Compiler-Directed Run-Time Monitoring of Program Data Access

Chen Ding Yutao Zhong

Computer Science Department
University of Rochester
Rochester, New York
U.S.A.
{cding,ytzhong}@cs.rochester.edu

ABSTRACT

Accurate run-time analysis has been expensive for complex programs, in part because most methods perform on all data. Some applications require only partial reorganization. An example of this is off-loading infrequently used data from a mobile device. Complete monitoring is not necessary because not all accesses can reach the displaced data. To support partial monitoring, this paper presents a framework that includes a source-to-source C compiler and a run-time monitor. The compiler inserts run-time calls, which invoke the monitor during execution. To be selective, the compiler needs to identify relevant data and their access. It needs to analyze both the content and the location of monitored data. To reduce run-time overhead, the system uses a source-level interface, where the compiler transfers additional program information to reduce the workload of the monitor. The paper describes an implementation for general C programs. It evaluates different levels of data monitoring and their application on an SGI workstation and an Intel PC.

1. INTRODUCTION

An information system often needs to monitor its data usage to provide better service, improve performance, and guarantee correctness and reliability. Since checking all accesses to all data is too costly, run-time monitoring has to be selective. Till now, this has been the job of a programmer, who finds all accesses that needs monitoring and inserts appropriate code in appropriate places. The difficulty of programming and probability of error increase proportionally with the size of program and the complexity of data, especially in C programs that rely on efficient access through pointers. With the exponential increase in storage and computing capacity, today's systems process data at unprecedented volume and complexity. Manual data monitoring has become increasingly cumbersome to write, change, and maintain.

In this paper, we describe a new framework of selective data monitoring. It has two components: a source-to-source C *compiler* and a run-time *monitor*. The compiler analyzes a program and inserts run-time calls for the right data at the right location. During program execution, the monitor receives information from compiler-inserted calls and keeps track of the layout of monitored data and their accesses. The interface between the two components consists of source-level function calls, which allow various information about the program to be exchanged as parameters.

We use the compiler and the source-level interface for three reasons. The first is compiler selection and optimization. The compiler chooses what data to monitor and at which points in a program, so the system incurs run-time cost for only relevant parts of a program instead of its entirety. The second reason is run-time use of compiler information. The compiler deposits useful analysis results in calls to the monitor. For example, with compiler-supplied information, the monitor can view memory as a collection of data objects instead of memory cells. The last goal is machine-independent monitoring. All changes are made at the source level. The transformed program can run on any machine that has a C compiler. Furthermore, because of the run-time support, the new system can handle all features of C language.

Selective monitoring is useful for data off-loading. Embedded systems need to reduce the size of physical memory to minimize manufacture cost. They also need to reduce the size of active memory to minimize energy consumption. Both tasks involve relocating certain program data either to an off-device network or to an on-device but inactive (low-power) memory module. Data off-loading must be selective because it targets infrequently used data. It must be efficient both in time and space, considering the limited resources an embedded system has. Finally, it should be machine independent because embedded processors are often not binary compatible even between generations of the same product.

Our approach extends existing techniques in general-purpose data analysis. One such technique is binary profiling, which instruments object code to collect the target address at each memory operation. Binary instrumentation does not need program source, and it can access high-level program information if available. However, most techniques of binary profiling perform blindly on all data. Its low-level interface has limited latitude in collecting and representing program information other than the target address. Binary instrumentation tools are also machine and compiler dependent.

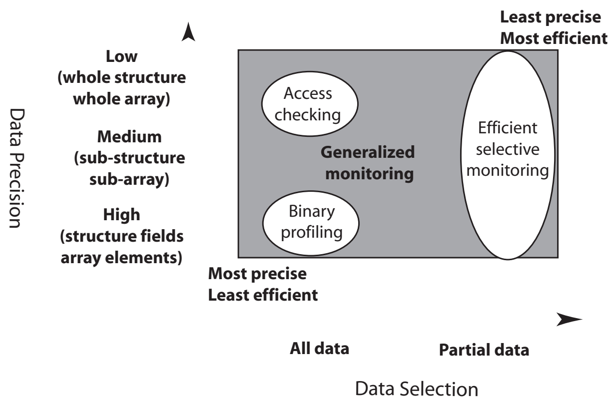

Figure 1: Two dimensions of data monitoring

Another technique is memory-safety checking for C/C++ programs. It checks accesses to all data, so it is not selective. It determines whether an access is inside an object but not its exact location inside the object. The problem of imprecision is also shared by a third technique, conservative garbage collection, which treats heap objects as indivisible. Different from previous efforts, we explore a more sophisticated compiler and a high-level interface supporting the compiler. We aim at data monitoring that is selective, efficient and flexible.

The diagram in Figure 1 shows the two dimensions of selective monitoring, as well as the relation of different techniques. The horizontal axis represents the selection of data and the vertical axis the selection of precision. Our framework provides generalized monitoring, which allows selection in both directions. As we will see later, data monitoring can be efficient if we target a subset of data instead of all data. Partial monitoring is shown by the right-most circle in the diagram. Both complete and partial monitoring have important applications, which will be discussed in Section 4.

The rest of the paper is organized as follows. Section 2 describes the monitoring system. The next two sections study its overhead and describe its application in data off-loading. The last two sections discuss related work and conclude.

2. THE MONITORING SYSTEM

The following description refers to program data as a collection of *data units*, each is an independent and contiguous memory section. No two data units overlap. Locations of data units are not related, i.e. a program can not access one data unit based on the address of another data unit. Data units include instances of global, local, parameter variables and dynamic allocation.

We will use C code as examples. As shown in the section on run-time support (Section 2.3), the system can be extended to handle all features of C language and to support library code with no program source. However, compiler selection and optimization are effective only if the program is in source form and observes a simple notion of type safety, where all data accesses are type correct, and type casting happens only between pointer values. We follow the convention that a single data item can be accessed as an array of one element. Type safety can be ensured by static type checking plus run-time range checking. We use conventional C compilers for static type checking. We support run-time checks as a by-product of our monitoring system. The generated code used in this section was obtained by our actual implementation, with minor syntax changes to make it more compact.

