Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2010) 89–104
www.elsevier.com/locate/entcs

SLAMM — Automating Memory Analysis for Numerical Algorithms

John M. Dennis[1,2]

Computational and Information Systems Laboratory
National Center for Atmospheric Research
Boulder, CO 80305, USA

Elizabeth R. Jessup[3,4]

Department of Computer Science
University of Colorado
Boulder, CO 80309, USA

William M. Waite[5]

Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309, USA

Abstract

Memory efficiency is overtaking the number of floating-point operations as a performance determinant for numerical algorithms. Integrating memory efficiency into an algorithm from the start is made easier by computational tools that can quantify its memory traffic. The Sparse Linear Algebra Memory Model (SLAMM) is implemented by a source-to-source translator that accepts a MATLAB specification of an algorithm and adds code to predict memory traffic.
Our tests on numerous small kernels and complete implementations of algorithms for solving sparse linear systems show that SLAMM accurately predicts the amount of data loaded from the memory hierarchy to the L1 cache to within 20% error on three different compute platforms. SLAMM allows us to evaluate the memory efficiency of particular choices rapidly during the design phase of an iterative algorithm, and it provides an automated mechanism for tuning exisiting implementations. It reduces the time to perform *a priori* memory analysis from as long as several days to 20 minutes.

Keywords: memory analysis, sparse linear algebra, source-to-source translation, MATLAB

[1] The work of this author was supported through National Science Foundation Cooperative Grant NSF01, which funds the National Center for Atmospheric Research (NCAR), and through the grants: #OCI-0749206 and #OCE-0825754. Additional funding is provided through the Department of Energy, CCPP Program Grant #DE-PS02-07ER07-06.

[2] Email: dennis@ucar.edu

[3] The work of this author was supported by the National Science Foundation under grants no. CCF-0430646 and CCF 0830458.

[4] Email: Elizabeth.Jessup@Colorado.edu

[5] Email: William.Waite@Colorado.edu

doi:10.1016/j.entcs.2010.08.034

1 Introduction

Traditionally, numerical algorithms have been designed to achieve the best numerical accuracy for a given number of floating-point operations [26,31,30,1,25]. However, this approach is based on the assumption that the time to perform floating-point operations dominates the overall cost of the computation. This assumption is no longer true in general because the cost of memory access has risen in comparison to the cost of floating-point arithmetic. While the performance of microprocessors has been improving at the rate of 60% a year, the performance of dynamic random access memory has been improving at a rate of only 10% [28]. In addition, advances in algorithm development mean that the total number of floating-point operations required for solution of many problems is decreasing [19]. Thus, the time to solution can no longer be considered a function of the number of floating-point operations performed alone but must rather be described as a combination of the costs of both floating-point arithmetic and memory access [19].

Creating new algorithms that demonstrate both numerical and memory efficiency [35,7,2,5,6,37,14,20,17] is a difficult task that has not been addressed extensively or in a systematic fashion. The all-too-common approach is to ignore memory efficiency until after the implementation is complete. Unfortunately, retroactive retooling of code for memory efficiency can be a daunting task. It is better to integrate memory efficiency into the algorithm from the start.

Manual analysis, which involves derivation of an analytical expression for data movement, requires extensive knowledge of computer architecture and software engineering. It is laborious, error-prone, and too complex to perform on a regular basis. Fortunately, we can automate the process by employing computational tools to quantify the memory traffic. A static analysis provides an estimate of memory use by counting variable references in the code itself; a dynamic analysis is performed as the code is run and so reflects actual data access.

One general design strategy is to prototype an algorithm in MATLAB using different memory layouts and to determine how the final version will perform by making measurements on the prototypes. Creating useful prototypes with appropriate statistics-gathering code requires specialized knowledge; this method would thus not be widely available if we didn't package that knowledge. We therefore developed the the Sparse Linear Algebra Memory Model (SLAMM) processor, a source-to-source translator that accepts MATLAB code describing the candidate algorithm and adds blocks of code to predict data movement [10].

Source-to-source translators have a long history in the development of numerical algorithms. In early 1970, Marian Gabriel reported on a translator that accepted a FORTRAN formula and produced a FORTRAN formula computing the derivative of the input formula [16]. That translator was part of a larger system of doing least-squares fits [15]. Gabriel's rationale for providing the translator echoes ours for SLAMM:

> The curve-fitting program ... requires partial derivatives of the function to be fitted, and thus the user had to supply FORTRAN expressions for these deriva-

```
SLAMM Memory Analysis for Body: BLGMRES
       TOTAL: Storage Requirement  Mbytes (SR) : 10.21
       TOTAL: Loaded from L2 -> L1 Mbytes (WSL): 574.64 +- 6.16
                           DGEMM   Mbytes      : 0.00 +- 0.00
                      Sparse Ops   Mbytes      : 79.50 +- 6.16
```

Fig. 1. The result of a memory analysis

tives. Finding them for a complicated function is, at best, a nuisance. At worst, it is a common source of errors.

