Assigning Blame: Mapping Performance to High Level Parallel Programming Abstractions

Nick Rutar and Jeffrey K. Hollingsworth

Computer Science Department
University of Maryland
College Park, MD 20742, USA
{rutar,hollings}@cs.umd.edu

Abstract. Parallel programs are increasingly being written using programming frameworks and other environments that allow parallel constructs to be programmed with greater ease. The data structures used allow the modeling of complex mathematical structures like linear systems and partial differential equations using high-level programming abstractions. While this allows programmers to model complex systems in a more intuitive way, it also makes the debugging and profiling of these systems more difficult due to the complexity of mapping these high level abstractions down to the low level parallel programming constructs. This work discusses mapping mechanisms, called variable blame, for creating these mappings and using them to assist in the profiling and debugging of programs created using advanced parallel programming techniques. We also include an example of a prototype implementation of the system profiling three programs.

1 Introduction

As parallel systems become larger and more powerful, the problems that can be solved using these systems become larger and more complex. However, there is a divide between those software engineers and parallel language designers who know how to program for and utilize these systems and the people who actually have the problems that could use such systems. To help bridge this gap, recent years have seen more and more parallel frameworks and libraries[1,2] arising to assist the application scientists in creating programs to utilize distributed systems. These environments have abstractions that hide away many of the lower level parallel language constructs and APIs that let developers program in terms of real world mathematical constructs.

All of the above points are positive for the scientific community, but they also bring about some interesting problems. An important issue introduced is profiling and debugging of these systems. There are many profiling tools that target parallel applications specifically. However, when these abstractions are introduced it becomes more and more difficult to diagnose runtime issues and debug them using conventional means. The higher level the abstractions, the harder it is to figure out the lower level constructs that map to them and subsequently discover where performance problems are occurring. This affects the end user as well as the designers of the frameworks who are looking to improve the performance of their software.

Programs with multileveled abstractions introduce unique ways of approaching how to profile the program. For traditional profiling tools, measured variables (time, cache misses, floating point operations, etc.) are given in terms of easily delimited program elements (functions, basic blocks, source lines). Programs developed utilizing parallel frameworks can also be measured in such ways, but we believe that these traditional measurements can be improved upon. The interesting thing for many of these parallel frameworks is the representation of container objects in terms of things that many scientific end users relate to, e.g. Linear Systems, PDEs, Matrices. We believe that this can be utilized in a profiling environment by taking the abstractions a step further and representing performance data in terms of the abstractions, mainly the instantiations of these abstractions in the form of program variables. The unique feature of our system is its ability to automatically combine and map complex internal data structures (such as sparse matrices and non-uniform grids) to higher level concepts.

The core of mapping performance data to the variables rests with the idea of variable "blame." Variable blame is defined as a measurement of performance data associated with the explicit and implicit data flow used to determine the value of a particular variable at various levels of an abstraction's hierarchy. The performance data metric is chosen by the end user based on what profiling information they are interested in, which is in turn based on the available measurable hardware counters for the system. For complex structures or classes that have multiple member variables ranging from other complex types or primitives, it is the aggregation of all the blame from its respective components. Blame is calculated at the lowest level, where data movement is made in the form of individual assignment and arithmetic operations, and then bubbled up the hierarchy of the abstraction.

2 Calculating Blame

The calculation of blame for a given variable is a multistep process utilizing both static and runtime information. When possible, information is gathered statically once to decrease the runtime footprint. The runtime operations are based primarily on information gathered through sampling utilizing hardware counters. The process for calculating blame is discussed in this section and is displayed in Figure 1.

2.1 Data Flow Relationships

The building blocks for variable blame lie in the data flow relationships between variables in a program. Put simply, at the end of a code region we want to examine what variable ultimately contains the product of all the work that went into producing a certain value. There are two types of data flow relationships we are interested in, explicit and implicit.

