Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 2058–2067

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

Gleipnir: A Memory Analysis Tool

Tomislav Janjusic, Krishna Kavi, Brandon Potter

University of North Texas, 1155 Union Circle, Denton 76203, Texas, U.S.A.

Abstract

This paper describes a program profiling and analysis tool called Gleipnir. Gleipnir collects memory access traces and associates each access with a specific program internal structure such as a thread, a function, a data structure or a scalar variable. The data provided by Gleipnir can be used to analyze how program variables and associated memory accesses map to L-1 as well as higher level cache memories. This information can be used to investigate techniques to refactor data or code to improve memory access performance. It is our hypothesis that optimizing cache performance at all levels is very important to both single-core and multi-core processors. In this paper we will describe the Gleipnir tool and some examples of its use in optimizing memory performance. The overall goal of our research is to develop techniques usable by application developers, compilers, and runtime systems to improve the performance of their applications.

Keywords: Program Profiling, Memory Access Traces, Cache Memories, Data and Code Refactoring

1. Introduction

1.1. Application Instrumentation

Embedded and high performance applications are often tuned to improve their performance. The tuning is achieved using analysis tools which provide insight into an application's behavior. An often used method is to instrument the application's code to observe its behavior during an execution on a target system. Typically instrumentation will collect a variety of information about the application. For instance, one may count the number of branches and the number of times a specific branch is taken, the number of function call trees to detect the most heavily invoked function, or the function that consumes most CPU cycles.

There are two types of instrumentation techniques that are used to observe an application's memory access behavior. Static instrumentation, which inserts code to collect desired information during the compilation process, consumes very little runtime overhead. Dynamic instrumentation inserts code after an executable is compiled. The dynamically inserted code examines the precompiled instructions as they execute and it then collects the desired information about the application. Dynamic instrumentation can incur a heavy runtime overhead; analyzing and adding instructions to the original stream can be costly. However, dynamic instrumentation provides greater flexibility and more detailed information than static instrumentation. We chose to use dynamic instrumentation techniques for our purpose.

The main contribution of this paper is the description of our memory analysis tool called Gleipnir, which is built on a widely used binary instrumentation tool called Valgrind [1]. Gleipnir can be used to trace memory accesses

Email address: tjanjusic@unt.edu (Tomislav Janjusic, Krishna Kavi, Brandon Potter)

1877–0509 Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.225

caused by an application, and relate each memory access to the source level variable name and the function/thread that caused the access. Such data may be used to fine-tune applications to improve memory performance.

1.2. Valgrind

Valgrind is a widely used dynamic instrumentation framework. It comes with a set of tools that perform specific program analysis. Valgrind's tools are widely used for debugging and identifying performance bottlenecks.

In essence Valgrind (*Figure 1*) is a core-tool that provides the necessary framework that any instrumentation tool can use for its specific purpose. Broadly speaking the core-tool transforms up to 50 instructions *(Super Blocks, or SB's)* of the executable binary code into a Valgrind specific intermediate representation (IR) which it hands to the instrumentation tool. The instrumentation tool adds code to the given *SB* to collect desired information about the application and returns the instrumented IR to the core-tool. Valgrind then recompiles the IR into executable code and runs the code on a synthetic CPU. It should be noted that the intervention of Valgrind and its instrumentation tools add to execution time, often slowing the actual execution of the application by a factor of 10 to 100 times.

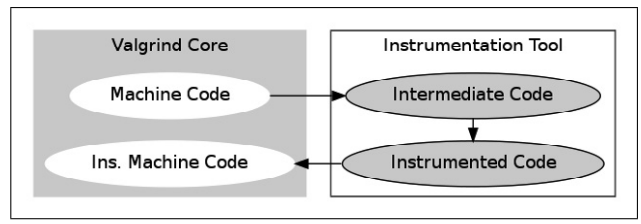

Figure 1: Valgrind Instrumentation

1.2.1. Why Valgrind

Although we decided to build Gleipnir on the Valgrind framework, we explored other available frameworks. We will describe some of the tools that we considered in the Related Work Section. We chose Valgrind for the following reasons.

- It is free and available under the GNU open-source licensing agreement. The core-tool's source code is widely available, and easily customizable.
- It is actively maintained by a set of very dedicated developers.
- It is ported to a variety of popular platforms including x86, AMD64, PPC, and recently to ARM.
- It is compatible with GCC.
- It is used to instrument programs written in a wide variety of source languages.
- It is widely used with a large number of applications, particularly in the high-performance computing arena.