Over the years, tools have been developed to automate production of analysis and translation algorithms from declarative specifications [22]; using them, processors like SLAMM can now be constructed more easily. We used Eli [18,13], a public-domain compiler generator with sophisticated support for standard tasks such as name analysis, to generate the SLAMM processor.

Section 2 explains how the SLAMM processor analyzes a prototype and translates it to a MATLAB program. In Section 3, we summarize our experiences in applying SLAMM to algorithm development.

2 The SLAMM processor

SLAMM's goal is to predict the amount of data moved by an algorithm to be implemented in a compiled language, given a MATLAB prototype for that algorithm. Figure 1 illustrates information that SLAMM provided to the user for an algorithm named `BLGMRES`, which is a sparse linear system solver for a matrix of order 25,228:

- The sum of the storage required (SR) for all of `BLGMRES`'s variables is 10.21 Mbytes.
- The working set load (WSL) size is the amount of data load from the memory to the processors, which for `BLGMRES` is 574.64 Mbytes. This figure may be in error by up to 6.16 Mbytes, due to the difficulty SLAMM has in predicting data movement for certain programming constructs.
- There were no dense matrix-matrix operations that could be implemented as calls to the Basic Linear Algebra Subprogram (BLAS) DGEMM [12].
- Sparse matrix-vector operations account for 79.50 Mbytes of the total loads and all of the possible error.

Both static and dynamic analysis of the prototype are necessary in order to obtain this information. SLAMM performs the static analysis directly, and also produces a copy of the prototype that has been augmented with additional MATLAB code blocks. The MATLAB interpreter executes the transformed code to provide the dynamic analysis.

The SLAMM processor accepts a MATLAB prototype containing SLAMM *directives*. All SLAMM directives consist of the prefix `%SLM`, a *command*, and a terminating semicolon. A command (Table 1) is a keyword followed by one or more arguments, separated by commas.

Table 1
SLAMM Commands

`start` *name*	Mark the beginning of a body of MATLAB code to be analyzed, and give it the name *name*.
`end` *name*	Mark the end of the body of MATLAB code named *name*.
`FuncStart` *name*	Mark the beginning of a MATLAB function to be analyzed, and give it the name *name*.
`FuncEnd` *name*	Mark the end of the MATLAB function named *name*.
`print` *name*	Request that a memory analysis be printed for *name*.
`Func` *ident*, ..., *ident*	Classify one or more identifiers as names of functions for which data movement is to be ignored.
`IncFunc` *ident*, ..., *ident*	Classify one or more identifiers as names of functions that contain significant data movement.

In Section 2.1 we explain how the number of memory accesses corresponding to a variable reference in the prototype is related to the region in which it occurs. Unfortunately, not every variable reference has the same effect, as discussed in Section 2.2. The translation based upon a static analysis of the prototype is described in Section 2.3.

2.1 Regions and variable accesses

In the simplest case, each occurrence of a variable in the prototype represents an access to that variable's memory location by the final algorithm. However, the number of times the memory location is accessed depends on the execution path and the region containing the variable occurrence. As an example, consider the prototype shown in Figure 2. The conditional guarantees that either B2 or B3, but not both, will be executed. Similarly, variables occurring in a region controlled by an iteration will be accessed many times.

SLAMM directives can also be used to delimit regions on which to perform memory analysis. In Figure 2, directives have been used to define the region B1 and give it the name `foo`. The `print` command causes an output similar to that described by Figure 1.

Name analysis is the process used to discover and record all of the relationships among identifier occurrences within program regions. This process is well understood, and can be described in terms of a few simple concepts [23]:

Range: A region of the program text. Each of the regions B0-B3 in Figure 2 constitutes a range. For each range, SLAMM calculates: the total storage requirement (SR), the amount of data loaded from memory, the working set load (WSL) size, and the amount of data stored to memory, the working set store (WSS) size.

Defining occurrence: An occurrence of an identifier that would be legal even if it

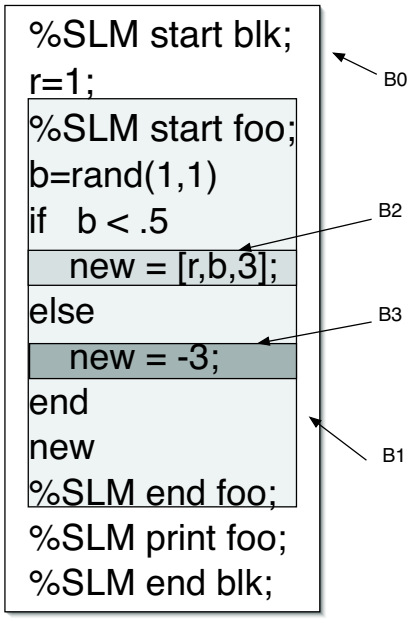

Fig. 2. Scopes for counter association

were the sole occurrence of that identifier. All identifier occurrences in SLAMM are defining occurrences. For each defining occurance, SLAMM creates a new data structure by prepending `slm_` to the identifier and captures type, extent, and required memory storage.

