Trace Driven Data Structure Transformations

Tomislav Janjusic and Krishna M. Kavi
Computer Science and Engineering
University of North Texas
Denton, Texas 76207-7102
Email: {tjanjusic,krishna.kavi}@unt.edu

Christos Kartsaklis
Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN 37830
Email: kartsaklisc@ornl.gov

***Abstract*—As the complexity of scientific codes and computational hardware increases it is increasingly important to study the effects of data-structure layouts on program memory behavior. Program structure layouts affect the memory performance differently, therefore we need the capability to effectively study such transformations without the need to rewrite application codes. Trace-driven simulations are an effective and convenient mechanism to simulate program behavior at various granularities. During an application's execution, a tool known as a tracer or profiler, collects program flow data and records program instructions. The trace-file consists of tuples that associate each program instruction with program internal variables. In this paper we outline a proof-of-concept mechanism to apply data-structure transformations during trace simulation and observe effects on memory without the need to manually transform an application's code.**

I. Introduction

A memory trace is a collection of all executed memory access references during program execution. A single trace-line is any instruction that modifies a memory location at a given address X. For rudimentary analysis it is sufficient to analyze a trace consisting of a 3-tuple trace-line consisting of an access type, address, and the size of the data access. This information is enough to estimate how well a program utilizes memory and the potential bottlenecks in memory performance.

However, if we wish to study an application in greater detail we must include additional meta-data that describe the collected traces. For example we need to know which program module executed the trace-line and which data element was accessed during that memory reference. This level of detail is necessary to reason about any possible modification's we may want to apply to the application.

There are various tools which enable trace-driven memory analysis [1], [2], [3], [4], [5]; however, for the purposes of this study we will concentrate on Gleipnir [6] and DineroIV [7]. Gleipnir is a memory profiling and tracing tool developed as a plug-in tool for a binary instrumentation framework Valgrind [8]. It collects fine-grained memory trace information which enables users to perform advanced memory analysis. DineroIV is trace-driven cache simulator which performs basic cache analysis. It was modified to take full advantage of Gleipnir's generated traces. Original DineroIV reports cache statistics of a given trace input that gives a user a general overview of a cache's behavior. The modified version tracks cache statistics that pertain to function and variable level accuracy. This means that a user is able to observe conflicts between program structures and analyze if any transformation should be considered to improve an application's cache behavior.

Trace transformation is the most recent work added to the simulator and is the core topic of this paper. It is a new module described in later sections which performs rule based structure transformations.

The rest of the paper is organized as follows. Section 2 describes related work in the area of program tuning and performance optimization. Section 3 describes the Gleipnir and DineroIV tool in greater detail. Section 4 describes the trace transformation module, the latest addition to DineroIV. Section 5 demonstrates a few basic transformation examples. Section 6 describes future research directions, and conclusions are discussed in section 7.

II. Related Work

In order to tune and optimize an application programmers (i.e. users) profile applications using various profiling tools. The tools can be categorized into analysis tools, simulations, and hardware performance analysis, depending on their implementation.

- Analysis tools come in a variety of flavors ranging from compiler driven[9] static analysis to full dynamic profiling. As a rule of thumb offline application analsis or static analysis is faster but less accurate [10], [4]. Dynamic or runtime application analysis is slower but more accurate [2], [3], [8]. This is because dynamic analysis allows the tool to account for any code paths which are not visible during static anlysis.
- A simulation is observing hardware behavior in software. This means that an application must be executed on simulated hardware to observe any effects it has on potential host hardware. The main advantage of simulation is that a user is in full control of the simulated hardware. This allows the user to pause, forward, or even execute an application in reverse. Simulation's drawbacks are in speed and accuracy.
- Hardware counter performance analysis is observing application's behavior through hardware performance counters. Performance counters are hardware units that ship with modern processors. When invoked they enable to user to collect information about the hardware that the application is running on. Enabling hardware counters

can be achieved through the operating system or directly from the application using library interfaces such as PAPI [11]. This also means that the application must be modified in order to insert the calls to activate and collect this information. Hardware performance counters cannot isolate an application's effects from the overall system meaning that inserting calls into the application already skewes the application's performance.

For our purposes we used Gleipnir. Gleipnir is built as a plug-in tool for Valgrind[8]. A binary instrumentation framework which falls under software analysis tools. The framework already comes with a set of cache-profiling and trace-generating tools but none provide the level of detail captured with Gleipnir. A cache profiling tool is a tool which collects information about a processor's cache state. Most tools show an overall statistic, and the more advanced tools isolate program's function behavior or report cache statistics per source code line.

In order to compare and contrast other tools in this area we must take note of what Gleipnir is and what it is not. First and foremost, Gleipnir is a memory tracing tool, the analysis of the traces is external and in our case left to our modified version of DineroIV [7]. Decoupling the trace collection from the simulation offers several benefits. In addition to cache analysis other uses for traces are possible. Gleipnir's traces are meant to provide detailed meta-data information on every application's memory access. This information may be used for a variety of research areas as well as performance profiling.

III. Tool Description

A. Tracing Instructions

Gleipnir takes advantage of two key Valgrind's components. It utilizes Valgrind's internal debug parser and the ability to instrument memory related events. Each event is either an Instruction Read, Data Read, Data Write, or Data Modify. A symbol table is a compiler generated table that contains all the source code information associated with a program's internal variables. A debug parser can process this information and retrieve debug information (source code variables) associated with instruction addresses. Gleipnir takes advantage of Valgrind's debug information parser and feeds the instrumented instructions into the parser to expand the trace tuple with additional meta-data.