The rest of the paper is organized as follows: Section 2 reviews related work. Section 3 introduces our Gleipnir tool. Section 4 contains a description of Gleipnir's usefulness. Section 5 describes Gleipnir's current status and future work. Section 6 includes our observations and conclusions.

2. Related Work

2.1. Three Classic Approaches

There are three classic approaches to gather performance data: hardware counters, simulation, and analysis models. Each of the three approaches has benefits and drawbacks which are relevant in view of a tool's purpose.

2.2. Hardware Counters

Hardware counters are available on most modern architectures. The counters are available to tools like OProfile, GProf, and DProf [2] [3] [4]. They require specialized CPU hardware and are not standardized in their implementation; counter capabilities vary. Examples of commonly supported architectures include Alpha processors, MIPS, ARM, x86-64, sparc64, and ppc64 [2].

Hardware counters are attractive because they do not impose heavy performance penalties. They achieve high performance by sampling instructions as the program executes. The sampled instructions are appended together to form an execution trace which can be analyzed to find performance bottlenecks. Typical sampling rates range from 1:100 to 1:10000 instructions. The sampling rate is important because it determines the analysis accuracy.

2.3. Simulation

Profiling simulators emulate the system architecture in software. The software emulation allows full control over important components like the CPU and memory. Relative to using hardware counters, simulation profiling is generally more accurate.

The most prominent shortcoming of simulation is overhead. Simulation must maintain the state of all of the components in question and thus is very slow. It is common for simulators to execute 10 to 100 times slower than the original executable.

2.3.1. Pin Framework

Pin and Valgrind are both dynamic binary instrumentation frameworks. Pin [5] is very similar to Valgrind but there are some differences. It supports Linux, Windows and MacOS executables for a variety of Intel processors. Pin comes with an API that abstracts the underlying framework for easy tool development. It supports features which enable it to pin itself onto already running applications and, in many cases, it is faster than Valgrind [5]. Currently, no Pin tool exists which evaluates memory accesses at data granularity, i.e. relating each memory access to a specific program variable.

2.3.2. Other Valgrind Tools

Valgrind already has two noteworthy cache simulation tools: Cachegrind [6] and Callgrind [7]. Cachegrind is a cache profiler tool that provides cache access statistics. Cachegrind comes with a built-in simulator. Since the addresses that Cachegrind simulates are virtual, cache statistics are not perfect. Even so, Cachegrind provides a good picture of the underlying cache behavior. Callgrind is an extension to Cachegrind which agglomerate cache hits and misses at the function granularity. Callgrind comes with a separate GUI representation tool, KCachegrind [8], that clearly depicts bottlenecks during a program's run-time execution. Once again, neither tool provides individual instruction/data accesses nor how they are related to program variables.

2.4. Analysis Models

Cache analysis models are static, statistical models that analyze source code and/or intermediate representations. Since they are static, they can be implemented directly in a compiler. Cache analysis models aim to identify relationships among elements within the code, such as the relationships among loop indices, array sizes and base addresses. With the relationships identified, transformations are applied that are known to improve data locality. One such framework is Cache Miss Equations (CMEs) [9].

The main problem with analysis models is that they predict how the program will execute before execution occurs. They cannot anticipate run-time events or the relationship between static and dynamic structures.

3. Cache Behavior Analysis

3.1. Overview

A CPU's execution is limited by its ability in receiving the necessary data fast enough to do computations. Applications that operate on relatively large amounts of data will require more cache space than is normally available; thus different data items will fall to the same cache locations, causing conflicts, leading to the eviction of previous occupants. In addition to conflicts because of the limited capacities of caches, it has been observed [10] [11]that cache

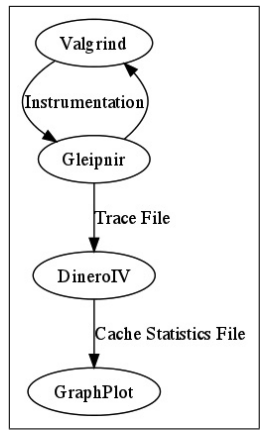

Figure 2: Simulation Cycle

lines are non-uniformly accessed: some cache lines are heavily accessed causing conflicts while other cache lines are underutilized. Poor placement of data is the primary reason for these cache conflicts. The nature of cache accesses and conflicts must be analyzed so that better data placement and code reorganizations can be explored. Careful data ordering is key to efficiently spreading the data across the cache uniformly and improving performance [10]. Current tools, as described in the related section, do not provide relevant information needed for this purpose. Gleipnir provides memory traces on an application's internal structures, i.e., threads, functions, data structures, and scalar variables. In this paper we will demonstrate how Gleipnir's data can be used to explore different data and code placements to improve L-1 cache performance. Improving the performance, through reducing cache conflicts, at L-1 level will produce improvements at higher levels of memory hierarchy. Fewer L-1 misses means fewer visits to higher level caches.

