An Approach for Symbolic Mapping of Memory References

Luiz DeRose[1], K. Ekanadham[2], and Simone Sbaraglia[2]

[1] Cray Inc.
Mendota Heights, MN, USA
ldr@cray.com
[2] IBM Research,
Yorktown Heights, NY, USA
(eknath,sbaragli)@us.ibm.com

Abstract. Understanding and tuning memory system performance is a critical issue for most scientific programs to achieve reasonable performance on current high performance systems. Users need a *data-centric* performance measurement infrastructure that can help them understand the precise memory references in their program that are causing poor utilization of the memory subsystem. However, a *data-centric* performance tool requires the mapping of memory references to the corresponding symbolic names of the data structure, which is non trivial, especially for mapping local variables and dynamically allocated data structures. In this paper we describe with examples the algorithms and extensions implemented in the SIGMA environment for symbolic mapping of memory references to data structures.

1 Introduction

A variety of performance measurement, analysis, and visualization tools have been created to help programmers tune and optimize their applications. These tools range from source code profilers, tracers, and binary analysis tools (e.g., SvPablo [4], TAU [8], KOJAK [10], VampirGuideView [5], HPCView [6], and Paradyn [7]) to libraries and utilities to access hardware performance counters built into microprocessors (e.g., Perfctr [9], PAPI [1], and the HPM Toolkit [2]).

In general, performance measurement and visualization tools tend to be *control-centric*, since they focus on the control structure of the programs (e.g., loops and functions), where traditionally application programmers concentrate when searching for performance bottlenecks. However, due to the advances in microprocessors and computer systems designs, there has been a shift in the performance characteristics of scientific programs from being computation bounded to being memory/data-access bound. Hence, understanding and tuning memory system performance is today a critical issue for most scientific programs to achieve reasonable performance on current high performance systems. Thus, users also need a *data-centric* performance measurement infrastructure that can help them understand the precise memory references in their program that are causing poor utilization of the memory hierarchy. Fine-grained information such as this is useful for tuning loop kernels, understanding the cache behavior of new

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 141–148, 2004.

algorithms, and to investigate how different parts of a program and its data structures compete for and interact within the memory subsystem.

In [3], we presented SIGMA (Software Infrastructure to Guide Memory Analysis), a new data collection framework and a family of cache analysis tools. The goal of the SIGMA environment is to provide detailed cache information by gathering memory reference data using software-based instrumentation. This is used to provide feedback to programmers to help them apply program transformations that improve cache performance. An important characteristic of the SIGMA infrastructure is that it is both control-centric and data-centric; it presents performance metrics expressed in terms of functions, as well as data structures defined in the source code.

In order to provide such relation to the source program, whenever a memory reference is made, SIGMA gathers the address of the instruction that made the reference and the address of the referenced data. It then associates the instruction address to the source line in the program that made the reference, and the data address to the symbolic name of the data structure that corresponds to the reference. While the transformation of the instruction address into the corresponding source line can be performed easily by analyzing the line number and symbol table stored in the executable, mapping the data address to the corresponding symbolic name of the data structure requires considerably more work. This is especially true if we want to support local variables and dynamically allocated data structures, which were not supported in the work presented in [3]. The main problem in supporting the mapping of local (automatic) variables and dynamically allocated variables is that the virtual address cannot be known statically at link time.

The remainder of this paper is devoted to describing the algorithms implemented in SIGMA that map each memory reference to the corresponding data structure. Section 2 presents a brief overview of the SIGMA infrastructure. In Section 3, we describe with examples our algorithms for mapping memory references to variable names. Finally, in Section 4, we present our conclusions.

2 The SIGMA Approach

The SIGMA environment consists of three main components: a pre-execution instrumentation utility that reads the binary file to locate and instrument all instructions that refer to memory locations; a runtime data collection engine that collects the streams of instructions and memory references generated by the instrumentation, performs the symbolic mapping, and a highly efficient lossless trace compression. It also provides a number of simulation and analysis tools that process the compressed memory reference trace to provide programmers with tuning information.