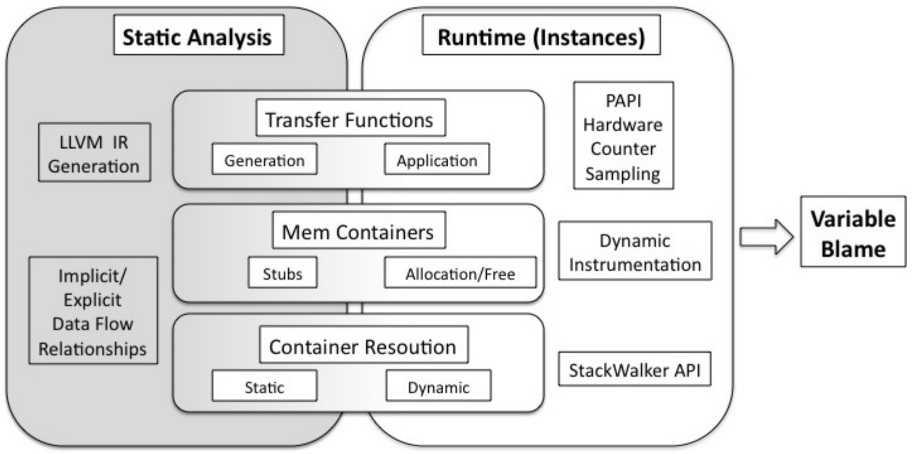

Fig. 1. Blame mapping for main function

Explicit transfers occur on data writes within a program. For example, consider the C snippet below:

```
int a, b, c;
a = 7;
b = 8;
c = a + b;
```

The blame for this snippet would be assigned to variable c. This is because the values of a and b are calculated directly for the purpose of having their sum stored in c. The variable c may then be used in another computation and will subsequently be included in the blame set of the variable that uses it (along with a and b).

Implicit transfer is more complicated and subjective than explicit transfer. It primarily concerns variables that are assigned a value that is never directly assigned to another variable. This occurs often in variables that involve control flow. For example, a loop index is incremented for every iteration of the loop and used in a comparison operation but is never actually assigned in some computation (besides incrementing itself). This is also true in flags for switch statements and standard conditional statements. In these cases, all of the variables implicitly affected by these variables (statements within loop body, conditional branch) will have an implicit relationship with these variables.

Both explicit and implicit relationships are computed statically and information is stored per function. We accomplish this by analyzing an intermediate, three address, representation of the program. We describe memory operations in Section 2.3. For explicit relationships, we build a graph based on the data flow between the variables. For implicit relationships, we use the control flow graph and dominator tree generated to infer implicit relationships for each basic block. All variables within those basic blocks then have a relationship to the implicit variables responsible for the control flow that resulted in the blocks present in that control flow path. For both implicit and explicit relationships, after the calculations are performed on the intermediate format, a mapping is used to relate data back to the original source code.

2.2 Transfer Functions

The data flow relationships are all recorded at the function level and calculated through intraprocedural analysis. For interprocedural analysis, we need a mechanism to communicate the blame between functions. We utilize *transfer functions* for this step. When looking at explicit and implicit blame, we utilize a form of escape analysis[3] to determine what variables, which we deem exit variables, are live outside of the scope of the function. These could be parameters passed into the function, global variables, or return values. All explicit and implicit blame for each function is represented in terms of these exit variables during static (pre-execution) analysis. During runtime, a transfer function is used to resolve the caller side parameters and return containers to the appropriate callee side exit variables.

When source code is not available, manual intervention is needed. For these cases, transfer functions can be created based on knowledge about a procedure's functionality. When faced with a complete lack of knowledge about a function (no source or documentation) a heuristic is used to divide up the blame between the parameters and return values from these functions.

For common library routines such as MPI, we include predefined blame assignments based on the prototype of the functions. For example, if we see a program with a call to MPI_Bcast without doing any additional analysis we can attribute the blame for that function to the variable passed in as the first parameter, the data that is being broadcast.

2.3 Mem-Containers

Up to this point, the discussion has not considered operations involving memory, whether stack or heap allocated. We represent these operations in our mappings with "mem-containers." A mem-container is an abstraction representing a unique contiguous region of memory for the scope of time the memory is available to be accessed by the program. In the case of heap objects, this would be the period of time between allocation and deallocation. For global variables, the period of time would be the length of the program. For structures, classes, and arrays using memory allocated on the stack, the supporting mem-container is active while the function that pushed the data to the stack is still represented on the call stack. It should be noted that mem-containers are not the final blame containers that are presented to the user. There needs to be an additional mapping that takes place to associate the mem-containers with the program variables they represent. Abstractions within frameworks for items such as matrices and vectors may have multiple mem-containers associated with them.

Like transfer functions, static analysis is used to reduce runtime data collection and analysis. For mem-containers, we can determine points in the program where memory allocation and deallocation takes place. At these places in the code, we create stubs signifying that an allocation can be expected at these points when the program is run. At runtime, by instrumenting allocation routines like *malloc*, we gather additional information such as the location in memory the allocation took place and the size of the allocation. Furthermore, we perform a stack walk at each allocation point to determine the call path that led to the allocation. This path information is used to match the allocation to the stubs we had generated through static analysis.