. The tools that comprise our environment are listed below:

- Valgrind - The dynamic binary instrumentation framework.
- Gleipnir - Our instrumentation tool.
- Dinero - A trace driven cache simulator.
- Gnuplot - A tool to represent cache accesses graphically.

3.2. Analysis Environment

Our analysis environment is outlined in (*Figure 2*). We first run the application through Gleipnir. Gleipnir collects all the necessary trace information so that we can later simulate the cache behavior. We use a modified version of DineroIV cache simulator [12] in order to collect cache simulation data related to individual data elements. A user is free to run his or her own cache simulation. In this paper we will show some results of cache simulations for different data layouts and how the accesses appear across each cache line.

3.3. Gleipnir

Gleipnir is a memory analysis tool that maps an application's source code variables to generated data traces. It is built as an intermediate between *Valgrind's Lackey* and *Callgrind*. The trace file generated by *Lackey* is purely a data and instruction trace with loads and stores corresponding to each data address. Gleipnir extends this information with the thread_id, originating function, scope, data structure and the corresponding element, or a single scalar variable name. The reason for adding this information is to allow for cache-simulations to pinpoint the behavior of program variables. In other words, one can explore how different variables use cache memories by observing the reuse distance between consecutive accesses to a given variable, liveness of a variable, the set of conflicting variables in a given

instruction window, etc. Although our current focus is memory accesses, Gleipnir can be modified to track other instruction types or instruction sequences.

```
struct typeA {
    double var1;
    int myArray[10];
};

struct typeA GlStrc;
struct typeA GlStrcArray[10];

int GlScalar;
int GlArray[10];

void func(struct typeA StrcParam[]);

int main(void)
{
    struct typeA LoStrcArray[5];

    int i, LoScalar;
    int LoArray[10];

    GlScalar = 321;
    LoScalar = 123;

    for(i=0; i<2; i++)
        LoArray[i] = GlScalar;

    func(LoStrcArray);

    return 0;
}

void func(struct typeA StrcParam[])
{
    int i;

    for(i=0; i<2; i++){
        GlStrcArray[i].var1 = GlScalar;
        GlStrcArray[i].myArray[0] = GlArray[0];

        StrcParam[i].var1 = GlArray[i];
    }

    return;
}
```

Listing 1: Example Source Code

```
1  S 7ff0001d0 main
2  S 00601040 main GV GlScalar
3  S 7ff0001c8 main LV 0 1 LoScalar
4  S 7ff0001cc main LV 0 1 i
5  L 7ff0001cc main LV 0 1 i
6  L 7ff0001cc main LV 0 1 i
7  L 00601040 main GV GlScalar
8  S 7ff0001a0 main LS 0 1 LoArray[0]
9  L 7ff0001cc main LV 0 1 i
10 S 7ff0001cc main LV 0 1 i
11 L 7ff0001cc main LV 0 1 i
12 L 7ff0001cc main LV 0 1 i
13 L 00601040 main GV GlScalar
14 S 7ff0001a4 main LS 0 1 LoArray[1]
15 L 7ff0001cc main LV 0 1 i
16 S 7ff0001cc main LV 0 1 i
17 L 7ff0001cc main LV 0 1 i
18 S 7ff0000a8 main
19 S 7ff0000a0 func
20 S 7ff000088 func LV 0 1 StrcParam
21 S 7ff00009c func LV 0 1 i
22 L 7ff00009c func LV 0 1 i
23 L 7ff00009c func LV 0 1 i
24 L 00601040 func GV GlScalar
25 S 00601060 func GS GlStrcArray[0].var1
26 L 7ff00009c func LV 0 1 i
27 L 00601240 func GS GlArray[0]
28 S 00601068 func GS GlStrcArray[0].myArray[0]
29 L 7ff00009c func LV 0 1 i
30 L 7ff000088 func LV 0 1 StrcParam
31 L 7ff00009c func LV 0 1 i
32 L 00601240 func GS GlArray[0]
33 S 7ff0000b0 func LS 1 1 LoStrcArray[0].var1
34 L 7ff00009c func LV 0 1 i
35 S 7ff00009c func LV 0 1 i
36 L 7ff00009c func LV 0 1 i
37 L 7ff00009c func LV 0 1 i
38 L 00601040 func GV GlScalar
39 S 00601090 func GS GlStrcArray[1].var1
40 L 7ff00009c func LV 0 1 i
41 L 00601240 func GS GlArray[0]
42 S 00601098 func GS GlStrcArray[1].myArray[0]
43 L 7ff00009c func LV 0 1 i
44 L 7ff000088 func LV 0 1 StrcParam
```