SIGMA uses a binary instrumentation approach so that it can gather data about the actual memory references generated by optimizing compilers rather than using source instrumentation which would gather data about the user specified array references. The simulation and analysis tools include a TLB simulator, a data cache simulator, a data prefetcher simulator, and a query mechanism that allows users to obtain performance metrics and memory usage statistics for current and hypothetical architectures. For more details on the SIGMA environment, please refer to [3].

The runtime symbolic conversion engine, which is the subject of this paper, performs the transformation:

$$(instructionAddress, dataAddress) \rightsquigarrow (sourceLine, varName, arrayElement) \tag{1}$$

and maintains a table where each column represents an *active* data structure and each row represents a function in the program. Each entry in the table contains counters for the memory events of interest, such as cache accesses, hits, and misses. Once the symbolic transformation (1) is completed the module updates the counters for the entry corresponding to the source line *sourceLine* and the data structure *varName*. If the variable is an array the precise element references is also available.

3 Mapping Memory References to Data Structures

In order to perform the transformation: $dataAddress \rightsquigarrow varName$ the runtime engine builds and maintains a linked list shown in Figure 1, where each entry corresponds to an allocated virtual address range and carries the information about the symbolic name of the data structure that corresponds to each address range. When an address "a" is accessed, the runtime engine searches the list for an entry "i" such that $a_i \leq a < b_i$, in order to match the reference to the data structure x_i.

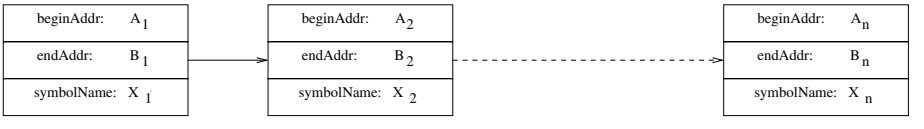

Fig. 1. Table to map memory addresses to symbolic names.

Clearly, since data structures can be allocated and deallocated dynamically, this list has to be dynamically updated at runtime. Moreover, the instrumentation engine must capture the information about allocations and deallocations. A further difficulty is presented by the fact that stack variables are not identified by a global address range, but by an offset within the stack pointer of the function where they are defined.

In this section we describe our approach to support the mapping of each data address to the corresponding symbolic name of the data structure. We divide the description in three parts: the algorithm for mapping of global variables, the algorithm for mapping of dynamically allocated variables, and the algorithm for mapping of stack variables. Our examples are based on the IBM compilers for AIX and their binary representation (XCOFF), but the ideas can be extended to other environments.

3.1 Mapping Global Variables

Global variables are allocated in the *Data Segment* of the binary, where the information about the virtual address assigned to each variable and its size is completely known at

link time. If the program is compiled with the debugger flag, this information is stored by the compiler in the executable in the form of tables. Hence, by analyzing these tables we can build the linked list of Figure 1 statically, before the application starts executing. For example, let us consider the following C pseudocode:

```
double var1;
double var2 = 1;
int array1[100];
int main(int argc, char *argv[]) { ... }
```

By analyzing the symbol table of the compiled executable, we find the following entries that refer to the data structure `var1`:

```
Symbol Class = C_EXT   Value = 0x20000660 Name = var1
Stab   Class = C_GSYM  Value = 0x00000000 Name = var1:G-13
```

The first entry classifies the symbol `var1` as a variable statically allocated at the virtual address $0x20000660$. The second entry identifies the variable as a global variable (symbol *G*) of type -13, which is the internal code for *double*. We can therefore infer that the variable `var1` will be attributed the address range $[0x20000660, 0x20000668)$ and we can build an entry in the linked list. Similarly, the array `array1` and the variable `var2` are represented as:

```
Symbol Class = C_EXT   Value = 0x200001e0 Name = var2
Stab   Class = C_GSYM  Value = 0x00000000 Name = var2:G-13
Symbol Class = C_EXT   Value = 0x20000668 Name = array1
Stab   Class = C_GSYM  Value = 0x00000000 Name = array1:G6
Stab   Class = C_DECL  Value = 0x00000000 Name = :t6=ar0;0;99;-1
```

where the last Stab entry defines the type 6 as an array $0, \ldots, 99$ of integers. The table for this binary is shown in Figure 2.