Binding: A relationship between an identifier and a unique entity.

Scope: The region of the program text within which a binding is valid. In SLAMM, a binding holds over the entire range containing a defining occurrence of the identifier. A defining occurrence for the same identifier in a nested range creates a new binding. Within the scope of that new binding, the old binding is invalid. The scope of the new binding therefore constitutes a "hole" in the scope of the old binding.

Name space: A collection of ranges, occurrences, bindings and scopes. SLAMM has two name spaces: the *variable* name space and the *counter* name space. Every identifier occurrence has a binding in each name space.

The only ranges that are relevant in the variable name space are those constituting files and functions. In Figure 2, the region B0 is the only range in the variable name space. The entity bound to an identifier in the variable name space represents the prototype variable named by that identifier.

All of the ranges are relevant in the counter name space. The entity bound to an identifier in the counter name space represents the number of times the prototype

variable named by that identifier is accessed in the scope of that binding.

2.2 Correcting the access count

An occurrence of an identifier in the prototype does not necessarily indicate that the corresponding variable must be loaded from the memory hierarchy. Fortunately, it is possible to improve the accuracy of the base identifier counts through a series of corrections. These corrections capture properties of the translation from MATLAB to a compiled language.

An efficient implementation of the expression `n = size(A,1);`, which uses the built-in MATLAB function `size` to set the variable `n` to the row dimension of the matrix `A`, does not require access to the entire matrix `A`. The necessary functionality can be provided in a compiled language by accessing a single integer variable. The *function call correction* addresses the counting mismatch by decrementing the counts for those identifiers that occur within a function call argument list. We are careful not to decrement the count for those identifiers that occur in an expression in an argument list. For example, the function call correction does not apply to the vector identifiers `r_m1` and `r` in `m = size(r_m1'*r);`. We describe our approach for memory analysis of function calls in the next section.

The calculation of the dot product `alpha = r'*r;` shows the need for another type of correction. In this case, the identifier `r` occurs twice in a single expression. However, the vector `r` is only loaded once from the memory hierarchy. The duplicate occurrence of `r` within a single expression represents cache reuse that cannot be ignored in the memory analysis calculation. To address cache reuse, we decrement the identifier counts for any identifiers that occur multiple times within a single statement. For simplicity, this *duplicate correction* does not address the possibility of cache reuse between multiple statements.

The expression `r_m1 = r;` copies the vector `r` to vector `r_m1`. Memory copies are necessary in MATLAB for renaming purposes because MATLAB lacks a pointer construct. Single variable assignments are implemented as either a memory copy or a pointer assignment in a compiled language. We assume that, for an efficiently implemented algorithm, variables with a large storage requirement (e.g., vectors) use pointer assignments. Variables with a small storage requirement, such as a single floating-point value, use memory copies. Because the cost of memory copy for small variables is insignificant, we assume for the purposes of memory analysis that the assignment of one variable to another is always a pointer assignment. The *copy correction* decrements the identifier counts to properly address pointer assignment.

The corrections discussed so far involve decrementing counts to properly account for particular identifier occurrences. A different form of correction is required in `r = A*w;`. Here SLAMM needs additional information about the types of entity represented by `A` and `w`. For example, `A` may be a sparse or dense matrix. Type information is only available to the MATLAB interpreter. We address our lack of type information by transforming certain operators into function calls. For example, to apply the *special operator correction* we transform the expression `r=A*w` to the equivalent `r=mtimes(A,w)`. A profiled version of `mtimes` calculates the data

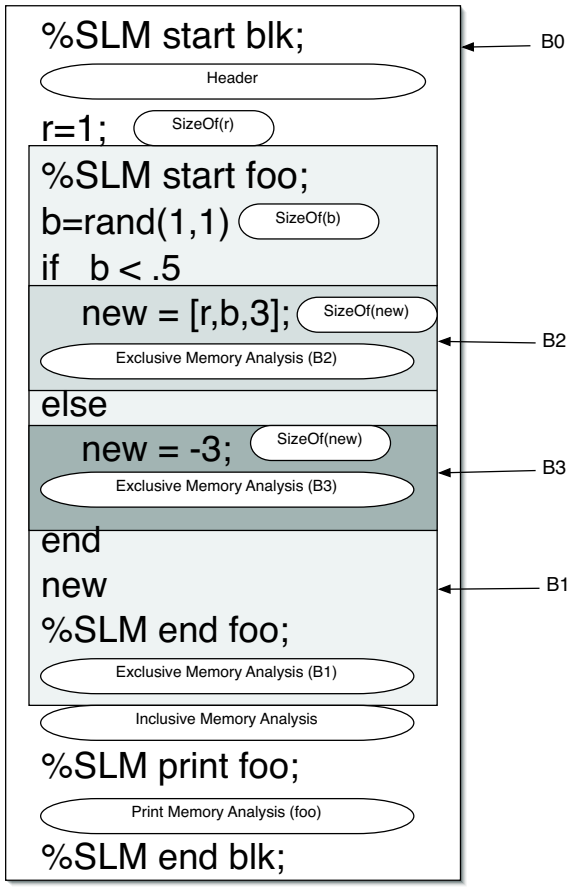

Fig. 3. Translation of code in Figure 2

movement based on the determination of type at runtime.

