Tracking a Value's Influence on Later Computation

Philip C. Roth

Oak Ridge National Laboratory, Oak Ridge, TN, 37831 USA

Abstract. Understanding how a program behaves is important for effective program development, debugging, and optimization, but obtaining the necessary level of understanding is usually a challenging problem. One facet of this problem is to understand how a *value* (the content of a variable at a particular moment in time) *influences* other values as the program runs. To help developers understand value influence for their programs, we are developing a tool that allows a user to tag a value as being of interest, and then track the influence of that value as it, or values that were derived from it, are used in later computation, communication, and I/O. We believe that understanding how a value's influence propagates will enable algorithm designers to more easily identify optimizations such as the removal of unnecessary computation and communication. In this paper, we describe our value influence tracking approach and our tool's design and implementation status.

1 Introduction

Understanding how a program behaves is necessary for effective program development, debugging, optimization, and fault tolerance. One facet of this problem is the challenge of understanding how a program's *values* are propagated through time and space as the program runs. For instance, one of a program's input values may contribute to nearly all of a program's output values, whereas another may contribute to very few (or none). We call this contribution the value's *influence* on other values. Knowing the degree of a value's influence can provide insight into how to optimize a program. For example, N-body algorithms often include the notion of a *cut-off distance*. If two entities are further apart than the cut-off distance, the algorithm does not consider the effects of either entity on the other, reducing the total amount of computation and computation required by an implementation of the algorithm. The results of a value influence analysis can

Support for this work was provided through the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

identify not only that this type of optimization is feasible, but can also suggest what a good cut-off distance should be by indicating when the value's influence becomes small enough that it can be ignored in the computation without causing too great of an error in the algorithm's result. In general, value influence analysis can help a developer determine how much detail needs to be included in a model to balance its fidelity with the cost of computing the model. We hypothesize that many developers of scientific aplications do not know with certainty the best level of detail to include in their model, and thus we expect there are often opportunities for optimizations that give results that are still "good enough." In addition to performance optimization, a value influence analysis may also inform decisions about fault tolerance by allowing a developer to know how *unintended* changes to a value can impact the program's results. If a value's influence over later computation is large, the developer may wish to protect the value with a strong resilience strategy.

To facilitate the understanding of value influence propagation, we are developing an empirical approach for tracking value influence as a program executes, and developing a prototype tool that implements this approach. In this approach, a value may be tagged at some point during the program's execution as being a value of interest. After a value is tagged, the approach tracks how the tagged value is used in later computations and how it propagates to other program address spaces. When a tagged value is used in a computation, its influence data is combined with that of other inputs to the computation and the resulting influence value is associated with the computation's output value. For instance, when the statement $c = a + b$ executes, if the influence associated with variables a and b are 0.5 and 0.2, respectively, then the influence associated with the output value c would be 0.35 assuming the value influence analysis uses the "average" function to combine influences. Although this example is presented at the statement level, we track value influence at the machine instruction level to reflect the computation that is actually executed on the system and to avoid the complexities of a source code-level analysis. Our initial work on value influence tracking targets scientific and engineering applications, and our approach supports applications that use the Message Passing Interface [2, 3] (MPI) for communication and synchronization, and multithreading technologies like OpenMP [8].

2 Value Influence

In this section, we detail the concept of value influence and discuss the challenges in propagating value influence as a program executes.

2.1 Assigning and Combining Value Influences

To track value influence, we maintain a set of pairs (L, I), where I is the influence associated with the current value held at location L, where L can represent either a location in memory or a register. Whenever the value held at a location L_k is used in an operation, its associated influence I_k is combined with the

influence values of the other operation inputs, and the resulting influence I_{output} is associated with the value at the output location L_{output} in our set of location-influence pairs. Influence values I are real-numbered values in the interval $[0, 1]$. When a value is tagged as being of interest, it is assigned an influence of 1. To control the storage requirements for our approach, if a location is not included in our set the influence of its current value is assumed to be zero.

When a value is used as an input to an operation, its influence must be combined with that of the other inputs and associated to the output value using some combining function, F. The simplest way to combine influences is to use a logical OR operation, such that the influence associated with the output value is 1 if any of the input values have a non-zero influence. With this combining function, the value influence analysis indicates which values depend (directly or indirectly) on the value(s) initially tagged as being of interest, but cannot tell how much the initial value contributed to later values. Combining functions that take into account the operation being performed provide a more nuanced view of value influence propagation. For instance, for a **sum** instruction the combining function may average the influences of the input values. In contrast, for a **max** instruction (available in some instruction set architectures such as Intel's Streaming Single Instruction Multiple Data Extensions), the combining function may use the influence associated with the maximum value as the influence of the output. As an example, if a program adds two variables u and v, u's value has an associated influence of 0.3 and v's value has an influence of 0.5, the value produced as the output of the instruction and stored in a destination *dest* is assigned an influence of 0.4, the average of the influences of u and v. A very sophisticated combining function might take into account not only the type of operation but also the input values themselves. For instance, if the two input values for a **sum** instruction are 1×10^9 and 0.5, the combining function may give the first value's influence much more weight than the second's influence.