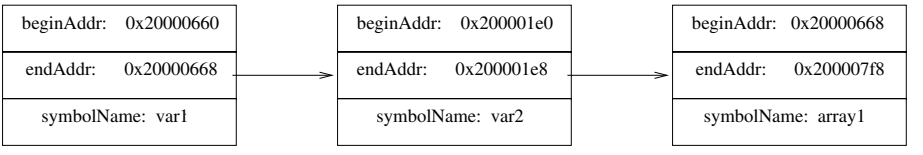

Fig. 2. Table to map memory addresses to symbolic names.

3.2 Mapping Dynamically Allocated Variables

Normally, the sizes of dynamic data structures depend on user input and are often unknown at compile time. Moreover, a data structure size may need to be changed during the program execution. Hence, an address range of a dynamically allocated data structure is assigned to the variable at runtime, and cannot be deduced uniquely from the executable tables. Furthermore, allocated variables can be "released" when they are no more needed, and the same address range (or a subset of it) can be re-assigned to some other dynamic variable.

In order to account for such situations, we needed to expand our instrumentation utility to capture the allocation and deallocation requests, the address range allocated or freed, and the symbolic data structure that is bound from time to time to the address range. To illustrate the algorithm, let us consider the following C example and its corresponding entries in the symbol table:

```
int *A;
A = (int *)malloc(n);

Stab   Class = C_DECL  Value = 0x00000000 Name = :t4=*-1
Symbol Class = C_EXT   Value = 0x2000055c Name = A
Stab   Class = C_GSYM  Value = 0x00000000 Name = A:G4
```

The symbol table of the executable contains an entry for the global variable `A` (of type 4, i.e., pointer to integer, as specified in the first Stab entry), which is associated to the virtual address $0x2000055c$. When the `malloc` function is called, it returns the address x of the newly allocated address range. By intercepting at runtime this return address and the corresponding argument passed to `malloc`, we can infer that a new address range $[x, x + n)$ has been allocated. Unfortunately, the `malloc` call does not directly provide any information about the symbolic name that the new address range is associated with. However, the compiler usually stores the new address x into the memory location identified by `A`. The code generated for the `malloc` call is usually of the following type, where `R3` indicates the register that is used to pass the first argument to a function and to collect its return code:

```
store n into R3
call malloc (which returns the address x in the register R3)
store R3 into 0x2000055c
```

An optimizing compiler might avoid storing the address x and just keep it in a register. In this case, we are unable to bind the address range with the symbolic name, and we classify such references to belong to a dynamically allocated area whose name is unknown. In practice, however, this situation only occurs when the allocated memory is used for a very limited amount of time and then released, and is not encountered often in real applications where the allocated memory is heavily reused (for example in a loop) before being released.

The algorithm to track dynamically allocated memory is the following: first, by analyzing the executable tables we create an entry for `A`, as shown in Figure 3. Second, from the `malloc` instrumentation, we obtain the parameter passed to `malloc` (size n) and the returning value (address x), and instantiate an entry, as shown in Figure 4, where the name is still undefined. Then, when a store instruction: `store x into a` is executed, we search the list for an entry whose name is undefined and x is in the begin address of its range. If we find such an entry, we infer that the address range is now associated with the data structure whose virtual address is a. In the example above, when `store R3 into 0x2000055c` is executed, we change the list by assigning the name `A` to the allocated variable, as shown in Figure 5.

Finally, each time *free* is called, we capture the address passed to it and search the table for an entry that corresponds to it. We then remove the entry from the table, since it is no more “active”.