2.3 Translating SLAMM to MATLAB

Figure 3 illustrates the translation of Figure 2 by the SLAMM processor. The oval boxes in Figure 3 represent additional code blocks introduced to carry out the dynamic analysis. Note that the SLAMM directives, being MATLAB comments, can be copied into the output as documentation.

`Header` contains the definitions and initialization of MATLAB structures used to accumulate the information for each scope and identifier.

A `SizeOf` code block is inserted just after an assignment is made to a variable. It typically contains a single call to the MATLAB `whos` function. Here is the assignment and `SizeOf` block in scope B2 of Figure 3:

```
new = [r,b,3];
[slm_new] = whos('new');
```

This captures the type, extent, and required memory storage for `new` in a generated variable (`slm_new`) named for the user's variable. Field `bytes` of the structure

created by the `whos('new')` call specifies the amount of storage occupied by variable `new`.

The purpose of an exclusive memory analysis block is to accumulate information from a particular scope. Each range requires an exclusive memory analysis code block. The exclusive memory analysis block for B1 (shown in Figure 3) contains (among other things):

```
slm_foo.wsl = slm_foo.wsl + slm_new.bytes + slm_b.bytes;
slm_foo.wss = slm_foo.wss + slm_b.bytes;
```

This calculates the amount of data loaded from the memory hierarchy by the `if`-expression and the fetch of `new`, accumulating the result in field `wsl` of the structure `slm_foo`. Similarly, the amount of data stored to the memory hierarchy (in the assignment to `b` at the beginning of the block) is recorded in field `wss`. Each of these assignments has two components: the total amount loaded or stored before this execution of the code in B1 (e.g., `slm_foo.wss`) and the amount loaded or stored during this execution (e.g., `slm_new.bytes`). The former is initialized to zero in `Header`, the latter is a constant determined by `whos`.

Note that this calculation accounts only for the accesses in the scope of the bindings of B1. Both `new` and `b` have defining occurrences in B2, and thus B2 constitutes a hole in the scope of the B1 bindings for those identifiers. Memory access information associated with the assignment in B2 is accumulated by code in the exclusive memory analysis code block B2.

In general, each term defining a variable access is multiplied by the count of occurrences described in Section 2.2. This count is 1 for all variable accesses in Figure 3, and therefore SLAMM omits it.

The purpose of an inclusive memory analysis block is to gather the data from all of the exclusive memory analysis blocks in a function or program. It is generated at the end of the text of that function or program, where it will be executed exactly once. The inclusive memory analysis block of Figure 3 contains (among other things):

```
slm_foo.wsl = slm_foo.wsl + slm_elseLn7.wsl;
slm_foo.wsl = slm_foo.wsl + slm_ifLn5.wsl;
```

Here `ifLn5` and `elseLn7` are SLAMM's names for B2 and B3, respectively.

`Print Memory Analysis(foo)` consists of a call to a library function called `SlmPrtAnalysis` with an appropriate structure as its argument.

SLAMM differentiates between identifiers that correspond to functions and identifiers that correspond to variables. Functions are further classified into those that provide a significant contribution to data movement and those that do not. We refer to those functions that provide a significant contribution to data movement as *profiled functions*. SLAMM maintains the proper classification in a manually constructed table for a collection of commonly used built-in MATLAB functions. For example, the MATLAB function `size`, which determines the extent of a variable, typically does not contribute to data movement but the function `qr`, which calculates the QR factorization of an input matrix, certainly does.

Two types of code transformations are necessary for profiled functions. The first involves changing the function call, while the second involves changes to the function itself.

SLAMM transforms a function call to a profiled built-in MATLAB function by prefixing the string `SLM` to its name and adding an output argument. The additional output argument is a MATLAB structure containing the SLAMM-calculated memory analysis. The profiled function's contribution to data movement is subsequently accumulated in the appropriate exclusive memory analysis code block of the caller.