2.4 Container Resolution

Container resolution refers to the resolution of blame within complex data types. For instance, a structure or class may have multiple fields which acquires blame through the course of the program. Some of these fields may themselves be classes. Container resolution is simply the bubbling up of blame until it reaches the top most container type. Much of this resolution can be taken care of statically, though there could be cases where a field may be referenced through a set of pointers where runtime information will be needed to fully attribute blame to the proper container.

2.5 Instance Generation

An instance is an abstraction that represents the operations that occur at a specific sampling point. Since each sample generates a unique instance, the instance will carry with it the blame of the metric that caused the program to generate the sample. The metric is chosen by the user based on what aspect of the program they would like to measure, and typically are of the set of hardware counters available on a given platform. Instances are then mapped up to either mem-containers or program variables for aggregation of blame. In the case that the access pertained to a primitive within the code, the instance will map to a specific program variable and no record of the instance will need to be maintained. In the case the sample occurred during a memory operation, more exhaustive bookkeeping about the instance is maintained and an upmapping is created to its corresponding mem-container. Within each instance are also identifiers that describe which node in a distributed system and/or thread these accesses came from.

This sampling can be done not only at specific time intervals, but can also be based on program events. For example, threshold driver overflow interrupts from hardware counters can be used. The user is allowed to choose what metric(i.e. cache misses, floating point operations) they want to sample and how often the interrupt will be triggered. When the interrupt occurs, our system uses a handler that records the program counter and state of the machine when the overflow occurs. We use this interrupt as an opportunity to do a stack walk at the point of interrupt. This allows us to utilize the transfer functions in a context sensitive matter.

2.6 Final Variable Blame

The final blame for a given variable is presented to the user at "blame points" throughout the program. The majority of blame points are automatically identified through static analysis as places within the program where blame can not be mapped up any further to another variable, the most obvious of these functions being the *main* function. A blame point can also be any point in the program explicitly designated by the user to be a point of interest or that falls into a set of criteria that would make it interesting to examine. One such set of criteria could be having a function with variables that contain more than a certain percentage of the total blame for the program. It should also be noted that although blame propagates up the call stack, not all blame will necessarily make it back to the original *main* function. An example of this, as well as detailing multiple blame points, is shown in Section 3.3.

3 Experimental Results

To implement the concepts introduced in Section 2 we used various components. To generate the intermediate format used to calculate the implicit and explicit relationships, we used LLVM[4]. To perform the stackwalking that assisted in creating a context sensitive represenation of the mem-containers and instances, we used the Stackwalker API[5]. Finally, for generating instances through sampling we used PAPI[6]. We also utilized PAPI for accessing hardware counter information.

To show some of the capabilities that our mapping offers that differ from traditional techniques, we have chosen three programs that directly exhibit properties that would be found in large parallel programming abstractions. For all three programs, the blame metric concerns cycles spent with the sampling triggered every predetermined number of cycles. Therefore, for each sampling point (instance), whatever variable gets the blame for that instance is essentially responsible for the cycles that were used between measured samples. For these experiments, we present the absolute blame numbers that are matched one to one with the samples taken while profiling the program. We also present the percentage of the program cycles that were used in the calculation of the variable according to our blame mappings.

It should be noted that although we used cycles for the blame metric for these tests, once the mappings are in place any measurable metric on the system can be applied. In the case of parallel programs, we can utilize metrics such as blocking time and bytes transferred in cases where MPI calls contribute to the blame.

3.1 FFP_SPARSE

One of the test programs we examined was FFP_SPARSE[7], a small open source C++ program that uses sparse matrices and a triangle mesh to solve a form of Poisson's equation. It consists of approximately 6,700 lines of code and 63 functions. Although this program is sequential, the problem space and data structures utilized make it an attractive case study.

We ran the FFP_SPARSE program and recorded 101 samples which are the basis of the mappings discussed in this section. After removal of the debugging output, the only blame point for this program is the main function, with the program culminating in the output of the final solution vector.

This program does not have complex data structures to represent vectors and matrices, but the variable names for the primitive arrays map nicely to their mathematical counterparts in many cases. Table 1 shows the blame mappings for the variables in main. The "Base Data Centric" column represents explicit memory operations, meaning that the sampling was taken when an assignment was occurring for these arrays. "Blame" refers to the number of samples in which blame was assigned to those variables (whether directly to the variable or the mem-containers that mapped to that variable).