The following three sub-sections describe first the high-level interface and then the components on the two sides of the interface: the compiler and the monitor. The focus is on the role of the compiler in enabling a high-level interface and reducing monitoring overhead, both directly at compile time and indirectly at run time.

2.1 High-Level Interface

The interface links the two components of the system by transferring compiler-supplied information to the run-time monitor. The interface is high-level because it describes program data and their accesses through source-level function calls. We first describe each part of the interface and then discuss its benefits.

2.1.1 Data Definition

Data definition includes data allocation and its type definition. We use terms *structure* and *type* interchangeably: they both mean source-level content of a data unit, which is either a basic type or an aggregated type. The basic types are primitive data types such as an integer or a pointer. All pointers are considered as a single primitive type. Aggregated types are defined as *structure*, *union*, or *array* of other types. Data allocation includes static allocation of global data, run-time allocation of heap data, and stack allocation of local and parameter data.

The compiler inserts a run-time call for each type definition. For a structure or a union, it describes the name of the structure, its fields, and their offsets. Multiple fields of a union may have the same offset. For an array, it describes the size of the array and the type of its elements. One problem is that the offset of a field and the size of an array element are dependent on the back-end compiler. For example, a byte field in a structure can occupy one byte on one machine but two bytes on another. The solution is to record symbolic values based on a sample data item. When the program is compiled into machine code, symbolic values become precise. The use of the sample item does not affect the allocation of global or heap data: it is a local pointer variable allocated on the call stack.

A run-time call is also inserted for each data allocation. Global variables are recorded before the first executable statement of a program (in the *main* function in a C program). Dynamic memory allocation is recorded after each call (e.g. *malloc*). Only those type definitions used by monitored data are recorded. Memory recollection (i.e. free or delete operation) is also recorded. Local variables are recorded at the beginning of a function and recollected at the end of the function. Parameters are recorded before a function call and recollected after the call. Our implementation currently records allocation of only global and dynamic data.

Figure 2 shows an example code segment, which defines a

```
struct s {
  int a[5], *c;
} dat;

int _RecordGlobalData_() {
  struct s* s_sample;

  RecType(s_sample, "struct s", 2, "a", (&(*s_sample).a ), 5, (&(*s_sample).a [1] ), "int",
     "c", (&(*s_sample).c ), 1, sizeof(int*), "int *");

  RecObj("dat", "global variable dat in demo.c at line 5", (&dat), 1, sizeof(struct s), "struct s");
}
```

Figure 2: Example Part 1: data definition. *RecType* records the content of type *s*, which includes the name, size, and offset of each field. The dummy pointer is used to extract exact field offsets. *RecObj* records the name and type of variable *dat*, as well as the source-level location of its declaration.

structure type *s* and a global variable *dat* of that type. The compiler records the type and data definition in a function *_RecordGlobalData_*. It has two monitor calls. The first, *RecType*, records the structure of type *s*, using a local pointer to extract symbolic field offsets. The second, *RecObj*, records the global variable *dat* and its type.

2.1.2 Data Access

Access monitoring has two basic requirements: to capture each memory access and to pinpoint the accessed location. A C program accesses a data element through a sequence of structure, union, or array access operators. The sequence is often called an access path. An access path may contain multiple data references. For example, the path, *a.b.c.d*, contains a single memory reference; but $e \rightarrow f \rightarrow g \rightarrow h$ includes four memory references. A more interesting case is *j.k[i]*, which can be either one or two references depending on whether the type of the *k* field is a static array or a pointer. A compiler distinguishes the two cases by static type analysis.

A compiler records each data reference with a call to *RecAccess*. It has two parameters: the first is usually the starting address of the accessed data unit, and the second is the actual address of the access. We call the first parameter the *base* address of data access. The use of base address allows the run-time system to recognize data units. For example for an array reference *A[10]*, the compiler inserts *RecAccess(A, &A[10])*, where the first parameter is the base address that identifies the data unit at run time. As a result, all elements of array *A* will be managed as a single data unit by the monitor, making it efficient for large objects.

A compiler, however, cannot always determine the base address of the accessed data unit, because an address can be taken from the middle of a data unit and then used as a base for later accesses. For example, a function call, *bar(&A[10])*, would pass a partial array to function *bar*. The solution is to use another monitor call, *RecLink*, which informs the monitor that an internal address is taken and may be used as a basis in later execution. In this example, the compiler inserts *RecLink(A, &A[10])* before the function call. After this point, any access to the parameter array in function *bar* will be recognized as access to array *A*. Section 2.3 shows how the run-time monitor uses *RecAccess* and *RecLink*.

The compiler needs to insert each monitor call at the correct location. The placement becomes a problem in the presence of implicit assignment and control flow in C programs. Consider the statement *A[0][i==0? i+=1:i].a[i]=0*. The value of *i* may be changed. We cannot insert *RecAccess(A, &A[i][i])*, even if we know that *i* is zero upon entering the statement. The example also shows the problem of high-level control flow. An index expression may include conditionals and function calls. We need to insert *RecAccess* and *RecLink* before and only before each data access. In this example, we need to conditionally record the use of *i* in the two branches of the conditional expression.

The compiler solves the problem of implicit assignment and high-level control flow by making them explicit. It transforms implicit assignments into a sequence of single-assignment statements, introducing new temporary variables when necessary. It converts high-level control flows into low-level ones using *goto* and *if-goto* statements. The transformations are well understood as part of traditional compiler code generation. They can be made in different ways. The one in our compiler is adapted from the implementation in the lcc compiler [10].

A longer example is given in Figure 3, which uses the data definitions in Figure 2. The initial version has implicit assignments and control flow, as well as array, structure, and pointer accesses. The transformed version makes control flow and data accesses explicit. It adds two temporary variables. It calls *RecAccess* before each data access and *RecLink* before the last statement, which takes the internal address from an array.