2.2 Variable Lifetime

Our value influence tracking approach associates an influence with a memory location or register. Because the number of registers is limited, and because frames are added and removed dynamically from the stack as a thread runs, it is common that two or more variables share the same memory location or register at different moments during a program's execution. Compilers use the concept of *variable lifetime*—the execution interval during which its value is still needed for some future operation—to determine whether two variables can share the same register or memory location. When two variables share the same register or memory location, and the influence of at least one of them is being tracked, our approach takes variable lifetime into account so that it does not mistakenly use a dead variable's influence data instead of that of the live variable that shares its register or memory location. Information about variable lifetime is not explicitly present in the code generated by most compilers, but our approach correctly determines when to "forget" the influence data of a dead variable for most code, because the influence data associated with the dead variable's value

is replaced when live value is initialized. If the value used to initialize the live variable has no associated influence data, the influence data associated with the now-live variable is deleted. The only situation where our value influence tracking approach does not handle influence propagation correctly as a natural by-product of the code's initialization of live registers is when the code uses a register or memory location without initializing it.

2.3 Tracking Influences for Multiple Initial Values

Tracking the influence of multiple values within the same run of the program can be accomplished with minor additions to the approach as presented so far. We differentiate between two multiple-value scenarios: one where the program's run is separated into a sequence of phases, and another where the influences of multiple values are tracked simultaneously.

We support value influence tracking for one value per program phase by providing mechanisms (such as application programming interface [API] calls) that allow access to the current value influence data and that can reset the value influence data in preparation for the next phase.

To support tracking the influence of multiple values simultaneously, a "color" is associated with each influence value. The influence data associated with a location is now a set of $(color, influence)$ pairs. When a value is first tagged as being of interest, its initial influence is assigned a color that is unique within the current phase. Then, when an operation uses input variables to produce an output variable, the combining function combines the influences for each color individually to produce the influence data vector associated with the output value. For instance, if the influences associated with the values of variables u and v are $\{(0, 0.2), (1, 0.4), (2, 0.7)\}$ and $\{(0, 0.4), (1, 0.5)\}$, respectively, the influence associated with the output value when u and v are summed is $\{(0, 0.3), (1, 0.45), (2, 0.35)\}$ (assuming the influence combining operator is the "average" function). If the input does not have an associated influence for a given color (e.g., v does not have a color 2 influence), it is assumed to be zero. Thus, in the example, the color 2 influences of 0.7 and 0 are averaged to give 0.35, and this is assigned as the color 2 influence on the result.

Because of the need to store multiple influences per location, and the need for the combining function to combine influences for each color independently, tracking the influences of multiple values per phase can add a significant overhead in both space and time to the basic value influence tracking approach. The actual time and space requirements depend on the number of initial values being tracked and the number of times those values, and values derived from them, are used in operations during the computation.

2.4 Multithreaded Programs

Multithreaded programs present an interesting set of challenges when tracking variable influence. In a multithreaded program, a value in a memory locations is shared among all of the program's threads. Any thread may read or write

that value whenever the thread is executing on a processor core. To avoid race conditions in accessing this influence data, we use a mutual exclusion lock (also known as a mutex) to protect the internal data structure that maintains influence variables for values in memory. For values in registers, we use thread-local storage so that we are sure to use the correct influence data for a register as threads are switched onto and off of processor cores.

2.5 MPI Programs

We aim to support value influence tracking not only for multithreaded programs involving a single address space, but also for MPI programs involving multiple MPI processes each with its own address space. Such programs use MPI API functions to transfer data from one address space to another and to synchronize the program's processes. Our approach to support MPI programs require the ability to detect when the program transfers data, and the ability to modify the behavior of that transfer. The MPI standard defines a *profiling API* that provides both of these features. This interface allows link-time interposition of a function between the caller of an MPI function like `MPI_Send` and the library's implementation of that function. By defining a collection of interposition functions corresponding to MPI's data transfer functions, we can detect when the application is sending data from one address space to another using MPI those functions. Our interposed versions of the MPI data transfer functions use the underlying MPI data transfer functions to transfer both the application-level data and any influence values associated with that data. Two-sided and collective operations use the same two-sided and collective operations, respectively, to transfer the influence data as the application data. One-sided MPI operations require the use of a separate "value influence service thread" (VISTs) to run in each MPI process for maintaining and serving the value influence data, plus an out-of-band channel for transferring influence data between interposed MPI one-sided functions like `MPI_Put` and these VISTs.