Figure 1 shows a typical Gleipnir trace line for a static variable. Visible instructions are annotated with the address to be fetched, modified, or written to; the function which caused the access; the scope of the variable, thread that executed the code; and finally the data element itself. Figure 1 shows the general trace-line format. The first field is the access type, either a *Load, Store, Modify, or X (for miscellaneous instructions)*. The second field is the *virtual address* of the data to be accessed followed by the *function*. If any symbol information exists the trace will be supplemented with the element's *Local* or *Global* scope, and the element's type *Variable* or *Structure*. The next two numbers indicate the elements *Frame* and the originating *Thread*. The final value is the variable name itself. If the accessed element is part of a structure then the variable name is shown as a nested structure of the element name and the structure it belongs to.

S	7ff000108	malloc	LS	0	1	_zzq_args[5]

Fig. 1: Gleipnir's trace line

Gleipnir relies on Valgrind's internal debug parser, therefore any application that needs to be profiled for local and global structures must be compiled with the compiler's *-g* flag.

The source code in Listing 1 and 2 show a sample of static and global data structures and their respective trace when traced by Gleipnir. From the example we can observe several of Gleipnir's key components. The source code's *main* function starts with a Gleipnir specific macro that turns the instrumentation on. With the macros GLEIPNIR_START_INSTRUMENTATION and GLEIPNIR_STOP_INSTRUMENTATION the user can control which code regions to instrument. This is useful for fast forward the tracing.

Listing 1: Example source code

```
struct _typeA {
  double d1;
  int myArray[10];
};
struct _typeA glStruct;
struct _typeA glStructArray[10];

int glScalar;
int glArray[10];

void foo(struct _typeA StrcParam[])
{
  int i;
  for(i=0; i<2; i++){
    glStructArray[i].d1 = glScalar;
    glStructArray[i].myArray[i] = glArray[i+1];

    StrcParam[i].d1 = glArray[i];
  }
  return;
}

int main(void)
{
  GLEIPNIR_START_INSTRUMENTATION;

  struct _typeA lcStrcArray[5];
  int i, lcScalar, lcArray[10];

  glScalar = 321;
  lcScalar = 123;

  for(i=0; i<2; i++)
    lcArray[i] = glScalar;

  foo(lcStrcArray);

  GLEIPNIR_STOP_INSTRUMENTATION;
  return 0;
}
```

The code defines and declares global structures and a scalar element at line 4-13. The function *foo* is defined on lines 15-25. At line 31 and 32 the *main* function declares several structures and couple scalar elements. In lines 34-38 the structures and scalar elements are accessed. Finally in line 40 a call to function *foo* is made and the corresponding code in line 15-25 is executed.

The program's execution is observable from the corresponding trace. Each trace line represent the data elements accessed during the program execution.

Listing 2: trace-file snippet

```
START PID 13063                                          1
S 7ff0001b0 8 main LV 0 1 _zzq_result                    2
L 7ff0001b0 8 main                                       3
S 000601040 4 main GV glScalar                           4
S 7ff0001bc 4 main LV 0 1 lcScalar                       5
S 7ff0001b8 4 main LV 0 1 i                              6
L 7ff0001b8 4 main LV 0 1 i                              7
L 000601040 4 main GV glScalar                           8
L 7ff0001b8 4 main LV 0 1 i                              9
S 7ff000180 4 main LS 0 1 lcArray[0]                     10
M 7ff0001b8 4 main LV 0 1 i                              11
L 7ff0001b8 4 main LV 0 1 i                              12
L 000601040 4 main GV glScalar                           13
L 7ff0001b8 4 main LV 0 1 i                              14
S 7ff000184 4 main LS 0 1 lcArray[1]                     15
M 7ff0001b8 4 main LV 0 1 i                              16
L 7ff0001b8 4 main LV 0 1 i                              17
S 7ff000050 8 main                                       18
S 7ff000048 8 foo                                        19
S 7ff000030 8 foo LV 0 1 StrcParam                       20
S 7ff000044 4 foo LV 0 1 i                               21
L 7ff000044 4 foo LV 0 1 i                               22
L 000601040 4 foo GV glScalar                            23
L 7ff000044 4 foo LV 0 1 i                               24
S 0006010e0 8 foo GS glStructArray[0].d1                 25
L 7ff000044 4 foo LV 0 1 i                               26
L 0006010a4 4 foo GS glArray[1]                          27
L 7ff000044 4 foo LV 0 1 i                               28
S 0006010e8 4 foo GS glStructArray[0].myArray            29
    [0]
L 7ff000044 4 foo LV 0 1 i                               30
L 7ff000030 8 foo LV 0 1 StrcParam                       31
L 7ff000044 4 foo LV 0 1 i                               32
L 0006010a0 4 foo GS glArray[0]                          33
S 7ff000060 8 foo LS 1 1 lcStrcArray[0].d1               34
M 7ff000044 4 foo LV 0 1 i                               35
L 7ff000044 4 foo LV 0 1 i                               36
L 000601040 4 foo GV glScalar                            37
L 7ff000044 4 foo LV 0 1 i                               38
S 000601110 8 foo GS glStructArray[1].d1                 39
L 7ff000044 4 foo LV 0 1 i                               40
L 0006010a8 4 foo GS glArray[2]                          41
L 7ff000044 4 foo LV 0 1 i                               42
S 00060111c 4 foo GS glStructArray[1].myArray            43
    [1]
L 7ff000044 4 foo LV 0 1 i                               44
L 7ff000030 8 foo LV 0 1 StrcParam                       45
L 7ff000044 4 foo LV 0 1 i                               46
L 0006010a4 4 foo GS glArray[1]                          47
S 7ff000090 8 foo LS 1 1 lcStrcArray[1].d1               48
M 7ff000044 4 foo LV 0 1 i                               49
L 7ff000044 4 foo LV 0 1 i                               50
```