Another minor problem in code generation is that the C language does not allow direct type declaration of array or function pointers (for example, as the type of a return value). The compiler uses *typedef* to circumvent the problem. An example is shown at the beginning of the instrumented version in Figure 3 for the array parameter.

The total number of *RecAccess* in the instrumented version in Figure 3 is 28. The compiler section will show that compiler optimization can remove most of them without losing monitoring accuracy.

```
// Initial version

void foo(struct s A[][5], int *ip, int i) {
  int j, k;
  A[0][i==0?i+=1:i].a[i] = 0;
  j = *ip;
  A[0][j+1].c = ip;
  k = 1+*ip;
  A[0][k].c[0] = 1;
  bar(A[1]);
}

// Instrumented version with no optimization

typedef struct s struct_s_A_5_[5];
void foo(struct_s_A_5_* A, int* ip, int i) {
      {
      int j, k, tmp1, tmp2;
      RecAccess((&i), (&i));
      if ((i != 0)) goto L4;
      RecAccess((&tmp2), (&tmp2));
      RecAccess((&i), (&i));
      tmp2 = (i + 1);
      RecAccess((&i), (&i));
      RecAccess((&tmp2), (&tmp2));
      i = tmp2;
      RecAccess((&tmp1), (&tmp1));
      RecAccess((&tmp2), (&tmp2));
      tmp1 = tmp2;
      goto L5;
   L4:;
      RecAccess((&tmp1), (&tmp1));
      RecAccess((&i), (&i));
      tmp1 = i;
   L5:;
      RecAccess((&A), (&A));
      RecAccess((&tmp1), (&tmp1));
      RecAccess((&i), (&i));
      RecAccess(A, (&A[0][tmp1] .a [i] ));
      A[0][tmp1] .a [i]  = 0;
      RecAccess((&j), (&j));
      RecAccess((&ip), (&ip));
      RecAccess(ip, ip);
      j = (*ip);
      RecAccess((&A), (&A));
      RecAccess((&j), (&j));
      RecAccess(A, (&A[0][(j + 1)] .c ));
      RecAccess((&ip), (&ip));
      A[0][(j + 1)] .c  = ip;
      RecAccess((&k), (&k));
      RecAccess((&ip), (&ip));
      RecAccess(ip, ip);
      k = ((*ip) + 1);
      RecAccess((&A), (&A));
      RecAccess((&k), (&k));
      RecAccess(A, (&A[0][k] .c ));
      RecAccess(A[0][k] .c , (&A[0][k] .c [0] ));
      A[0][k] .c [0]  = 1;
      RecAccess((&A), (&A));
      RecLink(A, A[1] );
      bar(A[1] );
      }
   L3:;
}
```

Figure 3: Example Part 2: program function *foo*. Implicit control and assignment are made explicit. For each data access, *RecAccess* records the base and the address of the access. *RecLink* records the extraction of an internal address from array *A*.

2.1.3 Benefits of High-Level Interface

The source-level interface, consisting of various function calls, has three important benefits: it is independent of the compiler or the target machine; it is expandable; and more importantly, it is efficient because it can be jointly optimized with the rest of the program by a machine compiler. This section discusses the expandability and efficiency aspects.

The interface is expandable because monitor calls can contain any number of parameters. In the current system, *RecAccess* contains not only the address of the access but also the base address of the data unit being accessed, as shown in Figure 3. The base address saves the monitor from the work of finding the accessed data unit. The interface can contain other information for other types of program analysis. For example, it can pass the access type (i.e. read or write) if the monitor wants to track data modification. Another example is to monitor program statements instead of access traces. To do so, a monitor call needs to include all accesses in each statement. Since a statement may have any number of data references, a monitor call may contain any number of parameters.

The interface code is mixed with the program source, so it is generated and optimized by a machine compiler. The parameters of monitor calls will be converted into machine code with optimization such as register allocation. It is important that the interface code and the program code are generated together. For example, when recording an access to *A[10]*, the base address of the array is needed by both the access and the preceding monitor call. A machine compiler would easily save the base address in a register and reuse it in both places. If, however, the interface code is not inserted at source level and not seen by a machine compiler, the extraction of base address is not as easy and may lead to repeated computation and register loading.

Machine-level code generation such as register allocation is different on different machines, especially embedded processors that have unconventional CPU and register architecture (e.g. multimedia instructions and partitioned register files). Generating interface code without using a machine compiler would be a daunting task. Source-level interface allows arbitrary data passing and ensures correctness. Moreover, the interface code is jointly optimized with the program and they can share each other's results. The joint optimization is made possible when programs are compiled together, which means that the monitoring code has to be inserted in program source.

The high-level interface we use is different from past work in binary instrumentation, which inserts monitor code for load and store instructions at the binary level. Binary-level interface does not need program source. The address of data access is readily available in a register. However, if it needs other information such as the base address or the type of access, binary instrumentation will encounter the dilemma of either storing the additional information in memory, which is not efficient, or using registers, which means modifying program register allocation. In addition, code generation for monitor code is compiler and machine dependent. Given significant difference among special-purpose architectures, binary instrumentation is not easily portable.

The high-level interface may increase run-time overhead and register pressure in the instrumented program. It is a tradeoff between storing information in an interface and recomputing the same information at run time. The balance may be different for different problems. The best option is often a mixture of storing and computing. The goal of the framework is to provide these options. As we will see later, the use of base address in *RecAccess* results in significant space saving at run time.

2.2 Compiler Control of Access Monitoring

Not all accesses need to be monitored. The job of the compiler is to select a minimal set of data accesses for monitoring. The selection consists of two orthogonal tasks. The first selects distinct accesses and removes repeated monitoring of the same data. The second selects targeted program segments and removes monitoring in unrelated parts. The following two sections describe these two types of compiler selection. We assume that a program is type correct, as defined at the beginning of Section 2.