For profiled functions that return multiple output arguments, the code transformation only requires the addition of one extra output argument. For profiled functions that return a single output argument, additional code transformation may be necessary. For example, a single expression that uses the output argument of a profiled function as an operand requires the generation of multiple sub-expressions. All sub-expressions are linked by temporary variables. The generated MATLAB for the expression $t = sin(r) + cos(z)$, where $r, z, t \in \mathbb{R}^n$ is:

```
[slm_L1C5, slm_sin__L1C5] = SLMsin(r);
[slm_L1C14,slm_cos__L1C14] = SLMcos(z);
t = slm_L1C5 + slm_L1C14,
```

Here the single original expression $t = sin(r) + cos(z)$ is broken into three separate statements. The temporary variables `slm_L1C5` and `slm_L1C14` are the original output arguments of the *sin* and *cos* functions respectively and are used to calculate the expected result `t`. The structures `slm_sin__L1C5` and `slm_cos__L1C14` contain the results of the memory analysis of the *sin* and *cos* functions respectively.

The function call transformation requires the generation of new versions of the profiled functions. SLAMM provides a collection of profiled functions for all necessary built-in MATLAB functions. Each consists of a call to the original function and the assignment of the memory analysis structures. For a user-supplied MATLAB function, the SLAMM language processor generates the various memory analysis code blocks described above and alters the name and output arguments appropriately.

3 An Application of SLAMM

Automated memory analysis provides both the ability to evaluate the memory efficiency of a particular design choice rapidly during the design phase and the ability to improve the memory efficiency of a pre-existing solver. We illustrate both applications by using SLAMM to reduce the execution time of the Parallel Ocean Program (POP) [32], a global ocean model developed at Los Alamos National Laboratory. POP, which is used extensively as the ocean component of the Community Climate System Model [8], uses a preconditioned conjugate gradient solver to update surface pressure in the barotropic component. Parallelism on distributed memory computers is supported through the Message Passing Interface (MPI) [34] standard.

We used POP version 2.0.1 [29] to examine data movement in the preconditioned

Table 2
The data movement for a single iteration of preconditioned conjugate gradient solver in POP using the `test` grid. The values of WSL_P and WSL_M are in Kbytes

Solver		version	Ultra II		POWER4		R14K	
Implementation	WSL_P		WSL_M	error	WSL_M	error	WSL_M	error
PCG2+2D	4902	v1	5163	5.3%	5068	3.4%	5728	16.9%
		v2	4905	0.1%	4865	-0.7%	4854	-1.0%
PCG2+1D	3218		3164	-1.7%	3335	3.7%	3473	7.9%
Reduction	34%		39%		34%		39%	

conjugate gradient solver using the `test` and `gx1v3` grids. The `test` grid is a coarse grid provided with the source code to facilitate the porting of POP to other compute platforms. POP with the `gx1v3` grid, which has one degree separation between grid points at the equator, is higher resolution than the `test` grid and represents 20% of the total compute cycles at the National Center for Atmospheric Research (NCAR) [4].

POP uses a three-dimensional computational mesh. The horizontal dimensions are decomposed into logically rectangular two-dimensional (2D) blocks [21]. The computational mesh is distributed across multiple processors by placing one or more 2D blocks on each processor. The primary advantage of the 2D data structure is that it provides a regular stride-one access for the matrix-vector multiply. The disadvantage of the 2D data structure within the conjugate gradient solver is that it includes a potentially large number of grid points that represent land. In effect, a number of explicitly stored zeros are added to the matrix. An alternative 1D data structure that uses compressed sparse row storage would avoid the inclusion of land points but would introduce indirect addressing. Additional details concerning the changes in data structures are provided in [11].

We used SLAMM to evaluate the required data movement for conjugate gradient solvers based on 1D and 2D data structures. We describe two types of data movement, the predicted data movement (WSL_P) and the measured data movement (WSL_M). WSL_P is predicted by SLAMM, and WSL_M is measured using hardware performance counters and refers to data loaded from the L2 to the L1 cache. For the 1D data structure, we used the PCG solver routine provided by MATLAB. We wrote a new PCG solver in MATLAB which used the same 2D data structures as in POP. Using the SLAMM directives described in Section 2, we created a region for one iteration of each of the algorithms. WSL_P for the algorithm with the 2D data structure (PCG2+2D) and the 1D data structure version (PCG2+1D) for the `test` grid are provided in the second column of Table 2. We tested both a naive implementation of PCG2+2D (v1) and an optimized one (v2) described later in this section. The SLAMM prediction is the same for both.