The trace starts with the main function storing a value to the global variable *glScalar* in line 4 coresponding to the source code line 34. Note that we do not store the frame number and thread id because global variables are globally visible; therefore, there is no need to identify the frame of the corresponding variable.

The *For* loop's 2 iterations on lines 37-48 are shown in trace lines 6-17. Notice that trace lines are loading the loop iteration variable *i* as expected. In trace lines 19-50 the trace shows the executed function *foo*. At each step the trace identifies the structure or elements scope, the calling function, the frame id and thread id if this info is necessary, and each structure's element. It also identifies the corresponding offset of each element if the access is in an array of structures.

Because we are only interested in data structure transformation's we do not explicitly trace instruction fetches from memory and therefore disabled this option in our examples.

IV. Transformation Environment

A typical analysis procedure involves three steps as outlined in figure 2. A user first runs the application through Gleipnir. This generates the trace-file necessary for further parsing or trace-analysis. Gleipnir collects all the necessary information for fine-grained cache-simulation. A modified version of DineroIV ships with Gleipnir tailored to take advantage of Gleipnir's traces. A user is free to run his or her own simulator or apply any additional trace analyzers.

Plotting the graphs is supplemented through scripts that parse DineroIV output. A work on a graphical user interface client is in the works but not available at this time.

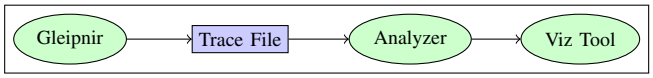

Fig. 2: Tracing and Analysis Cycle

A. Trace Transformation

We perform trace transformation during cache analysis. Referring back to Figure 2 this means that our module is added to the analyzer component (i.e. cache simulator). The key additions to an already modified DineroIV include the trace transformation rules and trace transformation functions. A transformation rule is read through a separate file provided by the user. This implies that the user must provide the full structure definition. The rules describe the original and the transformed structure. The user provides a full structure definition with includes structure element type information. We will describe the rules in greater detail in later subsection for now note that each rule is one to one mapping between the two structures. The rules are hard-coded and the mapping between an *in* rule and an *out* rule is not bi-directional. This means that if a structure with the same nesting is encountered the simulator will simply ignore it. The transformation functions determines if a trace line is described by structure meta-data

and applies the necessary transformation as described by the rules.

This process is summarized as follows:

1) Initialize the rules: at the beginning of each simulation the simulator will read the *in* and *out* rules and set up a new base address and size for the new structure. Listing 5, 8, 11 describe our three example rules.
2) Check validity: the trace file is processed one trace line at a time. The simulator will read each trace line and break the meta-data variable into a nested list of structures with the final element at the bottom. If the variable is part of the *in* rules then that trace line needs to be transformed.
3) Apply transformation: The *in* rule is mapped to the *out* rule. This involves an intermediate step to calculate a newly accessed address. If the new structure is referenced through indirection then additional instruction insertions are applied that deal with pointer indirections.
4) Print the transformation: Any new trace generated will be traced into a *transformed_trace.out* file.
5) A complete and transformed trace is compared with the original trace (eg. Figure 5). This may be performed using a *diff* tool or other mechanism.

In this paper we will describe the following three rules: structure-of-arrays (SoA) to array-of-structure (AoS) transformations, single level nested structures to single level indirect structures, and access displacement for cache set-pinning purposes.

1) Structure of arrays to array of structures transformations: Listing 3 and 4 show an example of structure of arrays to array of structures transformations. Listing 3 shows a simple structure of arrays. In data addressability terms this means that *mX* and *mY* array elements are offset by the array size. Assuming that elements are offset by larger than a cache block size any access two both *mX* and *mY* elements will result in two cache loads. To avoid this we must produce a transformation which collocates both indexed elements for example through an array of structures.

Listing 3: Transformation 1A

```
int main(int aArgc, char **aArgv) {                1
  typedef struct {                                 2
    int mX[LEN];                                   3
    double mY[16];}                                4
  MyStructOfArrays;                                5
  MyStructOfArrays lSoA;                           6
  GLEIPNIR_START_INSTRUMENTATION;                  7
  for (int lI=0 ; lI<LEN ; lI++) {                 8
    lSoA.mX[lI] = (int) lI;                        9
    lSoA.mY[lI] = (double) lI;}                    10
  GLEIPNIR_STOP_INSTRUMENTATION;                   11
  return (0); }                                    12
```

Note that transformations in Listing 4 are user transformations. Our goal is to produce a trace from 1A to 1B dynamically through the simulator relying only on a rule described in Listing 5.