Listing 2: Gleipnir Generated Trace-file

Listing 1 and *Listing 2* show an example of trace information collected by Gleipnir. *Listing 1* shows a very simple program which contains a few data structures and a single function. *Listing 2* shows a segment of the trace file generated by Gleipnir for program code in *Listing 1*. The format of the output generated by Gleipnir is straightforward.

[*LOAD* | *STORE*; *ADDRESS*; *CALLING_FUNCTION*; *SCOPE*; *FRAME No.*; *THREAD Id*; *ELEMENT*]

The first column indicates if the access is a load (L) or a store (S). The second column is the address of the (data) memory accessed. The remaining columns show the name of the variable, the function that caused the memory access, if the variable accessed is a local (L) or global (G), and a formal parameter or a structure (S). In case of structures and arrays, Gleipnir will also identify the element that is accessed.

Observe that Gleipnir displays only information relevant to a variable's loads and stores. The listing does not show other instructions executed by the code which may intervene on memory access instructions. Gleipnir may, provided that the option is enabled, capture memory accesses for both instructions and data. This allows users to improve both instruction and data caches or minimize conflicts at higher level caches which are often unified.

The example source code in *Listing 1* includes local and global structures, a function call and structure manipulation through parameters. We need to stress that this is for illustration purposes only and thus the code was compiled

with disabled optimizations. *Listing 2* shows the relevant portion of the trace file. Observe that variable *i* appears to be repeatedly loaded and stored, this is due to two reasons. Firstly, the trace omits intervening instructions that are the cause of the load or store, thus we see repeated load and stores. Secondly, compiler optimizations have been disabled in order to show the example trace. Current compilers support the debug flag at various optimization levels; however, enabling optimization for a simple program such as *Listing 1* would virtually strip any memory load and store reference for this example and would serve little to illustrate Gleipnir. Furthermore, non-trivial programs that have been optimized should retain significant portion of their debug even through optimizations, but this depends on the compiler.

Listing 1 shows variables with the same name even though they are declared in different scopes. Gleipnir distinguishes variables with scope identities. For instance, observe the for-loop on lines 36-41. The corresponding trace file is shown in *Listing 2* on lines 23-44. Access to *LoStrcArray* will change scope however a user would not be able to infer this from the name alone. Scoping value serves as additional information in order to accurately represent a memory access pattern.

3.4. DineroIV

DineroIV is a trace-driven cache simulator [12]. It is highly customizable and straightforward to modify. We choose to use DineroIV for our simulations since it is widely used in academia and the source code is available. We modified DineroIV to use the traces generated by Gleipnir so that its cache-analysis can reflect per variable statistics. Original Dinero used traces that only identified memory accesses and instructions, data load and data stores. Our modifications changed the format to include the additional information as outlined in Section 3.3. We also report cache accesses (hits and misses) caused by each program variable and how these accesses map to individual cache lines. Also, we are developing filters so that the analysis is restricted to the desired function, thread, instruction window or data structure. Such analyses can be used to observe conflicting data variables and explore different data layouts. We will provide some examples of such uses of Gleipnir in Section 4.

3.5. Plotting

Using Dinero's cache statistics we represent the data access patterns using a graphing utility. As stated above, we can filter the cache data generated by the modified DineroIV and observe only the cache behaviors of selected program components. Figures 3, 4, and 5 show some examples of the graphs generated. The graphs depict the data access patterns generated by a simple matrix multiplication program. However, we have modified the access patterns in these matrix multiplication programs. We will discuss the generated graphs in the subsequent sections.

We want to emphasize that the traces generated by our Gleipnir tool can be used with other trace driven simulators with appropriate reformatting. Likewise, one can use other graphical tools to display the cache statistics.