SLAMM predicts that the use of the 1D data structure reduces data movement by 34% versus the existing 2D data structure. Based on the expectation of a 34% reduction in data movement, we implemented the 1D data structure in POP. If SLAMM had predicted a minor reduction in data movement, or even an increase in data movement, the 1D data structure would have never been implemented. *A priori* analysis like this provides confidence that the programming time to implement

Table 3
Description of the microprocessor compute platforms and their cache configurations

CPU	Ultra II	POWER4	R14K
Company	SUN	IBM	SGI
Mhz	400	1300	500
L1 Data-cache	32KB	32KB	32KB
L2 cache	4 MB	1440 KB	8 MB
L3 cache	–	32 MB	–

```
! ================================
! code block: solver v1
! ================================
do iblock=1,nblocks
     P(:,:,iblock) = Z(:,:,iblock) + P(:,:,iblock)*beta
     Q = operator(P,iblock)
     WORK0(:,:,iblock) = Q(:,:,iblock)*P(:,:,iblock)
enddo
delta=global_sum(WORK0,LMASK)

! ================================
! code block: solver v2
! ================================
delta_local=0.d0
do iblock=1,nblocks
     P(:,:,iblock) = Z(:,:,iblock) + P(:,:,iblock)*beta
     Q = operator(P,iblock)
     WORK0 = Q(:,:,iblock)*P(:,:,iblock)
     delta_local = delta_local + local_sum(WORK0,LMASK(:,:,iblock))
enddo
delta=gsum(delta_local)
```

Fig. 4. A code block that implements a piece of the PCG algorithm for versions v1 and v2.

code modifications or new algorithms is not wasted.

To evaluate the quality of our implementations and to check the accuracy of the SLAMM-based predictions, we instrumented all versions of the solver with a locally developed performance profiling library (Htrace), which is based on the PAPI [27] hardware performance counter API. Htrace calculates data movement by tracking the number of cache lines moved through the different components of the memory hierarchy. We focus on three primary microprocessor compute platforms that provide counters for cache lines loaded from the memory hierarchy to the L1 cache: Sun Ultra II [24], IBM POWER 4 [3], and MIPS R14K [36]. A description of the cache configuration for each compute platform is provided in Table 3.

The measured data movement (WSL_M) is also presented in Table 2 for the 2D data structure implementations (PCG2+2D v1 and v2) and the 1D version (PCG2+1D) on each of the compute platforms. We report average data movement across ten iterations. While the discrepancies between the measured and predicted WSL for the PCS2+2D v1 solver are minimal for both the Ultra II and POWER4 platforms, the measured value of 5728 Kbytes for the R14K is 17% greater than the predicted value of 4902 Kbytes. The difference in data movement between the three compute platforms may be due to additional code transformations performed by the Ultra II and POWER4 compilers. That the PCG2+2D v1 solver is loading 17% more data from the memory hierarchy than necessary on the R14K is an indication that it is possible to improve the quality of the implementation.

While SLAMM provides an indication of a potential performance problem, it

does not isolate the source of the problem. Locating and addressing the problem is still the responsibility of the software developer. An examination of the source code for the PCG2+2D v1 solver indicates that a minor change to the dot product calculation reduces data movement. Code blocks for version v1 and v2 of the solver are provided in Figure 4. The function `operator` applies the local matrix product, and the array `LMASK` is an array that masks out points that correspond to land points. In version v1 of the `do` loop, a temporary array `WORK0` is created that contains the point-wise product of two vectors `Q` and `P`. Outside the `do` loop, the dot product of `LMASK` and the `WORK0` array is calculated by `global_sum`. If the size of data accessed in the `do` loop is larger than the L1 cache, then a piece of the `WORK0` array at the end of the `do` loop is no longer located in the L1 cache and must be reloaded to complete the calculation.

In the optimized version v2 of the `do` loop, we employ a scalar temporary `delta_local` to accumulate each block's contribution to the dot product of `Q` and `P`. We then use a function `local_sum` that applies the land mask to complete the dot product. Finally, we replace the function `global_sum` with a call to `gsum`. The subroutine `gsum`, when executed on a single processor, is an assignment of `delta_local` to `delta`. Because version v2 of the code block does not access `WORK0` outside the `do` loop, it potentially reduces data movement.

The WSL_M values in Table 2 for the v1 and v2 versions of the PCG2+2D solver on the R14K indicate that the rearranged dot product calculations reduce data movement by 18%. Note, that data movement is also reduced on the Ultra II and POWER4 platforms but to a lesser extent.

Table 2 also provides the relative error between predicted and measured data movement for the solvers. SLAMM predicts data movement to within an average error of 0.6% for the PCG2+2D v2 solver and within 4.4% for the PCG2+1D solver. (The deviation for the poorer quality PCG2+2D v1 is greater.) Table 2 indicates that the actual percentage reductions in data movement are very similar to the predicted reductions. The results in Table 2 clearly demonstrate that it is possible for automated memory analysis to predict the amount of data movement accurately and so to provide *a priori* knowledge of the memory efficiency of a particular design choice before implementation in a compiled language.

Note that it is entirely possible to perform the same analysis manually without SLAMM. While the calculation for the 2D case would be straightforward, the 1D case would be more complicated. In particular, an accurate prediction of the amount of data movement for the 1D data structure would require detailed information about location of land points.