Listing 4: Transformation 1B

```
int main(int aArgc, char **aArgv) {                1
  typedef struct { int mX; double mY; }            2
      MyStruct;                                    3
  MyStruct lAoS[LEN];                              4
  GLEIPNIR_START_INSTRUMENTATION;                  5
  for (int lI=0 ; lI<LEN ; lI++) {                 6
    lAoS[lI].mX = (int) lI;                        7
    lAoS[lI].mY = (double) lI;                     8
  }                                                9
  GLEIPNIR_STOP_INSTRUMENTATION;                   10
  return 0;}                                       11
```

Figure 3 and 4 are visual representations of the different structure layouts. Using graphical analyses we can visualize the transformations as they are layed out on our cache model. For example in Figure 3 and 4 we have simulated a 32k byte size, 32bytes per block directly mapped cache. The difference in the graphs show the more uniformly access pattern observed in Figure 4. Note that the goal of our automated transformations is to give the user the ability to visualize various structure layout transformations.

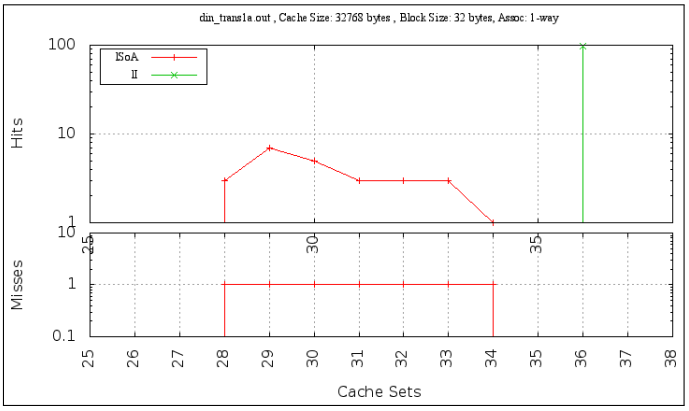

Fig. 3: Structure of Arrays to Array of Structures

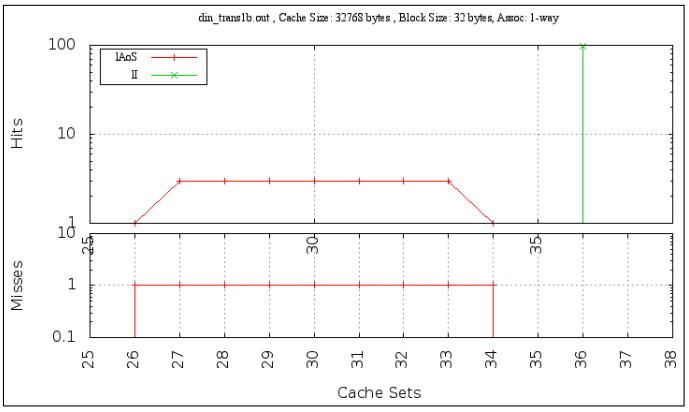

Fig. 4: Structure of Arrays to Array of Structures

Listing 5 describes an input file that outlines the original and transformed structure. The current limitation is that structure's element names must match because we rely on the element's name to map and determine if a rule is specified.

Listing 5: Rules for transformation of 1A to 1B

```
in:                                  1
struct lSoA {                        2
   int mX[16];                       3
   double mY[16];                    4
};                                   5
out:                                 6
struct lAoS {                        7
   int mX;                           8
   double mY;                        9
} [16];                              10
```

Figure 5 shows the original and the transformed trace. On the left side we see the original trace while on the right side we can observe the trace generated by the simulator. The base address of structures has changed because the structure mapping changes due to allignment. The tool used to show the trace differences is a graphical diff tool. Figure 5 is only for illustration purposes it aims to show how we can use new trace information to study transformed structures. Struct-of-Arrays to Array-of-Structs transformations were explored in [12].