4. Trace Driven Memory Analysis

4.1. Data Mapping

The motivation for developing Gleipnir is the poor cache behaviors observed for most applications. In addition to misses due to limited cache capacities, it has been observed [10] [11] that cache lines (or sets) are non-uniformly accessed - some sets are accessed 100s or even 1000s of times more often than other sets. The heavily accessed lines cause most cache conflicts. Conflict misses can be reduced by spreading accesses more uniformly across the cache lines. In order to achieve this goal, information on how program variables are mapped to cache lines, the reuse distance between successive accesses to the same variable, liveness of variables and the set of variables that conflict within a range of accesses are needed. Gleipnir provides detailed traces of memory accesses with this information.

4.2. Example Use of Gleipnir

We provide a simple example to show how Gleipnir can be used to visualize an application's memory behavior. We implemented matrix multiplication using different data layouts by declaring three matrices differently. In the first we declared matrices conventionally as shown in (*Listing 3*). In the second example we tiled the matrices as shown in (*Listing 4*). Finally, since the matrices are accessed in the same manner, we grouped elements of *MatrixA* and *MatrixResult* as shown in (*Listing 5*).

```
int MatrixA[100][100];
int MatrixB[100][100];
int MatrixResult[100][100];
```

Listing 3: Regular Matrix Declaration

```
1 #define N 100
2 #define nl 10
3 #define nh 10
4
5 int matrixA[nh][nh][nl][nl];
6 int matrixB[nh][nh][nl][nl];
7 int matrixC[N][N];
```

Listing 4: Reordered Matrix Declaration

```
1 struct mType{
2     int Res;
3     int A;
4 };
5
6 struct mType matrix[100][100];
7 int B[100][100];
```

Listing 5: Structure Matrix Declaration

Figures 3, 4, and 5 show the overall accesses pattern to different cache lines in the three different versions of the matrix multiplication. The figures depict a 32kilobyte direct mapped L1 cache with 32bytes per block, but a user is free to manipulate cache configurations as they see fit. The *X axis* shows cache sets and the *Y axis* shows the corresponding hits and misses. The figures help depict the overall access pattern for the entire cache. Notice the spikes in each corresponding figure. This is happening because of statically allocated data which is accessed during loop iterations, variables such as i, j, and k. From these and similar graphs a user can observe cache alignment conflicts between variables and data structures. A user may use variable offsetting to remedy such poor placements and force variables into a particular region of the cache. Due to space limitations, we did not include analyses that focus on specific ranges of memory accesses.

Note that we tiled the data which also requires tiling of loops. However, tiling of data implies changes to array strides and thus it can impact prefetching of data elements. Ultimately we would like to note that the purpose of this example is to show how a user can visualize and ultimately explore new strategies for both data and code modifications in order to improve memory access performance. For instance, the co-location of *MatrixA* and *MatrixResult* may be beneficial since their elements will be fetched together.

We feel that the access information to individual program variables can help compilers and programmers to explore different data layouts as well as different code structures, e.g., loop tiling, fusion, or exchanges. Such information can also be used to evaluate different memory allocation techniques.

4.3. Gleipnir and Cache Simulation Combined

We are still debating if the cache simulator, such as DineroIV, should be integrated with Gleipnir or not. Integrating will simplify the cache analysis requiring the use of a single tool. However, we feel that Gleipnir traces may be useful for other types of memory analyses and thus we should expose the traces generated by Gleipnir.

4.4. Memory Analysis Using Gleipnir

A programmer that has a visual picture of how the data is accessed, and more importantly where the data ultimately resides in the cache, can make intelligent choices on data placement. Even though compilers and memory allocators determine the placement of program variables, the programmer has the ability to affect the data placement. To illustrate an embarrassingly simple example, consider inserting dummy variables between actual variable declarations to change the location of variables in the cache. Another example may be to reorder the structures' elements to change access patterns to these elements.

5. Current State of Gleipnir, Limitations and Future work

At this time, the core functionality of Gleipnir is implemented and is ready for use. But as is the case with most ongoing research, there are some limitations with the current version of Gleipnir. Some of the limitations are due to our reliance on Valgrind. At present, the speed of Gleipnir is an issue; our tests show that Gleipnir runs 30-100 times slower than the original executable without any instrumentation. These numbers are very similar to other Valgrind tools. We have gained some performance improvements by modifying Valgrind's core and we will continue to further improve the performance. The table below shows the trace file size in relation to the number of instructions collected.