2.2.1 Removing Redundant Monitoring

Our base scheme monitors every data access. However, if a variable is accessed repeatedly in a short code sequence, we can record the first access and omit the rest. The problem is as follows. Given two data references, we remove *RecAccess* for the latter if and only if two conditions hold. First, the two data references always access the same data element. Second, they are always executed in a bounded-length sequence. We call the second requirement *bounded-distance guarantee*, since it ensures that any unmonitored access must follow a monitored access to the same location by a bounded distance.

We can maintain bounded-distance guarantee by not optimizing across function calls and back edges in a control flow graph. Hence the guaranteed distance is the maximal length of an intra-procedural path that does not include any call site and any loop structure. Our current implementation is more limited and does not optimize across basic code blocks. The general problem is similar to global redundancy elimination [7, 1], except that we need to bound the distance of all paths between an available expression and a redundant expression.

To detect repeated references to the same data, we expand a version of local value numbering [1]. Our technique has three novel aspects. First, it unifies analysis for both data content and reference location. For example for a reference $a.b$, the analysis gives two value numbers: the first is the value or the content of $a.b$, and the second is its address, *&* $a.b$. The content and the location information complement each other. For example, immediately after an assignment *i=j*, *a[j].b* accesses the same location as *a[i].b*. Here, the content of i partially determines the location of *a[i].b*.

In order to compare locations of data references, the compiler standardizes address expressions. It represents expressions as a sequence of operations with an operator and a varied number of operands. For example, a reference *a[i][j].b* has two operations. The first is an array reference, whose three operands include the base and two subscripts. The second is a field access, whose two operands are the structure base and the field name. The compiler represents operations in the prefix form and uses '#' as a separator. Pointer dereference is considered as a dereference plus an array access to the first element. This allows us to detect the equivalence, for example, between **a* and *a[0]*.

Finally, our technique is performed at the source level. The basic scheme follows the framework used by Halvak, who analyzed source-level values in Fortran programs [12]. We extend it to deal with complex data structures in C. Our technique handles arbitrarily long expressions such as $a.b.c.d$. The source-level information also allows the compiler to understand nested logical structures, information of which would be lost at the assembly level. Source-level analysis is required for high-level transformation to work. Since monitor calls are inserted at the source level, we must perform value numbering analysis in program source. In addition, source-level analysis does not lose precision as assembly-level analysis does. At the assembly level, code generation may introduce (untyped) temporary variables and lose source-level information. For example, a reuse of a register-spilling location may merge two values that are initially separate at the source level.

Figure 4 gives an outline of our value-numbering algorithm. It handles one basic block (single-entry, single-exit code sequence) at a time. The algorithm flushes the value number table at the beginning of each code block and after each function call. After each assignment statement, the left-hand side takes the value number of the right-hand side. It also resets the value number of all possible aliases. In the absence of global alias analysis, we assume all data of the same type are aliases. All pointers are in the same alias set. The only exception are local and parameter variables whose address is never taken in its local function. In that case, their name is their unique identifier.

The example program in Figure 3 tests the strength of our algorithm. Consider the assignment to *A[0][k].c[0]* at the second to last statement. It represents three data references—*k*, *A[0][k].c*, and *A[0][k].c[0]*. Our algorithm can discover that all three have been accessed in the same code block. The detection of the last two is only possible because the algorithm analyzes both the content and location of data references. Figure 5 shows the instrumented program after value numbering and compiler selection (discussed in the next section).

The compiler also removes *RecLink* calls when an internal address cannot escape to the outside. A common example is array traversal through incrementing a pointer. Although the address of each element is taken, it is immediately overwritten and cannot be seen by the remaining execution. In general, compiler liveness analysis can be used to determine the propagation of pointer values. We do not yet handle this general case.

2.2.2 Selection Based on Data and Access Type

A compiler selects monitored data by data location, data type, access location, and access type. This section discusses these four types of selection. They are orthogonal to each other and can be freely combined. The result is a wide range of choices from complete monitoring to partial monitoring.

```
Input: A basic block of C code, all nested code blocks have been flattened
       by renaming block-local variables.
Output: Two value numbers associated with each memory reference:
       one for the value of its address and the other for its content.
       Two expressions are equal if they have the same value number.
Data and data structures:
       variable valTable: a map from a string to a value number.
       variable aliasTable: a map from data type name to expression names
                          that have that type.
       structure ValInfo: represent an access path in five fields.  The first
                          is expression string (exprStr), the second is expression
                          value number (exprVal), the third and fourth are
                          string and value number of the address expression
                          (addrStr and addrVal), and the last field is the data
                          type of the expression (exprType).
       structure AstNode: a node in the abstract syntax tree.

Algorithm:
   procedure CodeBlockValNum(BasicBlock cb) {
     for each statement s in cb
       if (s is an assignment statement)
          ValInfo rhs = ExprAnalysis(s.rhs)
          ValInfo lhs = ExprAnalysis(s.lhs)
          valTable.UpdateValNumber(lhs.exprStr, rhs.exprVal)
          lhs.exprVal = rhs.exprVal
          aliasTable.DeleteAliasValues(lhs.exprType)
       else
          ExprAnalysis(s)
       end if
     end for
   end CodeBlockValNum

   procedure ExprAnalysis(AstNode expr) {
     if (expr is an arithmetic operation or a function call)
        apply ExprAnalysis to operands/parameters
        if (expr is a function call)
           aliasTable.DeleteAllAliasValues()
           info.exprVal = valTable.GetNewValNumber()
        else
           // For associative operators, order operands in canonical order
           // exprStr is a prefix form of the expression, using '#' as separator.
           info.exprVal = valTable.GetValNumber(exprStr)
        end if
        info.addrStr = info.addrVal = empty // no address information
     else // expr is an access path
        if (expr is a variable name)
           info.exprStr = name of the variable
           info.exprVal = valTable.GetValNumber(exprStr)
           info.addrStr = '<addr>#'+exprStr
           info.addrVal = valTable.GetValNumber(addrStr)
        else
           // Apply ExprAnalysis to the closest base of the access path
           //   e.g. the base of a.b.c is a, but the base of a->b->c is *(a->b)
           // Construct exprStr and addrStr in four cases:
           //   taking address, dereference, array access, and structure access.
           //   The deference case is considered as accessing the first element
           //     of an array.  Taking address and dereference operators cancel
           //     each other.
           info.exprVal = valTable.GetValNumber(exprStr)
           info.addrVal = valTable.GetValNumber(addrStr)
        end if
        info.exprType = type of the expression
        if (expr is not a local/parameter variable OR expr has address taken)
           aliasTable.AddToAliasSet(exprType, exprStr)
        end if
      end if
      return info
   end procedure
```