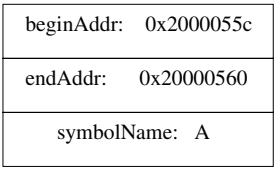

Fig. 3. Table entry for a pointer declaration.

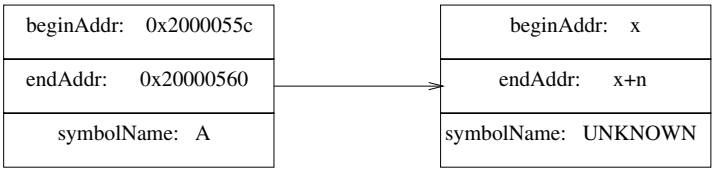

Fig. 4. Table entry after a dynamic allocation at address x of size n.

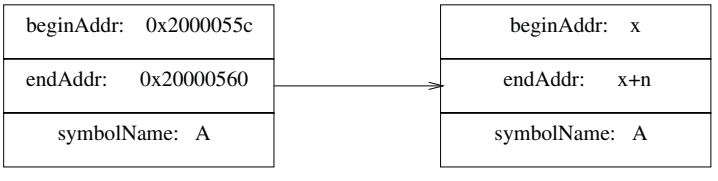

Fig. 5. Table entry of a dynamically allocated variable after symbolic mapping.

3.3 Mapping Stack Variables

When a stack variable is declared in the source code, such as an automatic variable in C or a local variable in Fortran 90, the virtual address that will be assigned to the variable will depend on the position of the stack pointer when the function is called, and is therefore unknown at compile time. However, the compiler stores in the executable the offset of each stack variable in the stack frame. For instance, the pseudocode

```
int foo(void) {
   int var1;
   int array1[100];
   ...
}
```

would carry in the executable the following symbols:

```
Stab   Class = C_FUN  Value = 0x00000000 Name = foo:F-1
Symbol Class = C_FCN  Value = 0x100003d4 Name = .bf

Stab   Class = C_LSYM value = 0x00000040 Name = var1:-1
Stab   Class = C_LSYM Value = 0x00000048 Name = array1:3
Stab   Class = C_DECL Value = 0x00000000 Name = :t3=ar0;0;99;-1

Symbol Class = C_FCN  Value = 0x100003f0 Name = .ef
```

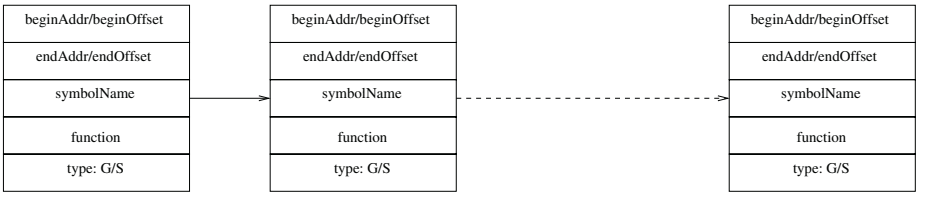

Fig. 6. Table to map memory addresses to symbolic names.

which identify `var1` as a symbol defined locally in the function `foo`. `var1` is a symbol of type *int*, allocated at offset $0x40$ in the stack frame, and `array1` is an array $0, \ldots, 99$ of integers, allocated at offset $0x48$. The `.bf` and `.ef` symbols denote the begin and end of a function variable definition.

A local variable is therefore defined by a pair *(function, offset)*, where *function* indicates the function where the variable is defined and *offset* indicates the stack offset where the variable will be allocated. In order to map stack memory references to the symbol associated with them, we need to be able to identify, for each stack reference, the stack offset of the reference and the function whose stack frame we are accessing. The approach is the following: first, the table in Figure 1 was extended to accommodate the information about local variables, as shown in Figure 6. The *beginAddr* and *endAddr* fields are interpreted as absolute addresses or offsets, depending whether the variable is a global or local symbol (G or S in the type field). The field *function* indicates the function where the variable was declared and is empty for global variables. Second, we extended the instrumentation utility to fetch the value of the stack pointer. Each time the stack pointer changes, we record the event as a couple *(stkPointer, function)* where *stkPointer* is the new value of the stack pointer and *function* is the function that is currently executing.