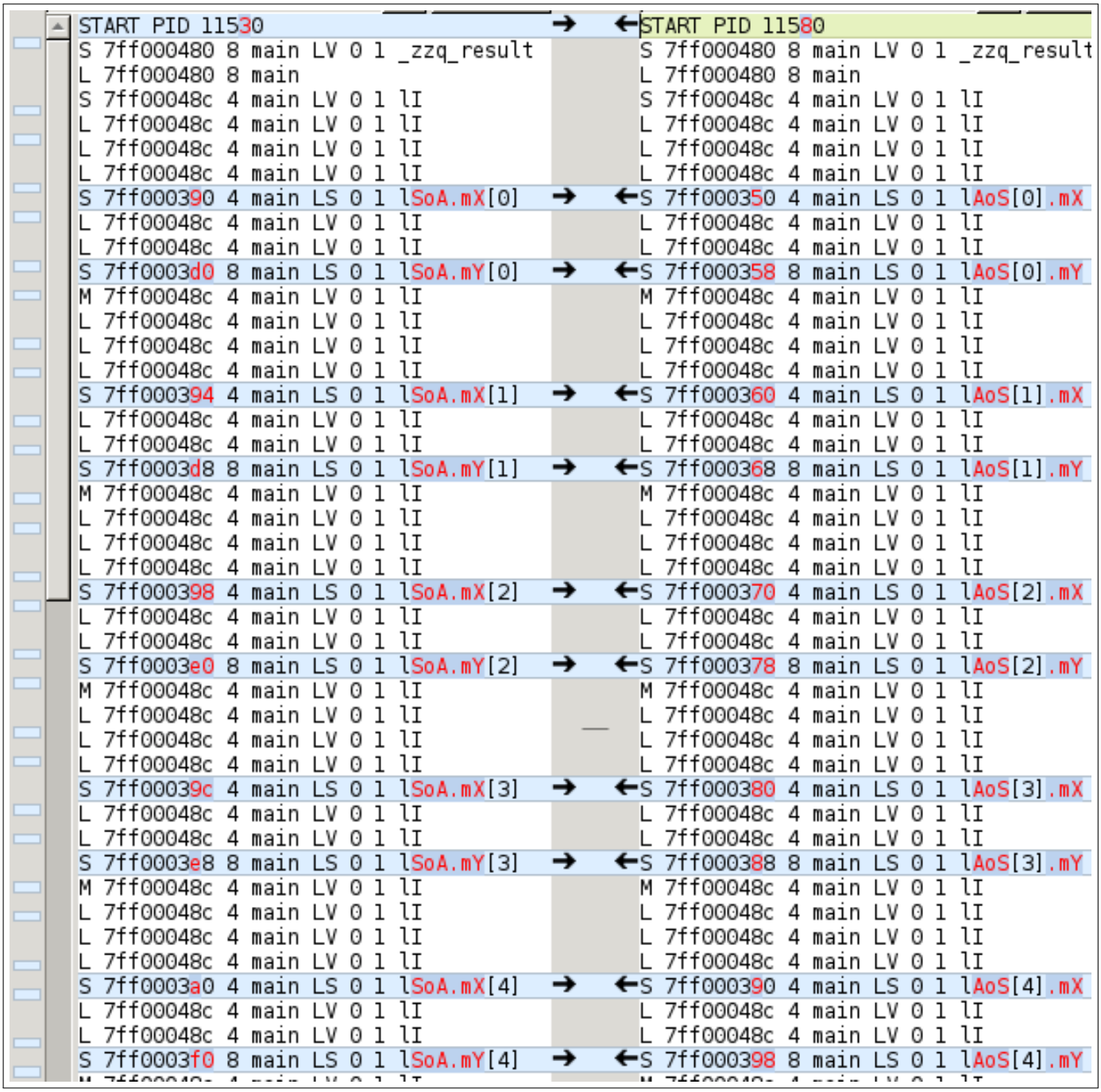

Fig. 5: Structure of Arrays to Array of Structures

2) Single level nested structures to single level indirect structures: Another useful transformation is to offset a nested structure into a pool of memory. A common example is traversing a list. The goal is to collocate elements of similar temporal locality into unique spatial memory pools.

Listing 6 and 7 are examples of nested structures transformed into structures accessed through pointers.

Listing 6: Transformation 2A

```
int main(int aArgc, char **aArgv) {                    1
  typedef struct {                                     2
    int mFrequentlyUsed;                               3
    struct { double mY; int mZ; } mRarelyUsed;         4
  } MyInlineStruct;                                    5
                                                       6
  MyInlineStruct lS1[LEN];                             7
  GLEIPNIR_START_INSTRUMENTATION;                      8
  for (int lI=0 ; lI<LEN ; lI++) {                     9
    lS1[lI].mFrequentlyUsed = lI;                      10
    lS1[lI].mRarelyUsed.mY  = lI;                      11
    lS1[lI].mRarelyUsed.mZ  = lI;                      12
  }                                                    13
  GLEIPNIR_STOP_INSTRUMENTATION;                       14
  return (0);                                          15
}                                                      16
```

Listing 7: Transformation 2B

```
int main(int aArgc, char **aArgv) {                    1
  typedef struct { double mY; int mZ; }                2
      RarelyUsed;
  typedef struct {                                     3
    int mFrequentlyUsed;                               4
    RarelyUsed *mRarelyUsed;                           5
  } MyOutlinedStruct;                                  6
                                                       7
  RarelyUsed lStorageForRarelyUsed[LEN];               8
  MyOutlinedStruct lS2[LEN];                           9
                                                       10
  for (int lI=0 ; lI<LEN ; lI++) {                     11
    lS2[lI].mRarelyUsed =                              12
        lStorageForRarelyUsed+lI;}
                                                       13
  GLEIPNIR_START_INSTRUMENTATION;                      14
  for (int lI=0 ; lI<LEN ; lI++) {                     15
    lS2[lI].mFrequentlyUsed = lI;                      16
    lS2[lI].mRarelyUsed->mY = lI;                      17
    lS2[lI].mRarelyUsed->mZ = lI;                      18
  }                                                    19
  GLEIPNIR_STOP_INSTRUMENTATION;                       20
  return (0);}                                         21
```

Listing 6 defines a structure of a frequently used element and a rarely used structure, lines 2-5. The goal of this transformation is to keep the rarely used structure in an outside pool of memory and collocate frequently used elements. An example of this transformation is seen in Listing 7. The *mRarelyUsed* structure, line 2, is accessed through a pointer *mRarelyUsed*, line 5. Note that an access to *mRarelyUsed* introduces a level of indirection due to the pointer. The transformed trace must reflect this transformation because the new trace should reflect any additional memory accesses which result from transforming structures. In this case the indirection is an extra trace line that inserts a load to the pointer reference *mRarelyUsed*. Our options are to copy the preceding line and change the effective address or arbitrarily insert a call to any stack memory reference.