Figure 4: Source-level value numbering algorithm. For an access path, both its value and its address receive a value number. Address expressions are represented in a canonical form.

```
void foo(struct_s_A_5_* A, int* ip, int i) {
      {
      int j, k, tmp1, tmp2;
      if ((i != 0)) goto L4;
      tmp2 = (i + 1);
      i = tmp2;
      tmp1 = tmp2;
      goto L5;
   L4:;
      tmp1 = i;
   L5:;
      RecAccess(A, (&A[0][tmp1] .a [i] ));
      A[0][tmp1] .a [i]  = 0;
      RecAccess(ip, ip);
      j = (*ip);
      RecAccess(A, (&A[0][(j + 1)] .c ));
      A[0][(j + 1)] .c  = ip;
      k = ((*ip) + 1);
      A[0][k] .c [0]  = 1;
      RecLink(A, A[1] );
      bar(A[1] );
      }
   L3:;
}
```

Figure 5: Example Part 3: optimized instrumentation after compiler selection of global and dynamic data and after redundancy removal by value numbering.

We consider data location to fall into one of the three categories: global data, dynamic or heap data, and local or stack data. For each data reference, the compiler uses its symbol table and value-numbering analysis to determine the location of data. At the absence of global pointer analysis, it assumes that any pointer dereference can reference any data except when the content of the pointer is statically known by the compiler. An important use of this feature is to limit monitoring to only global and dynamic data. The difference is shown by the treatment of parameter *ip* in statement *j=*ip* in Figure 5. Its assignment is not recorded because the value resides on the call stack, but its subsequent dereference is recorded because it may point to global or heap data.

After selecting only global and dynamic data and monitoring only distinct data accesses, the number of *RecAccess* statements is reduced from 28 (in Figure 3) to 3 (in Figure 5), a reduction by a factor of 9.

We can monitor different access types, including read, write, and both. The selection is straightforward since each statement modifies at most one data item. The direct left-hand side is the location of a write access, and the rest are read accesses.

We can monitor individual variables or their elements. Two examples are a global variable and the set of data units allocated at a *malloc* call. The compiler analysis needs to determine whether a data reference can refer to targeted data. For programs where all accesses are type correct, our compiler uses symbol and type information to infer the possible locations of a data reference. Different global variable names represent different data. They are also disjoint from local, parameter, and heap data. Under the assumption of type safety, two accesses of two different types cannot refer to the same location, except when they are pointers. For sub-structures, if base structures are of different types, accessed locations are different even if the accessed elements have the same type. For example, two accesses in $a.b.c = g.f$ must refer to distinct locations if a and g are variable names.

Pointer de-references need careful treatment because it may change the content of aliased variables. For example, dereferencing an integer pointer may access any integer element in any data. Except in three cases, all pointer de-references are monitored. The first is when the de-referenced type is different from the monitored type (and it is not a pointer type). The second is when a compiler recognizes the exact content of a pointer, for example, pointer *A[0][k].c* in Figure 5. The third case is when an internal address is never taken from an array or from fields of a structure. In that case, no pointer de-reference can reach those internal data. Our experience shows that these three cases cover many removable de-reference monitoring in practice.

2.3 The Run-Time Monitor

A transformed program invokes the run-time monitor through the high-level interface consisting mainly of four types of function calls: *RecType* for a type definition; *RecObj* for a data allocation; *RecAccess* for a data access; and *RecLink* for an extraction of an internal address.

For each structure definition that appears in *RecType*, the monitor records all its fields including those in nested structures. For each field, it records the name and offset. The field offset may differ depending on the machine compiler. However, at run time, the offset is fixed and is recorded precisely. The monitor stores all structure definitions in a type table. For each data unit in a *RecObj* call, the monitor creates a record we call *shadow*, which contains its memory address and a pointer to its structure definition in the type table. For fast retrieval of shadow data, we store them in a hash table indexed by the starting address of data units. The size of the hash table does not depend on the amount of data inside structures or arrays. For example, an array may contain a million elements, but it needs only a single entry in the hash table.

Access recording in *RecAccess* has two steps: hash-table search to find the shadow data, and type-table search to locate the accessed element. The first parameter of *RecAccess* is used in hash-table search. It is either the starting address of a data unit or an internal address. The hash entry is initialized by *RecObj* in the first case and by *RecLink* in the second case. Recall that *RecLink* happens before a program takes an internal address from a data unit. At *RecLink*, the monitor inserts the extracted address into the hash table and links it to the shadow record of its data unit. In the worst case, a program stores the address of every data element, and the hash table has one entry for each data element. However, our experience shows that a program usually takes at most a constant number of internal addresses from any data unit. In exceptional cases, a compiler can choose not to monitor certain data to avoid the run-time explosion of the hash table. Compile-time analysis can conservatively identify these cases.

The monitoring is precise. There is no unknown pointer at run time. In addition, pointer dereferencing is no different from accessing the first element of an array variable. For example, *a[0]* and *(*a)* are identical from the view of the monitor.

