To appear: 3rd International Workshop on Automatic Differentiation Tools and Applications, May 2006. (c) Springer.

Hybrid Static/Dynamic Activity Analysis*

Barbara Kreaseck[1], Luis Ramos[1], Scott Easterday[1], Michelle Strout[2], and Paul Hovland[3]

[1] La Sierra University, Riverside, CA
[2] Colorado State University, Fort Collins
[3] Argonne National Laboratory

Abstract. In forward mode Automatic Differentiation, the derivative program computes a function f and its derivatives, f'. Activity analysis is important for AD. Our results show that when all variables are active, the runtime checks required for dynamic activity analysis incur a significant overhead. However, when as few as half of the input variables are inactive, dynamic activity analysis enables an average speedup of 28% on a set of benchmark problems. We investigate static activity analysis combined with dynamic activity analysis as a technique for reducing the overhead of dynamic activity analysis.

1 Introduction

In forward mode Automatic Differentiation (AD), the derivative program computes a function f and its derivatives, f'. Activity analysis [5, 8, 12, 10, 7] determines which temporary variables lie along the dependence chains between inputs and outputs of the function f. When only a subset of the inputs and outputs are being studied, activity analysis can be used to identify an associated subset of local variables that are defined and used along the dependence chains from the independents (inputs of concern) to the final calculation of the dependents (outputs of concern).

Activity analysis has the potential to significantly reduce the number of calculations needed to produce the dependents from the independents. Unfortunately, static activity analysis (done at compile time) may in the presence of control flow be too conservative. On the other hand, dynamic activity analysis (done at runtime) may introduce a significant amount of overhead.

In this paper, we quantify the overhead of performing dynamic activity analysis on a number of benchmarks. Our results show that when all variables are active, the runtime checks required for dynamic activity analysis incur a significant overhead. However, when as few as half of the input variables are inactive, dynamic activity analysis enables an average speedup of 28% on a set of benchmark problems.

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational Technology Research, U.S. Department of Energy under Contract W-31-109-Eng-38 and by the National Science Foundation under Grant No. OCE-020559.

```
void f(double x, double &y,
       double z)
{
   double a, c;

   a = z * z;
   c = x * 9;

   y = a * c;
}
```

Fig. 1. Function

```
void fprime(double x, double dx,
            double &y, double &dy,
            double z, double dz)
{ /* dx = 1, dz = 0 */
   double a, c, da, dc;

   a = z * z;
   da = dz*z + dz*z;
   c = x * 9;
   dc = 9 * dx;
   y = a * c;
   dy = da*c + dc*a;
} /* dy = ∂y/∂x */
```

Fig. 2. Derivatives

In Section 2 we provide the motivation for our studies. In Section 3 we present currently available activity analysis and our extensions. Next, we present our study of the overhead of dynamic activity analysis in Section 4. In Section 5 we present our hybrid static/dynamic analysis. Finally, we discuss future work and conclude in Section 6.

2 Motivation

We demonstrate the importance of activity analysis to AD with the following examples. In Figure 1, we show an example function `f` with an input variables `x` and `z` and an output variable `y`. AD would generate the derivative code shown in Figure 2 to calculate the derivative of `y` with respect to `x` (where we represent $\partial y/\partial x$ as just the variable `dy`). Activity analysis is applied to the original program and determines which temporary variables lie along the dependence chain between independent variables (a subset of the inputs) and dependent variables (a subset of the outputs). In the example, local variable `c` is active while local variable `a` is not. Variable `a` is inactive because it does not depend upon the value of `x`. Variable `c` is active because it depends upon the value of `x` and is used to compute the value of `y`. Activity information enables an AD tool to avoid generating the code that has been crossed out in Figure 2. In real applications, one typically uses the vector mode of AD** and the variables `da`, `dc`, and `dy` are arrays. Furthermore, the update `dy = a*dc + c*da;` becomes

```
for(i=0;i<nindeps;i++)
   dy[i] = a*dc[i] + c*da[i];
```

Thus, activity analysis offers the opportunity for substantial savings, especially when the number of independent variables is small.

