exana, we need to perform the trace generation on the Exana and the memory access pattern generation by the offline MemPat implemented by Python, respectively. Therefore, we need to sum up the processing time for the Exana and the offline MemPat when we compare the total analysis time.

From the results, we find the online MemPat is the order of magnitude faster than the offline pattern analysis. This is because the offline MemPat analyzes a memory access trace file obtained from the Exana and it requires additional overheads compared with the online MemPat. Since the online MemPat implementation is inside the Exana, it does not generate a trace file. Therefore, it is faster than the processing time of the Exana part for the offline version. Moreover, the online MemPat feeds the memory access information from the Exana profiling component to the MemPat component within a single program that shares data structures. Therefore, the online MemPat can reduce the processing time dramatically compared with that of the Exana with trace file generation and the offline MemPat written in interpreted language.

Also, Table I shows those of the memory usages. Here, we observe the maximum memory size provided by operating system by checking the VmHWM size at `/proc/<pid>/status`. From the results, it is observed that the online MemPat increases its memory usage about 14 times compared with the native execution. For offline implementation, the Exana with trace generation increases it about 12 times, and the offline MemPat increases about 6 times, respectively. The results also show that the increase of the memory usage of the online MemPat from the Exana is reasonable and the additional usage is smaller than the total memory usage of the offline MemPat.

For the offline MemPat implementation, the generated trace file size is about 12.6 GB. While this evaluation is performed on a very small timestep, the actual execution needed to generate such files. To analyze production-level HPC applications, the trace file size will easily become bottleneck. Therefore, it can be concluded that the online MemPat implementation is much scalable especially in terms of processing speed and handling of memory trace data.

VI. Conclusions

In this paper, we have presented an algorithm for detecting memory access patterns from the actual memory access trace obtained at runtime. We have implemented this algorithm as an offline method using a memory trace file as an input and an online method implemented on our dynamic binary transformation system. We have evaluated the memory usage and the processing time of each implementation. From the results, we have confirmed that the processing time of the online version of memory access pattern analysis tool is about 18 times faster than the total time required for the offline version.

References

[1] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st conference on Computing frontiers*, ser. CF '04, 2004, pp. 162–167.

[2] Y. Sato, Y. Inoguchi, and T. Nakamura, "Identifying program loop nesting structures during execution of machine code," *IEICE Transaction on Information and Systems*, vol. E97-D, no. 9, pp. 2371–2385, Sep. 2014.

[3] "Himeno benchmark," http://accc.riken.jp/2145.htm.

[4] Y. Sato, Y. Inoguchi, and T. Nakamura, "Whole program data dependence profiling to unveil parallel regions in the dynamic execution," in *Proceedings of 2012 IEEE International Symposium on Workload Characterization (IISWC2012)*, Nov. 2012, pp. 69–80.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. tur Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. H. azelwood, "Pin: building customized program analysis tools with dynamic instrumen tation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 190–200.