With an additional cost, the monitor can support unsafe C programs or binary code with no program source by using the same approach of binary instrumentation. The compiler inserts *RecAccess* before each memory operation. The monitor organizes the address range of all data units in a tree and search for any address in logarithmic time, as often used in profiling analysis. In our scheme, the search is only needed when the base address of an access is not in the hash table. However, the cost is still significant because every access may need this search in the worst case. In addition, the cost of redundant *RecAccess* becomes more expensive. In type-safe programs, we can abort the monitoring upon failing the hash-table search. For unsafe programs, however, we must then search through all data units. In our current work, we consider only type-type programs. However, the framework can be extended to support unsafe programs.

The run-time monitor is adjustable. It supports compiler selection from monitoring every data at every access to monitoring a single element at a single program point. It supports three levels of precision: collecting memory addresses without hash-table and type-table search, recognizing data units with hash-table search, and locating data elements with both hash-table and type-table search. It may use different precision for different data at the instruction of the compiler.

3. PRELIMINARY EVALUATION

This section measures the efficiency of data monitoring for two test programs, which use standard data structures including an array, a tree and a hash table.

3.1 Implementation

We have implemented the monitoring system as described before. The compiler uses the lcc compiler as front-end [10]. We modified lcc to generate source code. We adapted its code generation to make assignment and control flow explicit. The generated code is fed into our compiler, which implements value numbering and supports selective monitoring as described in Section 2.2. Our compiler generates instrumented code in C. The run-time monitor is implemented as described in Section 2.3.

3.2 Two Test Programs

The first program is quicksort. It is a well-known program that has irregular access to regular data. The primary data unit is a dynamically allocated array. The access is input dependent: at each step, it partitions a sub-array based on a seed element. We used the program written by Xiao et al., which has 107 lines of C code [18]. The input to quicksort is a randomly generated array containing 100K numbers of type long long.

The second program is *Cheetah* simulator, which measures capacity misses of fully-associative cache of all sizes for an access trace [17]. It uses a balanced binary tree (splay tree) and a hash table to sort and locate past accesses. The program uses arrays, structures, pointers, and dynamic memory allocation. At the abstract level, their implementation is similar to our monitor system. *In a sense, we are monitoring the monitor.* We could analyze our own monitor code, but we choose *Cheetah* because it is familiar to other people and readily available on the Internet. The program has 718 lines in its main program file and 2287 lines counting all user-written header files. The input to *Cheetah* simulator is an access trace from *JPEG* encoding of a 2.5KB test image.

Our monitoring system has been applied to a number of other C programs including small test cases from lcc distribution, an image-processing benchmark from DoD, and a N-body simulation program. The largest instrumented program for which we recently completed a correct code generation is a 20-thousand-line *JPEG* program from Mediabench. We do not report these programs because we have not carefully analyzed them. We use a version of instrumented *JPEG* to collect access trace and then feed it into *Cheetah*.

3.3 Cost of Monitoring

As reported by our compiler, quicksort has no global variables but a dynamically allocated array, and *Cheetah* has 32 global variables and 8 dynamic allocation sites. No internal address is taken in quicksort but *Cheetah* has eight such operations.

Table 1 lists the performance of different monitoring configurations. The first column lists different levels of monitoring. The second column measures the number of *RecAccess* calls inserted in program source. The other columns measure the run-time cost in terms of slowdown factors. A slowdown of two means that the monitored program runs twice as long as the original version. Because of the source-level instrumentation, we can measure programs on any machine with a C compiler. Here we collect the timing result on two platforms. The first is a 250MHz MIPS R10K processor, using SGI MIPSpro compiler with full optimization (*-n32 -Ofast*). The second is a 1.7 GHz Pentium 4 processor, using gcc compiler with *-O3*.

The upper and lower parts of Table 1 list data for quicksort and *Cheetah* respectively. The rows in each part represent different monitoring coverage while the the columns stand for different monitoring accuracy. The monitoring coverage includes monitoring all data, global and dynamic data, global and dynamic data with redundancy elimination, modification of global and dynamic data with redundancy elimination, no data, and for *Cheetah*, the *nxt* field of its hash table. The monitoring accuracy includes collecting only memory address (marked by *v1*), recognizing the data object (*v2*), and locating the accessed element in its source-level form (*v3*). The first one is equivalent to source-level trace profiling, except that we do not store the trace. The second one is equivalent to safety checking. The last version is the most precise, where we know exactly the location of access in its source form.

For quicksort, the compiler inserts 0 to 108 *RecAccess* calls for different coverage of monitoring. On 250 MHz MIPS R10K, the cost ranges from 4% in no monitoring to 19.7

Versions of selective instrumentation	number of *RecAccess*	MIPS R10K, 250MHz v1	v2	v3	Intel Pentium 4, 1.7 GHz v1	v2	v3
Quicksort, sorting 102,400 random integers							
All data	108	1.08	7.46	19.7	6.00	19.5	26.5
Global + dynamic	13	1.08	2.13	5.73	1.75	3.75	8.75
Global + dynamic, optimized	11	1.08	1.93	4.88	1.75	3.50	7.25
Data writes only, optimized	5	1.08	1.35	2.08	1.25	1.75	3.25
No monitoring	0	1.04			1.00		
Cheetah, simulating JPEG encoding of a 2.5KB image							
All data	691	1.42	11.0	39.9	11.2	41.1	83.4
Global + dynamic	349	1.41	4.72	16.3	4.59	13.7	51.0
Global + dynamic, optimized	269	1.40	4.12	13.3	3.78	11.2	44.7
Data writes only, optimized	94	1.40	2.55	5.57	2.10	4.76	16.9
nxt field in *hash_table*	12	1.09	1.10	1.11	0.98	1.05	1.05
No monitoring	0	1.09			1.02		

v1: collecting address trace, v2: finding data unit, v3: finding data element

Table 1: Cost of selective monitoring in two programs on SGI and Intel machines. The numbers are slowdown factors.

in precise monitoring of all data. At *v3*, the program runs five times slower if monitoring global and dynamic data and 2 times slower if monitoring just their modifications. At a less precision, *v2*, the slowdown is less than two for monitoring global and dynamic data. The cost of collecting address trace in *v1* is 8% for all cases. Without any monitor calls inserted, the program runs 4% slower because of the transformation to explicit assignment and control flow. The same transformation does not increase running time when compiled with gcc and running on 1.7GHz Pentium 4. However, the slowdown is larger when any monitor call is inserted, suggesting that gcc does not optimize the inserted code as well as the SGI compiler. The slowdown factors range from 1.25 to 6 at *v1*, 1.75 to 19.5 at *v2*, and 3.25 to 26.5 at *v3*.