Activity analysis can be performed within a data-flow analysis framework. Unfortunately, due to control-path uncertainty, not all variables can be statically

** For simplicity, we restrict our discussion to the forward mode of AD. In the reverse mode, activity analysis offers substantial savings opportunities through reduced storage requirements.

```
bool flag;
double g, z;
void f2(double x, double &y)
{
    double a,b,c;
    if (flag) {
        a = g * z;
    } else {
        a = x * x;
    }
    c = x * 9;
    y = a * c;
}
```

Fig. 3. Example function, `f2`, where the control-path is not known at compile time. When `flag` is true, local variable `a` will be inactive. When `flag` is false, local variable `a` will be active.

identified as active or inactive. Consider the function `f2` in Figure 3. The control-path through `f2` is not decidable at compile time. Now, `a` will be active if `flag` is false and it will be inactive if `flag` is true. Statically we can characterize `a` as active to be conservative but this would result in more work than necessary. The amount of unnecessary work depends upon the number of independent variables that were selected when the derivative code was produced. Specifically, for `f2`, that would just be one (just `x`). For derivative code in general, that will probably not be the case.

We address the problem of activity analysis in the presence of control flow by characterizing `a` as *may active* and augmenting the derivative code to check the activity of `a` dynamically during run-time. This technique is called *dynamic analysis* or run-time analysis. Specifically we associate a boolean flag with each gradient vector (e.g., `da`) to indicate whether it is active or not.

A naive approach is to just use dynamic activity analysis on all variables. This involves the overhead of checking the active flag before every derivative computation. In the next section, we discuss current activity analysis implementations, along with our extensions. In Section 4 we will see that the overhead of dynamic activity analysis can be quite high. Thus, we describe a hybrid static/dynamic approach to activity analysis in Section 5.

3 Dynamic Activity Analysis and AD

Our work with activity analysis is based upon two AD tools: ADIC [6, 3] for C codes, and OpenAD [13] for Fortran codes. The following subsections discuss activity analysis with each tool.

Dynamic Activity Analysis in ADIC ADIC 1.2 does not perform static activity analysis. All floating-point variables are treated as active unless they are specifically designated as inactive by the user. The generated derivative code will include calls to `axpy` routines, which implement the process of combining gradient vectors and local partial derivatives according to the chain rule of calculus. The `axpy` routines are implemented as C preprocessor macros. ADIC 1.2 provides a set of macros that implements dynamic activity checking. These

Table 1. Characteristics of the Fortran benchmarks and dynamic overhead

Benchmark	Independent	Dependent	Size	Source	Overhead
bminsurf	x	f, fgrad	n = 400	NEOS	1.32
daerfj	x	fvec, fjac	n = 4	Minpack2	1.30
datrfj	x	fvec, fjac	n = 3	Minpack2	1.74
dchqfj	x	fvec, fjac	n = 11	Minpack2	1.68
dedffj	x	fvec, fjac	n = 5	Minpack2	1.51
dodcfg	x,lambda	f, fgrad	n = 20x20	Minpack2	1.24
dsfdfj	x,eps	fvec, fjac	n = 280	Minpack2	1.01
dsscfg	x,lambda	f, fgrad	n = 20x20	Minpack2	1.31

macros augment the gradient vectors with an activity flag and check (and set) the activity flags in the `axpy` routines.

At runtime, initially the activity flags of all independent variables are set to true and of all other inputs are set to false. The gradient accumulation macros check the activity flag of each gradient vector prior to execution to affect the following:

- When a right-hand-side gradient vector is active, its values contribute to the calculation of the left-hand-side gradient vector and the activity flag of the left-hand-side gradient vector is set to active.
- When a right-hand-side gradient vector is inactive, its values do not contribute to the calculation of the left-hand-side gradient vector.
- When all right-hand-side gradient vectors are inactive, the activity flag of the left-hand-side gradient vector is set to inactive.