During execution, we maintain an internal stack structure called *stkList*. Each time the stack pointer changes we search the *stkList* from the top and if we find an entry that matches the current stack pointer, we make that entry the top of the *stkList*. Otherwise, we add the current couple *(stkPointer, function)* to the top of the *stkList*. For instance, let us consider the following call and return sequence:

```
f1() --> f2() --> f3() --> f2() --> f1()
```

and let us further assume, for the sake of simplicity, that there are no stack pointer changes other than the ones involved in the function calls and returns. When `f1` is called, a new stack pointer s_1 is allocated. We capture the stack pointer change and create an entry $(f1, s_1)$ in the *stkList*. Then, when `f2` is invoked, and therefore a new stack pointer s_2 is allocated, we add it to our stack: $(f1, s_1) \rightarrow (f2, s_2)$. Similarly for `f3` we update the table as $(f1, s_1) \rightarrow (f2, s_2) \rightarrow (f3, s_3)$. When `f3` returns, its stack frame is poped and the new value of the stack pointer becomes s_2. We then delete the entry from the list: $(f1, s_1) \rightarrow (f2, s_2)$ and so on. In this way, if the function `f2` accesses a variable whose address x is, say, in the range $[s_1, s_2)$ we can immediately identify it as a reference to a variable declared in `f1`. We will then search our symbolic linked list for an entry whose function field equals `f1` and such that *beginOffset* $\leq x \leq$ *endOffset*.

4 Conclusions

In this paper we presented the algorithms implemented in the SIGMA environment to map memory references to data structures. With the mapping of global variables, dynamically allocated variables, and stack variables, such as automatic variables in C and local variables in Fortran 90, SIGMA provides *data-centric* performance information that is useful for tuning loop kernels, understanding the cache behavior of new algorithms, and to investigate how different parts of a program and its data structure compete for and interact within the memory subsystem in current and hypothetical systems.

In addition we described the extensions to SIGMA that were needed to support the symbolic memory mapping. These extensions include the binary instrumentation of the functions for allocation (`malloc`) and deallocation (`free`) of variables, needed for symbolic mapping of dynamic data structures, and the simulation of a stack frame in the runtime data collection engine, needed for symbolic mapping of local variables.

References

1. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.
2. Luiz DeRose. The Hardware Performance Monitor Toolkit. In *Proceedings of Euro-Par*, pages 122–131, August 2001.
3. Luiz DeRose, K. Ekanadham, Jeffrey K. Hollingsworth, and Simone Sbaraglia. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.
4. Luiz DeRose and Daniel Reed. Svpablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proceedings of the International Conference on Parallel Processing*, pages 311–318, August 1999.
5. Seon Wook Kim, Bob Kuhn, Michael Voss, Hans-Christian Hoppe, and Wolfgang Nagel. VGV: Supporting Performance Analysis of Object-Oriented Mixed MPI/OpenMP Parallel Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.
6. John Mellor-Crummey, Robert Fowler, Gabriel Marin, and Nathan Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, (23):81–101, April 2002.
7. Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, November 1995.
8. Bernd Mohr, Allen Malony, and Janice Cuny. TAU Tuning and Analysis Utilities for Portable Parallel Programming. In G. Wilson, editor, *Parallel Programming using C++*. M.I.T. Press, 1996.
9. Mikael Pettersson. *Linux X86 Performance-Monitoring Counters Driver.* *http://user.it.uu.se/~mikpe/linux/perfctr/*. Computing Science Department; Uppsala University - Sweden, 2002.
10. Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing'*, 49(10–11):421–439, November 2003.