One thing that stands out from this sampling is the lack of sample points (only two) where an explicit write was taking place to the arrays present at the top scope of *main*. This number includes any writes to these memory locations under all aliases as many of these arrays are passed as parameters throughout the program. If one were looking to examine which variables were the most important to the program based solely on this information, which would be the case for standard profiling systems, it would be hard to get any usable information. In a dense matrix, there may be many more reads and writes to actual memory locations tied to the defined variable for the matrix. However, in sparse matrix implementations, many of the computations take place behind layers of abstraction between the defined variable and where the work is actually taking place. When blame mapping is introduced we get a clearer picture of what the program is trying to accomplish. The solution vector and the coefficient matrix are the clear recipients for most of the blame of the program.

Table 1. Variables and their blame for run of FFP_SPARSE

Name	**Type**	**Description**	**Base Data Centric**	**Blame(%)**
node_u	double *	Solution Vector	0	35(34.7)
a	double *	Coefficient Matrix	0	24.5(24.3)
ia	int *	Non-zero row indices of *a*	1	5(5.0)
ja	int *	Non-zero column indices of *a*	1	5(5.0)
element_neighbor	int *	Estimate of non-zeroes	0	10(9.9)
node_boundary	bool *	Bool vector for boundary	0	9(8.9)
f	double *	Right hand side Vector	0	3.5(3.5)
Other	-	-	0	9(8.9)
Total	-	-	2	101(100)

Table 2. Variables and their blame for run of the QUAD_MPI

Name	Type	MPI Call	Blame (per Node)				
			N1(%)	N2(%)	N3(%)	N4(%)	Total(%)
dim_num	int	MPI_Bcast	27(27.2)	90(95.7)	97(84.3)	102(94.4)	316(76.0)
quad	double	MPI_Reduce	19(19.2)	1(1.1)	5(4.3)	5(4.6)	30(7.2)
task_proc	int	MPI_Send	15(15.2)	-	-	-	15(3.6)
w	double*	-	9(9.1)	-	-	-	9(2.1)
point_num_proc	int	MPI_Recv	-	1(1.1)	7(6.1)	-	8(1.9)
x_proc	double*	MPI_Recv	-	2(2.1)	5(4.3)	-	6(1.4)
Other	-	-	3(3.0)	-	-	-	3(0.7)
Output	-	-	6(6.1)	-	1(0.9)	1(0.9)	8 (1.9)
Total		-	99(100)	94(100)	115(100)	108(100)	416(100)

3.2 QUAD_MPI

QUAD_MPI[8] is a C++ program which uses MPI to approximate a multidimensional integral using a quadrature rule in parallel. While the previous program illustrated how a sparse data structure can be better profiled using variable blame, this program helps to illustrate how some MPI operations will be modeled. It is approximately 2000 lines of code and consists of 18 functions.

We ran the QUAD_MPI program on four Red Hat Linux nodes using OpenMPI 1.2.8 and recorded a range of 94-108 samples between the four nodes. As discussed in Section 2.2, all calls to MPI functions were handled by assigning blame to certain parameters based on the prototypes of the MPI programs utilized. The program exits after printing out the solution, represented by the variable *quad*.

The results for the run are shown in Table 2. The variables are listed in descending order based on the total amount of blame assigned across all nodes. For each variable, it is shown whether an MPI operation was a contributing factor, but not necessarily the only source, of the blame. This program spends a majority of its time reading in the data files and processing MPI calls with the computation between those calls minimal. The variable with the most blame, *dim_num*, is due to program input from the master node at the beginning of the program, which causes the three other nodes to create an implicit barrier. The second highest blame count goes to *quad*, which is the variable that holds the output for the program so a high number is to be expected.

3.3 HPL

HPL[9] is a C program that solves a linear system in double precision on distributed systems. It is an implementation of the "High Performance Computing Linpack Benchmark." Unlike FFP_SPARSE, the operations are done on dense matrices. HPL offers a variety of attractive features as a test program for blame mapping. It utilizes MPI and BLAS calls and has wrappers for the majority of the functions from both libraries. While the previous two programs were smaller programs, HPL has approximately 18, 000 lines of code over 149 source files.

HPL is also interesting to examine because it is similar to many parallel frameworks in that MPI communication is completely hidden from the user. This means tracing MPI bottlenecks using traditional profiling techniques may technically give you information about where the bottleneck is occurring. However, that information may be useless because the MPI operations are buried deeply enough in complex data structures that knowing how these bottlenecks affect variables at the top levels of the program is difficult to discover.