Static Activity Analysis in OpenAD OpenAD does not currently support dynamic activity analysis. Instead, it uses a static *may activity analysis*. Given user-identified independent and dependent variables, the may activity analysis conservatively identifies local variables that may be active. The generated derivative code will include calls to `sax` subroutines, whose function is similar to the `axpy` routines in ADIC. These `sax` routines will only be called using gradient vectors of variables identified as may active. In Section 5, we define may activity analysis more fully.

4 Overhead of Dynamic Activity Analysis

While dynamic activity analysis can reduce the number of gradient vector operations within derivative code, it does introduce extra activity flag checking as overhead. In this section, we quantify the impact of the overhead of dynamic activity analysis.

4.1 Methodology

We investigated the overhead of dynamic activity analysis on two benchmark testbeds. For C codes, we generated the derivative code using ADIC 1.2, which

Table 2. Characteristics of the C benchmarks

Abr	Benchmark	Independent	Dependent	Size	Source
C1	Ackley	x	f,g	n = 20	see [1, 2]
C2	Boxbetts	x	ret	n = 3	GlobOpt
C3	CamShape	par, r	obj	n = 144	ADIC
C4	GenRosenBrock	x	ret	n = 30	GlobOpt
C5	McCormic	x	ret	n = 2	GlobOpt
C6	Paviani	x	ret	n = 10	GlobOpt
C7	Plate2D	x	f, g	mx = 12	TAO
C8	Polygon	x	obj	n = 73	ADIC

provides two sets of `axpy` and related routines. The original set is a set of macros which do not implement the activity flag and thus provide no activity checking. The second set is a set of hand-coded macros that implement the activity flag and perform dynamic activity checking. By normalizing the pure dynamic activity analysis results to the those with no activity check, we quantify the overhead of dynamic activity analysis.

For Fortran codes, we generated the derivative code using OpenAD, which had no prior support for dynamic activity analysis. We created a tool to auto-generate `sax` subroutines that implement dynamic activity checking. Thus we created two execution units per benchmark: "No Activity Check" uses the OpenAD default routines that do not perform dynamic activity checking, while "Pure Dynamic" uses our auto-generated routines that do. Again, we normalize our Pure Dynamic results to the No Activity Check results to quantify the overhead of dynamic activity analysis.

4.2 Results

Table 1 summarizes the Fortran benchmarks used in our experiments. All are from the Minpack2 benchmark suite [11] except the bminsurf, an example problem from the TAO Toolkit [4]. The column labeled "Overhead" shows the average of four Dynamic execution times normalized against the Static execution time. Our runs represent the maximum possible overhead in that we set all inputs as independent, and all outputs as dependent prior to derivative code generation. The benchmarks display a broad range of overhead averaging 39%.

Table 2 summarizes the C benchmarks used in our experiments. Most of the problems were derived from a c++ test suite for global optimization [9]; the others are part of the ADIC test suite or TAO examples. Figure 4 displays the overhead of dynamic activity analysis for each of the C benchmarks. The arithmetic mean per analysis run is noted in parenthesis within the legend. When 100% of input variables are treated as independent variables (and are therefore active), all variables are active and the cost of dynamic activity checking is pure overhead. The overhead here is significantly lower than that found with the Fortran benchmarks and can be attributed to differences in benchmarks as well as dynamic activity analysis implementation. However, in the more realistic situation where only 50% of the inputs are active, dynamic analysis pays dividends,