3 Implementation

We are implementing the value influence tracking approach described in Sect. 2 in a tool called *VIT*. In this section, we discuss some of the *VIT* implementation details.

When tracking value influence, we associate influence data with a memory address (for values in memory) or a register ID (for values in registers). This approach gives us a concise way to represent locations, but because memory addresses and register IDs are from distinct name spaces, this approach requires us to keep separate collections of location-influence pairs for memory locations and registers. In our current *VIT* implementation, we use the C++ standard template library's `map` class to hold location-influence pairs. Because a given CPU core's registers are accessible only to the thread executing on that core,

we use thread local storage when associating a register-influence map with each thread.

VIT is built on the Intel Pin instrumentation meta-tool [7]. Pin allows developers to create tools (commonly called *Pintools*) that analyze and instrument the machine code of an x86 or x86_64 executable program. Pin uses *dynamic instrumentation*—the instrumentation code is added while the program is executing. The instrumentation code is Pintool-specific. For instance, the instrumentation for an instruction mix tool would analyze each instruction as it was executed and count the number of instructions of each type. In contrast, the instrumentation for a simple performance tool may count the number of times each function is called and to measure how long it takes to execute each function.

VIT uses Pin's trace instrumentation mode to insert the bulk of its instrumentation code. When Pin gives control to *VIT*'s trace-level instrumentation because a new trace of instructions is about to be executed, *VIT*'s trace-level instrumentation examines each instruction in the trace to determine whether the instruction writes a value to a destination location (a memory location or register). If so, the *trace-level* instrumentation code inserts *instruction-level* instrumentation for that instruction. This instruction-level instrumentation computes the value's new influence as described in Sect. 2.1 and associates the new influence data with the destination location in the appropriate location-influence map.

A user must specify to *VIT* which values are of interest for tracking their influence, and when to output the *VIT* influence data. *VIT* provides a simple C-based Application Programming Interface (API) for user control. The user can specify a value of interest using the API function `VIT_Track` by inserting a call to this function into the application's source code such that it will be executed when the user wants to start tracking the influence of that variable's value. For instance, to start tracking the influence of a variable's value in the tenth iteration of the application's main loop, the user might insert a call to `VIT_Track` as the body of an if-then statement whose condition is that the iteration number is 9 assuming iteration numbers start from zero. This approach to specifying for which values to track influence, and when to start tracking, provides flexible control over when to start tracking but requires the user to insert calls to `VIT_Track` in their application.

In our prototype implementation, *VIT* outputs the value influence data to standard output when the program completes. *VIT* also provides API functions `VIT_Report` and `VIT_Reset` for outputting the value influence data at other times and for resetting the data, e.g., to restrict value tracking to a certain phase or particular iteration of a loop.

4 Status

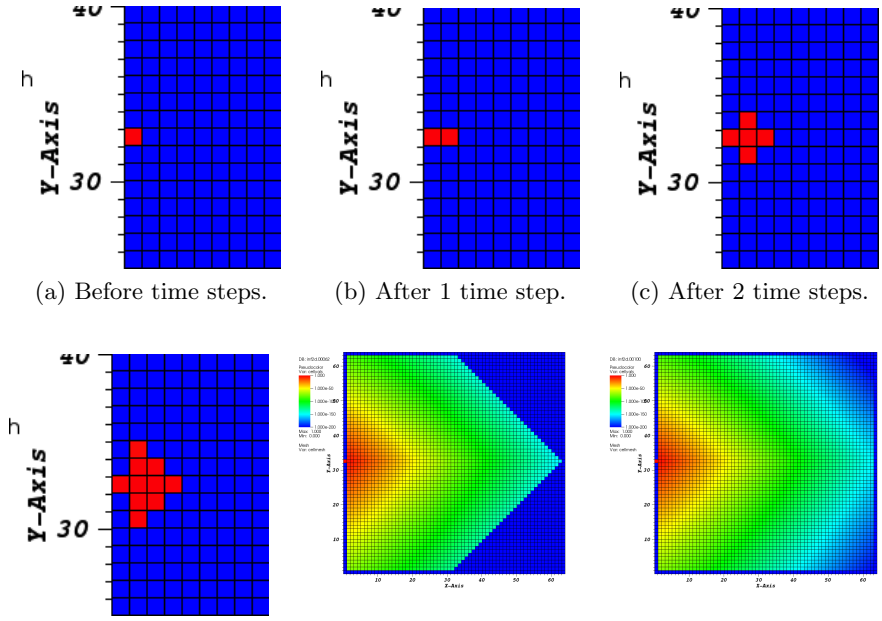
We have implemented the basic intra-process *VIT* functionality and are currently implementing inter-process influence tracking functionality for two-sided and some collective MPI data transfers. For our prototype implementation, we

are using the *average* operation when combining influences, regardless of the instruction's operation.