We ran the HPL program on 32 Red Hat Linux nodes connected via Myrinet using OpenMPI 1.2.8 and recorded a range of 149-159 samples between the nodes. The results for the run are shown in Table 3. This program differs from the other two in that we have multiple blame points. Two of these blame points would be explicitly generated. The other two are user selected and contain variables (A and $PANEL$) that have a large amount of blame associated with them. The *main* function serves primarily to read in program specifications and iterate through the tests, which have their own output. For this reason, only the computation associated with producing the computation grid is actually attributed to it while one its called functions (*HPL_pdtest*) has a much larger stake of the total blame.

In terms of the variables themselves, two different blame points, *mat* and A (which is a pointer to *mat*), are assigned the majority of the blame. This is

Table 3. Variables and their blame at various blame points for run of HPL

		Blame over 32 Nodes	
Name	**Type**	Node Mean(Total %)	Node St. Dev.
All Instances	-	154.7(100)	2.7

main

grid	HPL_T_grid	2.2(1.4)	0.4

main→HPL_pdtest

mat	HPL_T_pmat	139.3(90.0)	2.8
$Anorm1$	double	1.4(0.9)	0.8
$AnormI$	double	1.1(0.7)	1.0
$XnormI$	double	0.5(0.3)	0.7
$Xnorm1$	double	0.2(0.1)	0.4

main→HPL_pdtest→HPL_pdgesv

A	HPL_T_pmat *	136.6(88.3)	2.9

main→HPL_pdtest→HPL_pdgesv→HPL_pdgesv0

$PANEL \rightarrow L2$	double*	112.8(72.9)	8.5
$PANEL \rightarrow A$	double *	12.8(8.3)	3.8
$PANEL \rightarrow U$	double *	10.2(6.6)	5.2

intuitive since A contains a pointer to the local data for the matrix being solved as well as the solution vector and some book keeping variables. Going deeper down the call trace, we find the variable *PANEL* which is a structure with three fields that carry a large portion of the blame of the program. These variables are the storage containers for the computations in the LU factorization and through container resolution it can be presented to the user as the single variable *PANEL* or as separate fields. The blame is assigned locally to these three variables and through a transfer function is then assigned to A in the caller function.

All the variables in Table 3 had contributions to their blame total from MPI operations that occurred within HPL wrappers far down the call stack. HPL is a well established benchmark so one can see that the decomposition is very efficiently done with each variable being calculated almost equally across the processors. An interesting caveat to this balance can be found when looking at the fields within *PANEL(L2,A,U)*, which have many MPI calls contributing to the blame for these variables. For these nodes, there is some imbalance when calculating some of these individual components but whatever imbalance occurs immediately disappears when these variables have their blame absorbed by A one level up on the stack. This is an interesting result, as standard profiling programs might be concerned with the apparent bottlenecks occurring for these lower operations when in fact the calculation of the solution vector (a field in A) is calculated with excellent load balance between the nodes.

4 Related Work

The main area of related work deals with creating mappings in parallel programs at different levels of abstraction. Irvin introduces concepts involving mapping between levels of abstraction in parallel programs with his NV model[10] as utilized by the ParaMap[11] tool. The Semantic Entries, Attributes, and Associations(SEAA)[12], a followup to the NV model, addresses some of the limitations of the NV model and adds some additional features. The SEAA mappings are utilized in the TAU performance tool.[13]. The primary difference between our mapping and these is the way the performance data is collected and transferred. In NV and SEAA, regions of code are measured and mapped up different levels of abstraction with these regions eventually being mapped to variables in some cases. Our approach maps to variables at every level of abstraction and uses data flow operations as the primary mechanism for transferring performance data. Using data flow allows us to push much of the computation to static analysis, with runtime information supplementing that data whereas NV and SEAA focus primarily on runtime information. Finally, we use sampling as our primary technique for generating data versus delimited instrumentation of a code region and measurements taken of that region.

Profiling tools that utilize sampling are the most similar to our approach. These tools include prof[14], gprof[15], DCPI[16], HPCToolkit[17], and Speedshop[18]. Similar instrumentation based profiling tools include TAU[13] and SvPablo[19]. Like the mappings discussed above, the primary focus of these tools is code regions and base data structures.