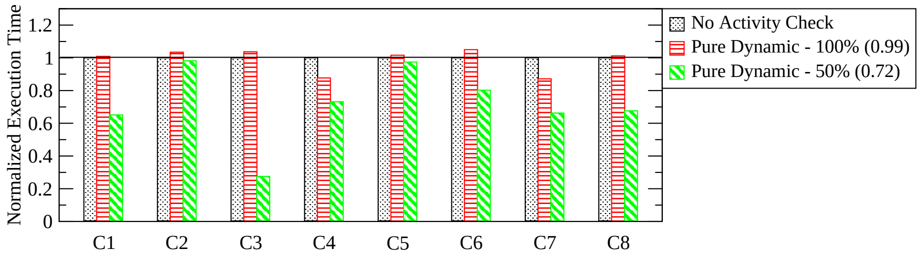

Fig. 4. Overhead of Dynamic Activity Analysis using C benchmarks and speedup when only 50% of input variables are independent. Arithmetic means are indicated within parentheses. See Table 2 for benchmark descriptions.

reducing the execution time from "No Activity Check" by about 28% on average and up to 70% in the case of camshape.

5 Static/Dynamic Analysis

As we saw in Section 4, dynamic activity analysis provides full accuracy at the price of noticeable overhead. OpenAD uses a static may analysis which may incur less overhead by statically determining which local variables are provably inactive. In the generated derivative code this overestimate of the set of active variables ensures correctness but the code may be sub-optimal. We propose a hybrid static/dynamic activity analysis that uses a static forward-direction must analysis.

5.1 Must-May Static Activity Analysis

Static activity analysis is based upon the following definitions. A variable, `v`, is *may-vary* when there is at least one control path to a define of `v` where the value of `v` depends directly or transitively upon an independent variable. A variable, `v`, is *must-vary* at a point in the function when *all* control-flow paths to that point cause the value of `v` to depend directly or transitively upon the value of an independent variable. A variable, `v`, is *may-useful* when there is at least one control path from the define of `v` to the define of a dependent variable where the value of a dependent variable depends directly or transitively upon `v`.

In may activity analysis, a variable, `v`, is classified *may-active* when there is at least one point in the function where `v` is both may-vary and may-useful. OpenAD uses the OpenAnalysis [14] toolkit to implement its data-flow analysis. OpenAnalysis provides a may activity analysis. Partial derivatives for non-may-active variables never have to be calculated.

In *must-may activity analysis*, a variable, `v`, is *must-may-active* at a point in the function when `v` is both must-vary and may-useful. Should the actual execution path arrive at this point, `v` will be dynamically active since it is must-vary. For each memory reference that can be determined must-may-active, we can remove the activity check of dynamic activity checking. Thus, for some benchmark/independent/dependent trios that exhibit must-may-activity, our hybrid

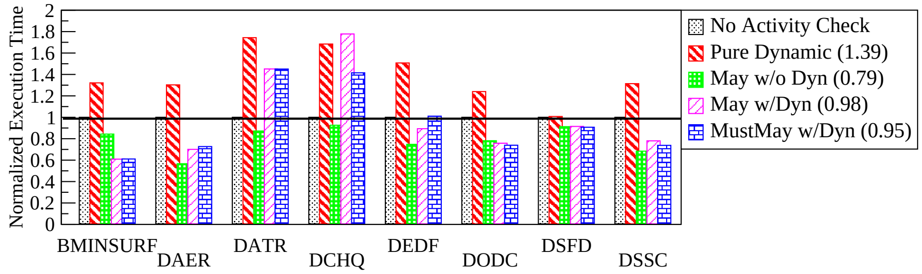

Fig. 5. Fortran results for a variety of static and/or dynamic activity analyses. *May* and *Must-May* are static analyses and *Dyn* indicates dynamic activity analysis. Arithmetic means are indicated within parentheses.

static/dynamic activity analysis may reduce the overhead of dynamic activity checking. Since we are concentrating on forward mode AD, a must-useful vs. may-useful analysis is not exploitable.

Using OpenAnalysis, we implemented the must-may activity analysis. We designed a new set of sax routines that would skip the check of the activity flag on the known must-may-active gradients. To avoid any extra checking in this regard, we re-order the arguments to the sax calls to identify by position the gradients that need to be dynamically checked and those that do not. We manually adjusted the sax calls in each benchmark's derivative code to comply with the new interface. Then we used the must-may activity results to re-order the arguments. We anticipate that this must-may activity analysis will become an option in OpenAD, generating calls under the new interface and automatically re-arranging the arguments.