Cheetah is more difficult to monitor because of its complex data structures. The number of *RecAccess* calls ranges from 0 to 691 in different versions. Programs run 40% to 42% slower when collecting memory address on the MIPS processor. The slowdown factors range from 10% to a factor of 11.0 at *v2* and from 11% to a factor of 40 at *v3*. The slowdown is two to four times larger on the Intel processor in most cases. Again, we suspect that the reason is due primarily to the different strength of optimization in SGI and gcc compilers.

Monitoring the *nxt* field in hash table data in *Cheetah*, however, has consistently low cost. As the second to last row shows, the cost of most precise monitoring is 11% on MIPS and 5% on Pentium. They suggest that run-time analysis for this subset of data is practical for actual use if a user is willing to tolerate 10% performance loss. When collecting only memory address, the instrumented version runs faster than the original version by 2% or 0.01 second on the Pentium 4 processor. It is possible that the inserted code triggered faster optimization in the gcc compiler.

3.3.0.1 Benefit of High-Level Interface

A key feature of our system is the source-level interface, as discussed in Section 2.1. It makes our system both machine and compiler independent and allows us to experiment with two different machines and two different compilers. In addition, we now show the space saving enabled by this interface. In *RecAccess*, the compiler supplies the base address of referenced data in addition to the actual address of the reference. The use of a base address allows for a smaller look-up table at run time. The table is indexed by the base address instead of the actual memory address. Hence the size of the look-up table is determined by the number of data units (and number of exposed internal addresses), not by the number of data elements. A smaller table occupies less space and makes table search more efficient. In this experiment, we measure the size of the look-up table in the two test programs. Quicksort has an array of 102,400 elements being monitored. As expected, the look-up table in the monitor contains only one entry. *Cheetah* has a total of 1,171,157 elements being monitored in its tree and hash-table data. Most data are allocated dynamically. Internal data address is taken in eight places in the program. Throughout the execution, the look-up table contains a maximum of 17,730 entries. The size of the look-up table is smaller than the size of data by a factor of 66. In both programs, the high-level interface enables dramatic reduction in terms of the run-time space overhead. We do not measure the impact in execution time, but we expect it to be significant given the large reduction in space cost. Optimizing hash-table search is critical in data monitoring because it accounts for 50% to 75% of monitoring time, as reported in Table 1.

3.3.0.2 Summary

The experimental results of two programs on two machines show a wide range of choices between monitoring coverage, precision and cost. The difference is up to a factor of 49 by static count and a factor of over 80 by running time. The base cost for enabling data monitoring is 4% in quicksort and 9% in *Cheetah*. The lowest monitoring cost is no more than 25% in quicksort and 11% in *Cheetah*. In addition, the high-level interface reduces the space cost of the monitor by orders of magnitude. Next, we describe two applications: one uses complete monitoring at profiling time, and the other uses partial monitoring at run time.

4. APPLICATION IN DATA OFF-LOADING

This section discusses the use of selective monitoring in data off-loading. Embedded systems need to reduce the size of physical memory to minimize manufacture cost. They also need to reduce the size of active memory to minimize energy consumption. Both tasks involve relocating certain program data either to a off-device network or to a on-device but sleeping memory module. This section first uses profiling analysis to find infrequently used data and then uses run-time monitoring to support off-loading of those data. Both use our afore-described monitoring system. We use the *Cheetah* program as an example.

4.1 Profiling Analysis of Data Access Pattern

Our system can collect data accesses for each element of global and dynamic data. To analyze the access pattern, we link the monitor to an analyzer [19], which measures reuse distance along with access frequency. The analyzer generates results for each global and dynamic variable. Dynamic data are grouped by their allocation site. Table 2 shows part of the output. For brevity, it omits the majority of the output including reuse-distance information.

The analysis captures all global and dynamic data. As shown by the table, the program allocates hash-table entries in two places. The first is in global array *slot*, and the second is through a dynamic allocation site. The analysis also finds difference in access pattern. The *lft* field of tree nodes accounts for 13% of all accesses, but the *nxt* field of hash-table entries accounts for only 0.11% in static entries and an additional 0.01% in dynamic entries. Other fields such as *grpno* of tree nodes and *grptime* of hash-table entries are not used at all by the program. The results suggest that we can reduce memory size by off-loading unused or infrequently used data.

4.2 Run-Time Monitoring

Data monitoring enables data off-loading at fine granularity. The hash table of *Cheetah* initially consists of an array *slot*. Each array element contains five fields. Two of them, *grptime* and *prty*, are never used. The *nxt* field is accessed only 0.11% of the time. Suppose a programmer wants to off-load these three fields. To maintain correctness, the program must monitor all accesses to these three fields. When an off-loaded element is needed by the program, the system must locate the element and re-load it from secondary storage. Precise monitoring allows precise re-loading: given an access to a *nxt* field, the system calculates the index in *slot* array from its shadow record in the hash table.

The additional space cost for the entire array is only a single entry in the hash table in the monitor. The time cost, as described in Section 3, is 11% on an SGI workstation and 5% on a PC. Therefore, off-loading saves 60% of space for *slot* array with no more than 11% of run-time overhead. Data off-loading can also reduce space requirement for dynamically allocated hash-table entries. Although it requires a larger hash table, the overall memory requirement can still be smaller if the size of shadow data is smaller than the size of off-loaded data.