Dataflow analysis is utilized in many areas. Guo et al. [20] dynamically records data flow through instrumentation at the instruction level to determine abstract types. This is similar to the explicit data flow relationships we use in our blame analysis. Other work dealing with information flow in control flow statements is similar to the implicit data flow relationships we use[21,22].

5 Conclusions and Future Work

In this paper, we have outlined a style of variable mapping called blame mapping. Blame mapping looks at the explicit assignments made to a variable directly, but also concerns itself with all of the computation that went into any variable that is eventually assigned to that variable. The computation of blame mapping takes place partly in an intraprocedural manner to determine the exit variables for a function and how much blame each exit variable is assigned. These exit variables are then combined into a transfer function so an interprocedural analysis can be constructed. This interprocedural analysis is combined with runtime information obtained by sampling to create a repository of information from a program run that can be used for performance analysis or debugging.

The target applications for these types of mappings are large parallel programs with many levels of abstraction, specifically large scientific frameworks. Their abstractions are often tied to mathematical constructs so a representation of performance data in terms of variables may simplify the analysis process. Furthermore, they are often large, long running parallel programs so the less intrusive footprint introduced by sampling is desirable.

Our current experiments have involved smaller programs that exhibit similar properties to those larger programs described above. The primary focus of our future work will involve applying our mapping to these complex abstractions and the larger programs that utilize them. This paper has included profiling data attributed to data structures that is intuitive to where the blame should lie and serves as a sanity check that this style of mapping can produce accurate results. In more complex data structures, these mappings will not be as intuitive. For our future work, we will provide detailed case studies with these large frameworks and how to utilize the mapping information. Furthermore, we will provide a quantitative comparison between results given from blame mapping and traditional profiling tools.

References

1. POOMA, `http://acts.nersc.gov/pooma/`
2. Balay, S., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page (2001), `http://www.mcs.anl.gov/petsc`
3. Deutsch, A.: On the complexity of escape analysis. In: POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 358–371. ACM, New York (1997)

4. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization, CGO 2004 (2004)
5. Univ. of Maryland, Univ. of Wisconsin: StackWalker API Manual. 0.6b edn. (2007)
6. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters, pp. 65–65 (2000)
7. FFP_SPARSE, `http://people.scs.fsu.edu/~burkardt/cpp_src/ffp_sparse/`
8. QUAD, `http://people.sc.fsu.edu/~burkardt/c_src/quad_mpi/`
9. HPL, `http://www.netlib.org/benchmark/hpl/`
10. Irvin, R.B.: Performance Measurement Tools for High-Level Parallel Programming Languages. PhD thesis, University of Wisconsin-Madison (1995)
11. Irvin, R.B., Miller, B.P.: Mapping performance data for high-level and data views of parallel program performance. In: International Conference on Supercomputing, pp. 69–77 (1996)
12. Shende, S.: The Role of Instrumentation and Mapping in Performance Measurement. PhD thesis, University of Oregon (2001)
13. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (2006)
14. Graham, S.L., Kessler, P.B., McKusick, M.K.: An execution profiler for modular programs. Softw., Pract. Exper. 13(8), 671–685 (1983)
15. Graham, S.L., Kessler, P.B., McKusick, M.K.: gprof: a call graph execution profiler. In: SIGPLAN Symposium on Compiler Construction, pp. 120–126 (1982)
16. Anderson, J., Berc, L., Dean, J., Ghemawat, S., Henzinger, M., Leung, S., Sites, D., Vandevoorde, M., Waldspurger, C., Weihl, W.: Continuous profiling: Where have all the cycles gone (1997)
17. Mellor-Crummey, J.M., Fowler, R.J., Whalley, D.B.: Tools for application-oriented performance tuning. In: International Conference on Supercomputing, pp. 154–165 (2001)
18. SGI Technical Publications: SpeedShop User's Guide
19. De Rose, L., Zhang, Y., Reed, D.A.: SvPablo: A multi-language performance analysis system. In: Puigjaner, R., Savino, N.N., Serra, B. (eds.) TOOLS 1998. LNCS, vol. 1469, pp. 352–355. Springer, Heidelberg (1998)
20. Guo, P.J., Perkins, J.H., McCamant, S., Ernst, M.D.: Dynamic inference of abstract types. In: ISSTA 2006: Proceedings of the 2006 international symposium on Software testing and analysis, pp. 255–265. ACM, New York (2006)
21. McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: PLDI 2008, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 9–11, pp. 193–205 (2008)
22. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. 4(2-3), 167–187 (1996)