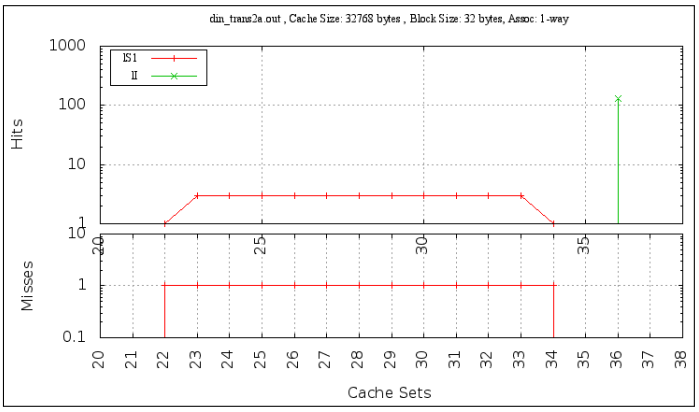

Fig. 6: single level nested structure

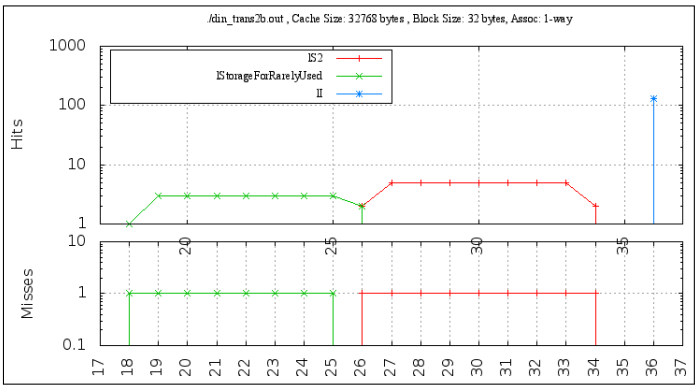

Fig. 7: Structure access through indirection

Figure 6 and 7 show the visual representation of the preceding transformations. It can be observed that the uniformity of cache accesses changed due to the extra load instructions. In addition Figure 7 shows the changes that are the result from offloading data to an external structure.

Listing 8 outlines the rule file needed to perform this transformation. The *in* rule defines a bottom-up approach to nesting. The top most defined rule is the deepest structure in the structure tree. This allows us to compute the outer structure's size easier and it also allows us to perform transformations in multiple structure nestings. The *out* rule defines the transformation. It consists of two structures and a pointer reference. The pointer type dictates which structure our rule applies for. The format is different in that we must specify the structure name to be transformed and the pointer name.

Figure 8 shows the example trace compared with the transformed trace. Notice the indirection throught the loaded pointer *mRarelyUsed* in the green highlighted lines.