To demonstrate *VIT*'s current functionality, we applied it to a self-written application that solves the two-dimensional heat equation $\partial T/\partial t = \alpha \nabla^2 T$ (where T is the temperature of the material and α is its thermal diffusivity) using an explicit discretization and forward time-centered space (FCTS) method [10] with a five point stencil. The program keeps the problem state in two 2-dimensional arrays of double precision values where each value represents a discretization point of the problem domain. One array holds the current state and its values are used to compute the updated state in the other array. This implementation is single threaded. We tracked the influence of one of the problem domain boundary values as the program executed its time step loop. Because the program uses a five-point stencil, the value's influence "diffuses" through the problem state array by advancing one row and column per time step similar to the way that heat diffuses throughout the problem domain as the simulation proceeds.

The bulk of *VIT*'s output is a sequence of $(address, influence)$ pairs for memory locations with non-zero influence values. If the program contains a symbol table, each address can then be translated to a symbolic name of a corresponding program variable using software like the GNU Binary File Descriptor (BFD) library or SymtabAPI [9]. (Our prototype implementation does not perform this translation, nor does it report influence values associated with registers.) Figure 1 shows visualizations of this *VIT* output for select time steps produced as influence was propagated through the application's state arrays. In the figure, each cell represents a memory location in the array. Each cell is colored according to the influence value associated with the cell's associated memory location. We wrote a simple utility to convert from the addresses in *VIT* output to the indices of cells within a two dimensional array. The influence values quickly decrease from the initial value of 1, so the visualizations use a log scale to retain contrast between the cells representing memory locations with non-zero influence and those with zero influence. Because the heat transfer application uses a double buffering approach, the value influence "wavefront" alternates between the two problem state arrays. Thus, for the figure we display value influence data for one array after even numbers of time steps and the other after odd numbers of time steps. As expected, the program took 62 time steps for the influence data to propagate across the 64-element-wide problem state array. (The array boundaries are one element wide and immutable). Note that this influence propagation pattern is independent of the heat transfer problem being solved.

Applying *VIT* to our example heat transfer application exposed three problems with the prototype implementation's intra-process value influence tracking. First, the influence values drop rapidly as the computation proceeds. We determined that our prototype implementation does not handle register-based addressing modes in the expected way, treating them as uses of a register in the computation of the instruction's result. Although this interpretation is technically correct, we are modifying *VIT* to distinguish between registers used for addressing and registers providing source values for the computation. Second, *VIT*



(a) Before time steps. (b) After 1 time step. (c) After 2 time steps.

(d) After 3 time steps. (e) After 62 time steps. (f) After 100 time steps.

Fig. 1. Value influence propagation for two-dimensional heat transfer application, starting with a single value on the boundary of the problem state array.

does not currently support Single Instruction Multiple Data (SIMD) instructions that operate on multiple values. However, because the SIMD arithmetic units tend to be faster than traditional x87 floating point hardware, modern compilers often generate code for the SIMD units even when operating on single values. The executable code produced by GCC for our example application uses this approach, and our prototype *VIT* implementation handles it correctly. Finally, our prototype *VIT* implementation does not correctly propagate influence data for instruction sequences that use the traditional x87 floating point hardware. Although we have not yet identified the root cause of this problem, doing so is a low priority in light of the common use of SIMD hardware by modern compilers.

5 Related Work

Program slicing [12, 13, 6, 14] is a technique for determining which statements were used to produce the value of a given variable at a point in the program's execution (backward slicing) or for determining the statements that will directly or indirectly use the value of a given variable at some future point in the program's execution (forward slicing). Our value influence tracking approach is similar to forward program slicing in that it focuses on the value of a variable at a specific point in the program's execution, and that both are concerned with

how that value is used in the future computation. Unlike traditional forward program slicing that is concerned with determining which future statements use the value (a code-centric view), our value influence tracking approach takes a data-centric perspective by determining which other values the initial value influences. Also, unlike the traditional analysis, our approach is concerned not only with *whether* a value had influence on future values, but also *how much* influence it had. Weiser's original program slicing approach [13] uses a static analysis of the program's source code that produces a conservative result containing all statements that *might* be executed for a given variable, regardless of the program's inputs. Dynamic slicing approaches [6] limit the output to only those statements that are actually executed for specific inputs. In that respect, our empirical value influence tracking approach is more similar to the dynamic program slicing approaches than the traditional static slicing approach. A forward program slicing approach could be used by a value influence tracking tool like *VIT* to limit the time required to perform the value influence analysis, since executing the program slice would not execute statements (and any associated *VIT* instrumentation) that could not possibly transfer influence.