It is possible to estimate the difficulty of manual memory analysis by comparing the lines of code associated with the MATLAB prototype to the memory analysis code. Table 4 shows the number of non-comment lines for the 1D PCG prototype in MATLAB. For the 1D PCG prototype, the prototype grew from 41 lines to a total of 48 lines with the addition of seven %SLM directives. The SLAMM generated output code (prototype and memory analysis code) is 290 lines. In this case, memory analysis requires approximately seven times the number of code lines required by

the algorithm alone. This illustrates that manual calculations are generally onerous and error prone and thus may not be performed regularly (or at all).

We next examine the impact of reducing data movement on execution time. The timestep of POP includes a baroclinic and a barotropic component. The barotropic component is composed of a single linear solver for surface pressure. We executed POP using the `test` grid on a single processor of each compute platform for a total of 20 timesteps. This configuration requires an average of 69 iterations of the PCG2 algorithm per timestep. Table 5, gives the barotropic execution time in seconds using the three implementations of the solver. We include the execution time for the initial implementation of the solver using 2D data structures to accurately reflect the overall impact automated memory analysis has on execution time. Note that the PCG1+1D solver consistently has a lower execution time than either of the 2D data structure based solvers. The last row in Table 5 contains the percentage reduction in barotropic execution time for the PCG2+2D v1 versus the PCG2+1D solver. Table 5 indicates that code modifications either evaluated or identified by automated memory analysis reduce execution time by an average of 46%. Curiously, a comparison of Tables 2 and 5 indicates that the percentage reduction in execution time is even larger than the percentage reduction in data movement. The reason for the discrepancy is unclear.

We next examine the impact of the 1D data structures on parallel execution time on POWER4 platform using the `gx1v3` grid. We focus on the execution time of POP on 64 POWER4 processors. This is a typical configuration of POP that consumes approximately 2.4 million CPU hours every year at NCAR. We wrote a generalized gather-scatter routine to provide parallel execution under MPI for the 1D version of the solver. Recently, an alternative preconditioned conjugate gradient algorithm that uses a single dot product [9] (PCG1) was added to POP. The PCG1 algorithm provides a performance advantage for parallel execution because it eliminates one of the distributed dot product calculations. We configured POP to execute a total of 200 timesteps on 64 IBM POWER4 processors, with an average of 151 iterations per timestep. Note that, while the cost of the solver is important, it

Table 4
Source code lines for MATLAB prototype.

Developer generated Source Lines		SLAMM generated prototype and analysis code
original	%SLM	
41	7	290

Table 5
Barotropic execution time for 20 timesteps of POP in seconds using the `test` grid on a single processor

Solver implementation	Ultra II	POWER4	R14K
PCG2+2D v1	21.17	4.57	8.58
PCG2+2D v2	20.49	4.01	7.97
PCG2+1D	12.74	2.11	4.61
Reduction	39%	54%	46%

does does not dominate the total cost of a timestep. The execution time for the time stepping loop in seconds for the four solver implementations is provided in Table 6. These results indicate that use of the PCG1+1D solver versus the PCG1+2D

Table 6
Total execution time for 200 timesteps with `gx1v3` grid on 64 POWER4 processors.

	Solver Implementation			
	PCG2+2D v1	PCG1+2D v1	PCG2+1D	PCG1+1D
total time (sec)	86.2	81.5	78.8	73.9

solver reduces total POP execution time by 9%. A 9% reduction in total execution time of POP is significant because that model has already been extensively studied and optimized [21,33]; the additional 9% translates into a reduction of 216,000 CPU hours per year at NCAR.

4 Conclusion

Data movement is an important characteristic of a numerical algorithm running on today's computers, contributing significantly to that algorithm's running time. The effects of design decisions on data movement should therefore be explored before undertaking a costly implementation. Predictions of data movement are possible by adding instrumentation to a MATLAB prototype of the algorithm. As in the case of the partial derivative calculation problem described by Gabriel nearly forty years ago, that process is at best a nuisance and at worst a common source of errors.

SLAMM automates the process of modifying the MATLAB prototype, reducing the cost of a test from days to minutes. It encapsulates knowledge about the characteristics of variable access and about strategies for accumulating information during a prototype run. If more detailed knowledge and better strategies evolve over time, they can be incorporated into SLAMM without the need for user re-training.

Most of the difficulty in automating the instrumentation process lies in the common tasks of analyzing the text of the prototype to select the appropriate instrumentation code. Because of advances in compiler technology, those analysis tasks can be described by declarative specifications from which code can be produced automatically. Doing so dramatically reduces the cost of developing a tool like SLAMM.

We have shown that SLAMM can be used to guide the performance improvement of large codes that are in current use as well as in the design of new algorithms. It provides one more example of the utility of source-to-source translators taking over the tedious and error-prone aspects of software development.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, and S. Hammarling. *LAPACK Users' Guide*. SIAM, 1992.

[2] A. Baker, J. Dennis, and E. R. Jessup. Toward memory-efficient linear solvers. In J.M.L.M. Palma, J. Dongarra, V. Hernandez, and A. A. Sousa, editors, *VECPAR '2002, Fifth International Conference*

on High Performance Computing for Computational Science: Selected Papers and Invited Talks, volume 2565 of *Lecture Notes in Computer Science*, pages 315–327. Springer, Berlin, 2003.