In order to relate memory accesses to program source code, Gleipnir requires that programs be compiled with debug option enabled. This adds to the overhead that already exists with Valgrind. We are exploring how to limit the overhead of accessing the debug information by restructuring Valgrind's internal debug storage. Current debug symbol table lookup is inefficient, and we feel the speed of debug accesses can be amortized to a constant factor.

Benchmark	L/S Instructions	Trace-file size
basic math	651,272,648	16GB
AES	192,740,716	5.6GB
dijkstra	166,177,792	4.5GB
fft	130,850,859	3.2GB
qsort	111,689,557	2.8GB

Table 1: Comparison of memory access counts and corresponding trace file sizes for the *mibench* benchmark suite.

At present if a user wants to track accesses to variables within a single function, Gleipnir first traces all accesses in the entire code and filters out unnecessary trace information. The current implementation also results in very large trace files that often exceed several gigabytes in size, see *table[1]*. Once again, filtering information at earlier stages can reduce the size of trace files. We are exploring the use of annotations to define when Gleipnir should start tracing memory accesses. In general, it is desirable to allow tracing of a specific data structure, accesses caused by a specific thread or function.

Instrumenting multi-threaded programs is an ongoing research within the Valgrind community. At present each thread is instrumented independently. Thus Gleipnir reports memory accesses caused by each thread independently. This does not reflect true execution where the execution of threads are interleaved. While waiting for support from Valgrind, we are exploring various models for combining the memory access traces of threads to simulate interleaved execution of threads.

Other extensions to Gleipnir are also planned. These include tracing other instructions (not just memory accesses), as well as sequences of instructions. Such analyses may be useful when dealing with heterogeneous cores (e.g., CPUs and GPUs) or systems with reconfigurable components (i.e., FPGAs). This is a long term plan but the primary reason behind the need to trace a sequence of instruction, i.e. the longest sequence of instructions, is to be able to determine sections of code that may potentially be shipped off to the GPU or FPGA.

6. Conclusions

In this paper we described our memory analysis tool called Gleipnir. Gleipnir is built using a well-known binary instrumentation framework, Valgrind. When programs are compiled with debug option, Gleipnir can trace memory accesses caused by the application and relate each access to a specific source level variable name, the function or thread that caused the access. Such information can be valuable in exploring different data layouts, code and data refactoring techniques (such as tiling, fusing, etc). While the tool in its current state can be used for these purposes, we are working to improve the usability by developing graphical user interfaces. At present Gleipnir is not optimized: execution speeds are 10s-100s times slower than un-instrumented code and the size of the trace files are very large. We are working to improve both the speed of Gleipnir and the sizes of output files generated. We are also working to allow user annotations so that tracing of specific data structures, functions or threads can be selected.

We want to emphasize that Gleipnir is evolving. But we will be glad to provide the current version of Gleipnir to interested readers.

7. Acknowledgements

This work is made possible in part by support from the NSF Net-Centric Industry/University Research Center, a grant from Advanced Micro Devices, and ORNL summer internship support for Janjusic. The name Gleipnir is taken from Norse mythology and refers to a deceptively strong ribbon or tie that was used to restrain a vicious wolf called Fenrir. We hope that Gleipnir will help to restrain the memory hogs in applications.