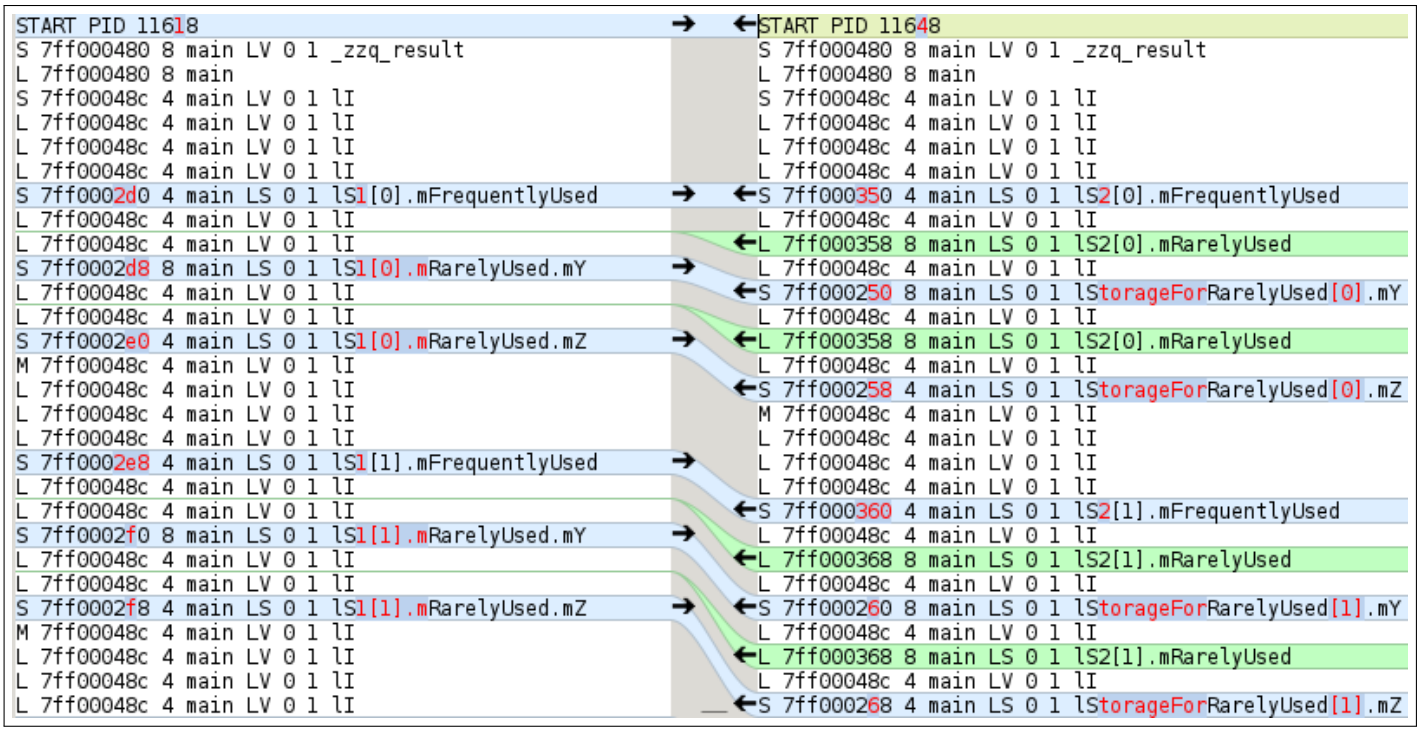

Fig. 8: Nested structure to Structure with indirection

Listing 8: Rules for transformation of 2A to 2B

```
in:
struct mRarelyUsed {
  double mY;
  int mZ;
};
struct lS1 {
  int mFrequentlyUsed;
  struct mRarelyUsed;
}[16];

out:
struct lStorageForRarelyUsed {
  int mY;
  double mZ;
}[16];
struct lS2 {
  int mFrequentlyUsed;
  * mRarelyUsed:lStorageForRarelyUsed;
}[16];
```

3) Stride Accesses: Stride Accesses are used to transform a structure such that accesses to structure elements are mapped to specific cache sets. The goal with a stride access pattern is to restrict structure or array elements access to specific cache lines.

Listing 9 shows an access pattern to a contiguous array. If the array is larger than the cache a contiguous access to the array will replace the entire cache and populate it with array elements. The desired transformation is to force array accesses to specific cache sets through striding. The idea is to conceive a layout that exploits the cache's set/column−mapping policy so that accesses to select data structures can be confined in a subset of the cache. Assume the PowerPC 440 cache[13] which is 32k bytes, 64ways per set with 32bytes per cache line and implements a round-robin eviction policy. Furthermore assume that we have 4096 bytes of contiguous accesses and a cold cache. This will consume the first eight columns by writting 256 bytes in each set (i.e. 8 cache lines/set). The transformation will direct all 4096 bytes of accesses to a single set, achieving 50% residency as a set cannot hold that many bytes (64 ways $\times$ 32 bytes $=$ 2048 bytes).

Listing 9: Transformation 3A

```
int main(int aArgc, char **aArgv) {
  int lContiguousArray[LEN];
  GLEIPNIR_START_INSTRUMENTATION;
  for (int lI=0 ; lI<LEN ; lI++) {
    lContiguousArray[lI] = lI;
  }
  GLEIPNIR_STOP_INSTRUMENTATION;
  return (0);}
```

Listing 10 shows a transformation where the array access pattern is restricted to a subset of cache sets. The stride formula is computed in line 9-10. The new *lSetHashingArray* is of length LEN*SETS, for a LEN value of 1024 this is 64k bytes (1024 * 16 * 4 bytes). In contrast the original array is of size LEN, 4k bytes. The idea is to force an array mapping to the same set by offsetting array elements. The downside to this technique is that space is wasted. The upside is that we can reduce cache trashing by maintaining the same amount of cache misses for the array structure. Another drawback is that a user must be aware of the host system's cache configuration

in order to apply successfull transformations.

Listing 10: Transformation 3B

```
int main(int aArgc, char **aArgv) {
  #define SETS 16
  #define CACHELINE 32
  const int ITEMSPERLINE=CACHELINE/sizeof(int);
  int lSetHashingArray[LEN*SETS];
  GLEIPNIR_START_INSTRUMENTATION;
  for (int lI=0 ; lI<LEN ; lI++) {
    lSetHashingArray[(lI/ITEMSPERLINE)*
  (SETS*ITEMSPERLINE) + (lI%ITEMSPERLINE)] =
      lI;
  }
  GLEIPNIR_STOP_INSTRUMENTATION;
  return (0);}
```

The rules in Listing 11 define an array and a stride formula. The stride formula to compute the rules is hard-coded. We only identify the index and use it to compute the stride. Any intermediate indexes are hard-coded into the simulator because they are used when computing a stride (eg. *ITEMSPERLINE*) and must be accounted for in the trace. To account for the additional instructions we have hand forced the simulator to inject additional instructions.

Listing 11: Rules for transformation of 3A to 3B

```
in:
int lContiguousArray[16]:lSetHashingArray;
out:
int lSetHashingArray[256((i/8)*(16*8)+(i%8))];
```

Figure 9 shows the original trace on the left and a semi-automatic[1] transformed trace. Notice that this is not a fully automatic transformed trace because the stride indirection is not fully implemented. The main limitation in our implementation is accounting for all scalar variables that might be accessed during a stride access.