[3] S. Behling, R. Bell, P. Farrell, H. Holthoff, F. O'Connell, and W. Weir. *The POWER4 Processor Introduction and Tuning Guide*. IBM Redbooks, November 2001. http://www.redbooks.ibm.com.

[4] T. Bettge. National Center for Atmospheric Research, Dec. 2005. Personal Communication.

[5] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.

[6] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical tiling for improved superscalar performance. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 239–245, Washington, DC, USA, 1995. IEEE Computer Society.

[7] A. T. Chronopoulos and C. W. Gear. S-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[8] W. D. Collins, C. M. Bitz, M. L. Blackmon, G. B. Bonan, C. S. Bretherton, J. A. Carton, P. Chang, S. C. Doney, J. J. Hack, T. B. Henderson, J. T. Kiehl, W. G. Large, D. S. McKenna, B. D. Santer, and R. D. Smith. The community climate system model: CCSM3. *Journal of Climate: CCSM Special Issue*, 11(6), 2005.

[9] E. F. D'Azevedo, V. L. Eijkhout, and C. H. Romaine. Conjugate gradient algorithms with reduced synchronization overhead on distributed memory multiprocessors. Technical Report 56, Lapack Working Note, August 2002.

[10] J. M. Dennis. *Automated Memory Analysis: Improving the Design and Implementation of Iterative Algorithms*. PhD thesis, University of Colorado, Boulder, July 2005.

[11] J. M. Dennis and E. R. Jessup. Applying automated memory analysis to improve iterative algorithms. *SIAM Journal of Scientific Computing*, 29(5):2210–2223, 2007.

[12] J. Dongarra, J. DuCroz, S. Hammarling, and I. Duff. Algorithm 679: A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:18–28, 1990.

[13] Eli: An integrated toolset for compiler construction. Documentation, examples, download from http://eli-project.sourceforge.net/.

[14] Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL. http://www.ibm.com/systems/p/software/essl.html, 2007.

[15] M. Gabriel. Special-purpose language for least-squares fits. Technical Report ANL-7495, Applied Mathematics Division, Argonne National Laboratory, September 1968.

[16] M. Gabriel. A symbolic derivative taker for a special-purpose language for least-squares fits. Technical Report ANL-7628, Applied Mathematics Division, Argonne National Laboratory, February 1970.

[17] K. Goto and R. van de Geijn. High-performance implementation of the level-3 BLAS. Technical Report TR-2006-23, The University of Texas at Austin, Department of Computer Sciences, 2006.

[18] R. W. Gray, S. P. Levi, V. P. Heuring, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992. http://eli-project.sourceforge.net/.

[19] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In A. Ecer et al., editor, *Proceedings of Parallel CFD'99*, pages 233–240. Elsevier, 1999.

[20] Intel Math Kernel Library. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm, 2007.

[21] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. III White, and J. Levesque. Practical performance portability in the Parallel Ocean Program (POP). *Concurrency Comput. Prac. Exper*, 17:1317–1327, 2005.

[22] U. Kastens, A. M. Sloane, and W. M. Waite. *Generating Software from Specifications*. Jones and Bartlett, Sudbury, MA, 2007.

[23] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[24] Sun Microsystems. The Ultra2 architecture: Technical white paper, 2005. http://pennsun.essc.psu.edu/customerweb/WhitePapers/.

[25] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the ibm-sp system. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 70, New York, NY, USA, 1999. ACM Press.

[26] D. P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra and its Applications*, 29:293–322, 1980.

[27] PAPI: Performance Application Programming Interface: User's Guide. http://icl.cs.utk.edu/papi, 2005.

[28] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yellick. A case for intelligent RAM. *IEEE Micro*, pages 34–44, March/April 1997.

[29] The Parallel Ocean Program (POP). http://climate.lanl.gov/Models/POP, 2006.

[30] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of Computation*, 37(155):105–126, 1981.

[31] H. D. Simon. *The Lanczos algorithm for solving symmetric linear systems*. PhD thesis, University of California, Berkeley, April 1982.

[32] R. D. Smith, J. K. Dukowicz, and R. C. Malone. Parallel ocean general circulation modeling. *Physica D*, 60:38–61, 1992.

[33] A. Snavely, X. Gao, C. Lee, L. Carrington, N. Wolter, J. Labarta, J. Gimenez, and P. Jones. Performance modeling of HPC applications. In *Parallel Computing (ParCo2003)*, 2003.

[34] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference: Volume 1, The MPI Core*. The MIT Press, 2000.

[35] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing 1992*, 1992.

[36] S. Vellas. Scalar Code Optimization I, 2005.
http://sc.tamu.edu/help/origins/sgi_scalar_r14k_opt.pdf.

[37] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.