Chopping [4, 11] is the technique derived from slicing that is closely related to our value influence tracking approach and *VIT* tool. For a given set of source variables and a set of sink variables, chopping is concerned with determining the set of statements that use the values of the source variables to produce the values of the sink variables. In our value influence tracking approach, we do not restrict the influence tracking to a particular set of sink variables. Also, the original chopping approach [4] supported only source and sink variables within the same function. This restriction was overcome by Reps et al [11]. Like Reps' work, our approach is able to track value influence across function boundaries.

Taint Analysis [5, 1] is a computer security technique for tracking data that is "tainted" as it propagates through a computation, similar to the way we track a value's influence. When a tainted value is used to produce another value the resulting value is also considered tainted, so that taint is propagated through the computation as it progresses. Kang et al [5] present a *Dynamic Taint Analyis* technique that tracks taint using an infrastructure similar to Pin, and Ganai et al [1] track taint for multi-threaded programs. These approaches are similar to our value influence tracking approach in the way that they propagate taint information, but differ from our approach in that the taint analyses consider taint as a boolean characteristic: either a value is tainted, or it is not. In our case, we quantify the amount of influence that is transferred when values of interest are used to compute a new value. Also, this existing work does not consider analyses for programs running on distributed memory systems such as the compute nodes of an HPC system.

6 Summary

In this paper, we presented the concept of value influence tracking and the design of a tool, *VIT*, that implements that concept. We described the challenges and an

approach for tracking value influence in multi-threaded programs, and in MPI-based parallel programs. We described our prototype *VIT* implementation that supports value influence tracking and demonstrated it with a simple scientific application.

Once we complete *VIT*'s basic intra- and inter-process functionality, we plan to evaluate the trade-offs of using more sophisticated influence combining operations such as those described in 2.1. We will also add the support for more MPI collectives, and support for MPI one-sided data transfers. We also plan to evaluate whether and how well this general influence tracking approach applies to other parallel programming models than MPI, and whether and how it can be applied to programs that use accelerators like graphics processing units (GPUs).

References

1. Ganai, M., Lee, D., Gupta, A.: DTAM: dynamic taint analysis of multi-threaded programs for relevancy. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 46:1–46:11. FSE '12, ACM, New York, NY, USA (2012)
2. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message-passing interface. Scientific and engineering computation, MIT Press, Cambridge, MA, 2nd edn. (1999)
3. Gropp, W., Thakur, R., Lusk, E.: Using MPI-2: Advanced Features of the Message Passing Interface. MIT Press, Cambridge, MA, USA (1999)
4. Jackson, D., Rollins, E.J.: Chopping: A generalization of slicing. Tech. rep., Pittsburgh, PA, USA (1994)
5. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic taint analysis with targeted control-flow propagation. In: NDSS. The Internet Society (2011)
6. Korel, B., Laski, J.: Dynamic slicing of computer programs. J. Syst. Softw. 13(3), 187–195 (Dec 1990)
7. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. 40(6), 190–200 (Jun 2005)
8. OpenMP Architecture Review Board: OpenMP application program interface, version 3.1. OpenMP Architecture Review Board (July 2011)
9. Paradyn/Dyninst Groups, University of Wisconsin-Madison and University of Maryland: SymtabAPI programmer's guide, 8.1 release (Mar 2013)
10. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes 3rd Edition: The Art of Scientific Computing. Cambridge University Press, New York, NY, USA, 3 edn. (2007)
11. Reps, T., Rosay, G.: Precise interprocedural chopping. SIGSOFT Softw. Eng. Notes 20(4), 41–52 (Oct 1995)
12. Weiser, M.: Program slicing. In: Proceedings of the 5th international conference on Software engineering. pp. 439–449. ICSE '81, IEEE Press, Piscataway, NJ, USA (1981)
13. Weiser, M.: Program slicing. Software Engineering, IEEE Transactions on SE-10(4), 352–357 (1984)
14. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes 30(2), 1–36 (Mar 2005)