Figure 10 and 11 show the visual output of the two transformations. In Figure 10 we have a contiguous access

[1]We preselected additional instruction after running the hand transformed code

Fig. 9: Structure of Array to Array's of Structures.

to 0-15 cache sets and each set has 64 columns. In the second figure we have offset the cache stride by a number of bytes according to the formula in Listing 11. The offsetting directs all accesses to a single set (Figure 11), hence giving the impression that the data structure is "pinned" on it. Because of *lSetHashingArray*'s base address every access is indexed to set 11, but a displacement may be used to yield another set.

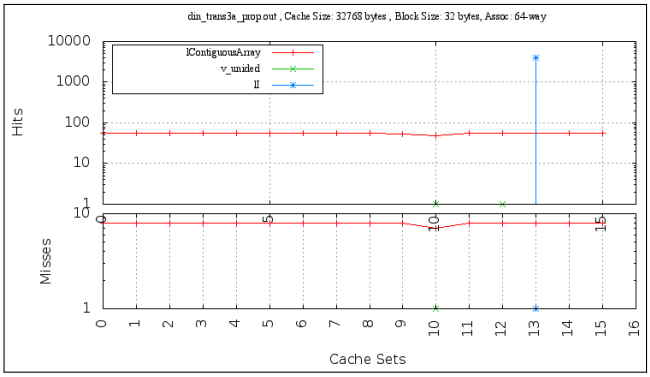

Fig. 10: Contiguous array

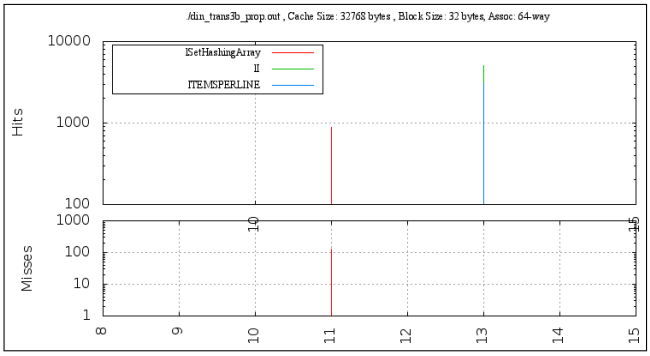

Fig. 11: Array striding

V. Conclusions

In this paper we presented a proof-of-concept trace-oriented technique for automated data-structure transformation. It is trace-driven because the main input is a non-transformed trace-file. The transformation is automated because the trace is transformed automatically during simulation execution. Transformations are rule dependent and we have elaborated on three common transformations which we explained in the paper. Software reengineering for performance purposes is a challenging task. We have presented a novel approach to this task of exploring the transformation space of data structures that does not require source code modifications. Similarly to computational steering, our method allows users to safely and interactively modify data structures in a kernel in order to see what behavior the transformation triggers in the cache including, importantly, how transformations may lead to unforeseen conflicts. In conclusion we want to emphasize that the proposed techniques are proof-of-concept rather than a finished product.

VI. Future Work

There are some shortcomings with our approach. Due to the nature of the tracing tool we can apply our transformations to static data structures only. This is in many ways a limitation and therefore we must explore the ability to transform dynamic structures as well. Moreover, the trace information is limited by the instrumentation tool to private caches only because the addresses used are virtual addresses. This is a limitation because if we wish to simulate a shared level cache we must take physical addresses into account. This can be remedied by using a more sophisticated instrumentation tool[2], or by mapping kernel page-maps information directly into the trace.

Acknowledgment

This work is made possible in part by support from the NSF Net-Centric Industry/University Research Center, and ORNL summer internship support. Computational resources were provided by UNT's High Performance Computing Initiative, and Oak Ridge National Laboratory.

References

[1] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools." ACM, 1994, pp. 196–205.

[2] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *In Programming Language Design and Implementation*. ACM Press, 2005, pp. 190–200.

[3] D. L. Bruening, "Efficient, transparent and comprehensive runtime code manipulation," Cambridge, MA, USA, Tech. Rep., 2004, AAI0807735.

[4] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, pp. 317–329, November 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=1080622.1080630

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

[6] T. Janjusic, K. M. Kavi, and B. Potter, "International conference on computational science, ICCS 2011 Gleipnir: A Memory Analysis Tool," *Procedia CS*, vol. 4, pp. 2058–2067, 2011.

[7] M. D. H. Jan Edler, "DineroIV Trace-Driven Uniprocessor Cache Simulator." [Online]. Available: http://www.cs.wisc.edu/ markhill/DineroIV

[8] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746

[9] G. Chakrabarti and F. Chow, "Structure layout optimizations in the Open64 Compiler: Design, Implementation, and Measurements," *International Symposium on Code Generation and Optimization*, 2008.

[10] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler (with retrospective)," in *Best of PLDI*, 1982, pp. 49–57.

[11] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[12] W.-C. Ma and C.-L. Yang, "Using Intel streaming SIMD extensions for 3D geometry processing," in *Proceedings of the Third IEEE Pacific Rim Conference on Multimedia: Advances in Multimedia Information Processing*, ser. PCM '02. London, UK, UK: Springer-Verlag, 2002, pp. 1080–1087. [Online]. Available: http://dl.acm.org/citation.cfm?id=648110.747979

[13] "The PowerPC 440 Core, A high-performance, superscalar processor core for embedded applications," IBM Microelectronics Division, Tech. Rep., 1999.

[2]To our knowledge, there does not exist any instrumentation tool capable of delivering trace information related to objects physical address placement.