5.2 Results

In Figure 5 we display the results of our hybrid static/dynamic activity analysis. All data has been normalized to the execution time of *No Activity Check*. The arithmetic mean per analysis run is noted in parentheses within the legend. The second bar represents *Pure Dynamic* activity analysis and visually shows the significant overhead (39% average) of dynamic activity checking. The third bar represents the static *May* activity analysis with no dynamic activity analysis and shows the advantage of pruning the derivative code at inactive variables with an average speedup of 21%. The fourth bar represents the hybrid combination of the static may activity analysis with dynamic activity analysis. In most benchmarks, the win from the static may analysis more than compensates for the overhead of dynamic checking. The fifth bar represents the hybrid combination of the static *Must-May* activity analysis with dynamic activity analysis. Three of the benchmarks show a decrease in execution time by using must-may activity analysis rather than may activity analysis. We anticipate that as we reduce the implementation overhead of our hybrid accumulation routines and examine complex applications where more variables can be statically identified as must-active, the benefits of our hybrid strategy will become more apparent.

6 Conclusion

We have implemented a hybrid static/dynamic strategy for activity analysis. This approach offers the opportunity to use runtime information to avoid unnecessary derivative accumulation operations, as may occur with conservative static analysis, while avoiding the overhead of unneeded runtime tests, as may occur with dynamic analysis. By restricting runtime tests to variables statically identified as may active and eliminating tests for variables statically identified as must-may active, we reduce the number of runtime checks. Our experimental results indicate that this hybrid strategy can sometimes pay dividends, offering improved performance over both a conservative static strategy and a dynamic strategy. We anticipate that as we examine more complex applications and eliminate some of the implementation overhead of the hybrid strategy, the benefits of the hybrid static/dynamic strategy will be even more pronounced.

References

1. D. H. Ackley. *A connectionist machine for hillclimbing*. Kluwer Academic Publishers, Boston, 1987.
2. B. Addis and S. Leyffer. A trust-region algorithm for global optimization. Technical Report ANL/MCS-P1190-0804, Argonne National Laboratory, August 2004.
3. ADIC Webpage. `http://www-fp.mcs.anl.gov/adic/`.
4. S. J. Benson, L. C. McInnes, J. Moré, and J. Sarich. TAO user manual (revision 1.8). Technical Report ANL/MCS-TM-242, Mathematics and Computer Science Division, Argonne National Laboratory, 2005. `http://www.mcs.anl.gov/tao`.
5. C. Bischof, P. Khademi, A. Mauer, and A. Carle. Adifor 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Comput. Sci. Eng.*, 3(3):18–32, 1996.
6. C. Bischof, L. Roh, and A. J. Mauer-Oats. ADIC: An extensible automatic differentiation tool for ANSI-C. *Software: Practice and Experience*, 27(12):1427–1456, December 1997.
7. C. H. Bischof, P. D. Hovland, and B. Norris. On the implementation of automatic differentiation tools. *Higher-Order and Symbolic Computation*, 2004.
8. M. Fagan and A. Carle. Activity analysis in Adifor: Algorithms and effectiveness. Technical Report TR04-21, Rice University, Dept. of Computation and Applied Mathematics, 2004.
9. Global Optimization Functions. `http://www2.imm.dtu.dk/~km/GlobOpt/testex/`.
10. L. Hascoet, U. Naumann, and V. Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8):1401–1417, October 2005.
11. MINPACK-2 webpage. `http://www-fp.mcs.anl.gov/otc/minpack/sectionstar3_1.html`.
12. U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In *International Conference on Computational Science*, pages 1039–1048. Springer, April 2002.
13. OpenAD Webpage. `http://www-unix.mcs.anl.gov/openad/`.
14. OpenAnalysis Webpage. `http://www-unix.mcs.anl.gov/OpenAnalysisWiki/moin.cgi`.