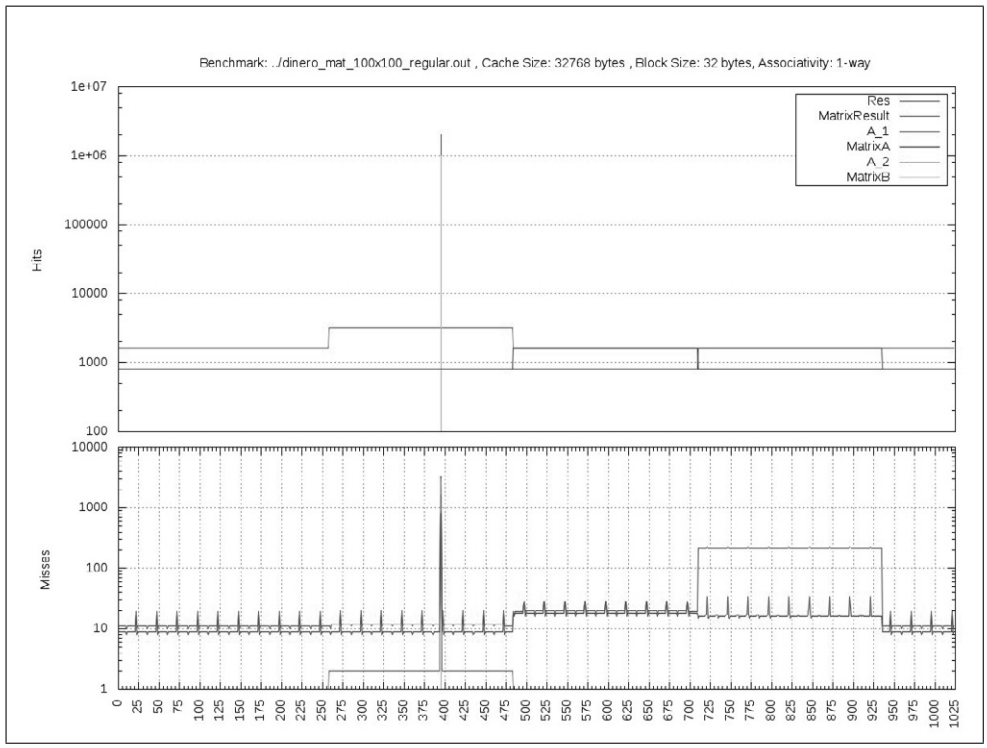

Figure 3: Standard Access Pattern.

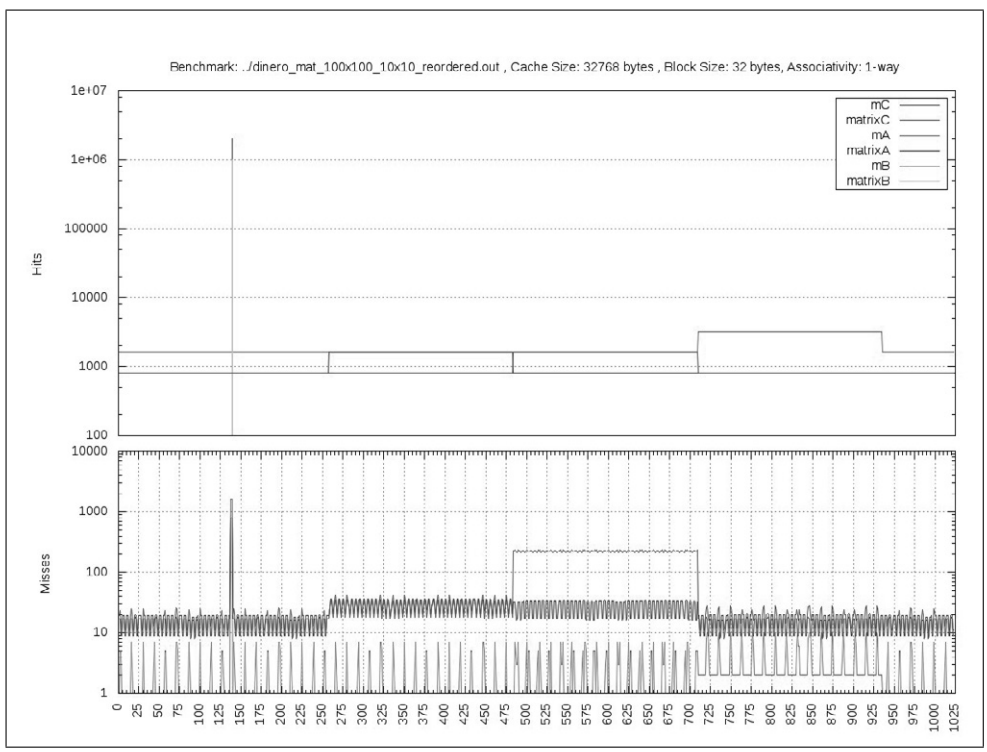

Figure 4: 10x10 Submatrices Access Pattern.

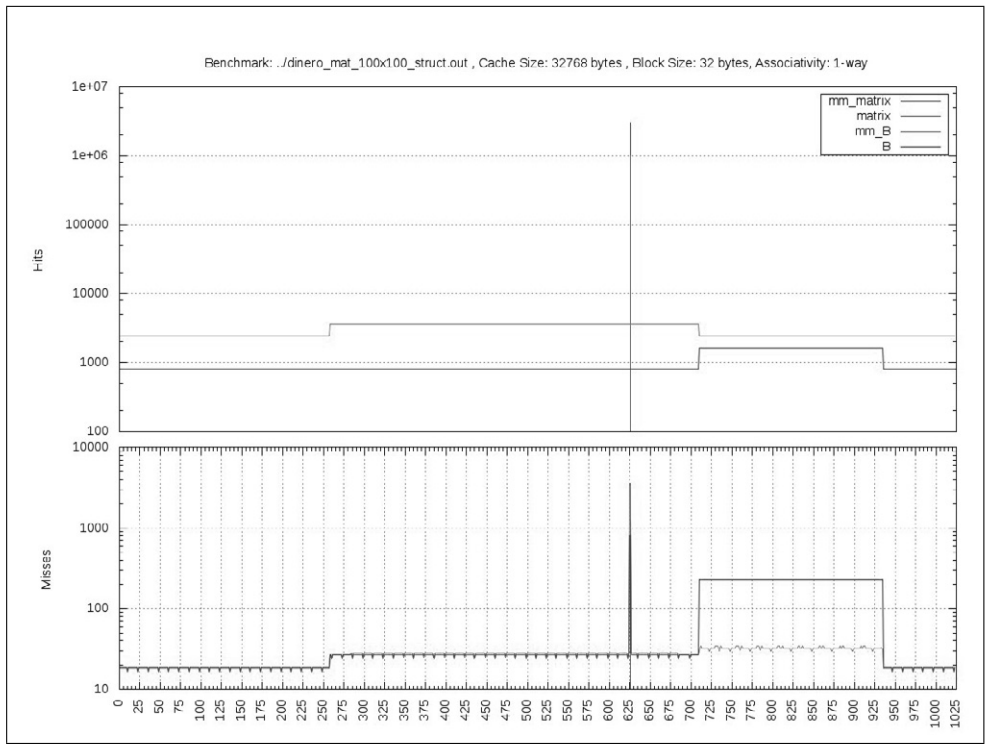

Figure 5: Structure Declared Access Pattern.

References

[1] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07, ACM, New York, NY, USA, 2007, pp. 89–100. doi:http://doi.acm.org/10.1145/1250734.1250746.
URL http://doi.acm.org/10.1145/1250734.1250746

[2] J. Levon, Oprofile manual.
URL http://oprofile.sourceforge.net/doc/

[3] S. L. Graham, P. B. Kessler, M. K. McKusick, gprof: a call graph execution profiler, SIGPLAN Not. 39 (2004) 49–57. doi:http://doi.acm.org/10.1145/989393.989401.
URL http://doi.acm.org/10.1145/989393.989401

[4] J. Tao, T. Gaugler, W. Karl, A profiling tool for detecting cache-critical data structures, in: Euro-Par, 2007, pp. 52–61.

[5] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, in: Programming Language Design and Implementation, Chicago, IL, 2005, pp. 190–200.

[6] N. Nethercote, Dynamic binary analysis and instrumentation., Ph.D. thesis, University of Cambridge (2004).

[7] J. Weidendorfer, M. Kowarschik, C. Trinitis, A tool suite for simulation based analysis of memory access behavior, in: ICCS 2004: 4th International Conference on Computational Science, Vol. 3038 of LNCS, Springer, 2004, pp. 440–447.
URL http://wwwbode.in.tum.de/ weidendo/publications/iccs04.pdf

[8] J. Weidendorfer, kcachegrind, call graph viewer.
URL http://kcachegrind.sourceforge.net/html/Documentation.html

[9] S. Ghosh, M. Martonosi, S. Malik, Automated cache optimizations using cme driven diagnosis, in: In International Conference on Supercomputing (ICS'00, 2000, pp. 316–326.

[10] A. Naz, O. Adamo, K. M. Kavi, T. Janjusic, Improving uniformity of cache access pattern using split data caches, in: ISCA on Parallel and Distributed Computing and Communication Systems, 2009, pp. 128–134.

[11] C. Zhang, Reducing cache misses through programmable decoders, ACM Transactions on Architecture and Code Optimization 4 (2008) 5:1–5:31. doi:http://doi.acm.org/10.1145/1328195.1328200.
URL http://doi.acm.org/10.1145/1328195.1328200

[12] M. D. H. Jan Edler, Dinero iv trace-driven uniprocessor cache simulator.
URL http://www.cs.wisc.edu/ markhill/DineroIV