Three properties of the monitoring system are critical for this application. The first is bounded-distance guarantee. It ensures that a program always finishes using a re-loaded element in a bounded code sequence, so that it needs not to store re-loaded elements for a long time. The second property is selectivity. Since only partial data are off-loaded, the monitor needs to look only at program accesses that may reference displaced data. All other memory accesses proceed as before with no run-time overhead. The last important feature is high-level interface. The use of a base address makes hash table efficient, which in turn allows for fast and precise monitoring for each data access.

Our estimate of 60% space saving at 11% time cost is only an estimate. We have made several simplistic assumptions. We did not include the cost of changing data definition and data access to hash table entries and the cost of communication between the program and the remote data storage. However, we show that selective monitoring supports automatic data control with possibly little run-time overhead. In fact, the cost of monitoring is proportional to the usage of monitored data. The less frequently used are the off-loaded data, the less run-time overhead is the run-time monitoring. Therefore, selective data monitoring fits well with the need of data off-loading.

5. RELATED WORK

Program instrumentation and data analysis have been an active area of research as well as commercial development. Most widely used are instrumentation at the binary level. Binary instrumentation does not require program source, but it is machine and compiler dependent. For data analysis, it collects the content of the address register in each memory load and store operation. It often uses a search tree to locate data accesses. Our system uses source-level instrumentation. It does not need to instrument all data references and it can be selective based on data identity and access type. It uses a high-level interface, which allows the exchange of non-trivial program information and the use of hash-table search at run time. The source-level instrumentation is portable across machines and compilers, although it analyzes only source-level data and their accesses.

Source-level instrumentation has been used in studying Fortran programs. Early examples include source-level profiling to examine maximal parallelism by Chen et. al. [6] and program locality by Callahan et. al. [5] and recently by Mellor-Crummey et. al [15]. They analyze array data without user-defined types and explicit pointers. Their goal was profiling, so they needed not to trade coverage for efficiency. To support run-time optimization, our earlier work used selective analysis in Fortran and simple C programs [8].

Selective analysis has been used at the assembly level. Most widely used is a technique called abstract execution developed by Larus [14]. For data monitoring, it instruments only statically unknown addresses and control flow. After execution, it regenerates the full trace from the abbreviated run-time trace. The instrumentation happens on an intermediate form after most compiler optimizations but before peephole optimization. For a set of four test programs on a MIPS R2K processor, profiling overhead was 0.8 to 4.8 times of the cost of program execution (up to 80% was writing files to disk), and the succeeding regeneration step was 19 to 60 times slower than the original. The abbreviated

Source-level data name	Access frequency
All accesses	100% (178 million)
Dynamic allocation of type tree_node in faclru.c at line 661	40%
structure field tree_node.addr	1.5%
structure field tree_node.lft	13%
structure field tree_node.rt	7.4%
structure field tree_node.grpno	no access
other fields omitted	
Global array slot of type hash_table in faclru.c at line 41	2.6%
structure field hash_table.nxt	0.11%
structure field hash_table.grptime	no access
structure field hash_table.prty	no access
other fields omitted	
Dynamic allocation of type hash_table in faclru.c at line 404	0.31%
structure field hash_table.nxt	0.01%
other fields omitted	
Global variable md_amode_str in ./machine.h at line 477	no access
Other variables and allocation sites omitted	

Table 2: Partial results from data-reuse analyzer

trace was a facotr of 10 to 40 smaller than the full address trace. In comparison, the time overhead of our system is 8% and 42% when collecting a full address trace (without writing out to files) of two different programs on a much faster MIPS processor. We do not need a regeneration step. Larus also selected data references based on whether the data are in registers or program stack. We support more fine-grained selections at the source level such as those based on data identity and data type. Larus' method can trace events other than data access. We focus on only data monitoring.

Orthogonal to data selection is control selection. A recent scheme used by Arnold and Ryder [2] and by Hirzel and Chilimbi [13] makes two versions of the program and let execution switch between lightly unmonitored version and heavily monitored version at selected times. Our work complements control-selection schemes by allowing selection based on data, not just on control.

Our compiler algorithms are adapted from classic code generation and local value numbering [1]. Our compiler operates at the source level similar to the framework used by Halvak for Fortran programs [12]. We handle complex data structures in C. Our value numbering algorithm analyzes both the content and location of data references. It guarantees bounded distance when removing monitoring for repeated accesses. It preserves and utilizes the full semantics of source-level data definition and data access. For type-safe programs, we use type-based disambiguation similar to Diwan et. al. [9]. We assume that locations are different for data of different types except for pointer values.

Access monitoring has also been used in program safety checking. Steffen [16] and Austin et. al. [3] augmented each pointer with the size and location of the targeted object. At each pointer de-reference, a simple run-time system checks whether the access is within the allowed range. The scheme adds significant space overhead (15 bytes per pointer). For a set of large C programs, Austin et. al. reported time overhead up to 540% and space overhead up to 330% [3]. In comparison, the time overhead of our system is 93% and 312% on MIPS for the same monitoring accuracy. Our space overhead is mostly proportional to the number of data objects instead of the number of pointers. Our system is more precise because it keeps complete data information including type definitions. Austin et al. implemented a system that monitored every memory access to every source-level data. They used run-time optimization, which did not have bounded-distance guarantee. We use selective analysis and have more control over the trade-off between coverage and cost.

Another approach to access checking is through a run-time system that keeps track of all live memory objects, as in conservative garbage collectors by Boehm [4] and in commercial software such as Purify [11]. These methods do not use source-level compiler support. Although applicable to any binary code, they cannot have the benefit of source-level selection and high-level compiler optimization. Both on purpose and by design, access-checking methods do not support partial checking.

The last class of related work we compare with is optimization for Java programs. Java language enforces stricter constraints on the shape of data and the type of their access. It cannot have nested structures and indirect pointers (pointers that point to pointers). Java byte code is also much heavier than typical program binary because the byte code needs to store type and symbol information for each class. In contrast, our system supports a wider range of data structures and allows for analysis and instrumentation of only parts of a program instead of its entirety.

6. SUMMARY

We have presented a selective data-monitoring framework that consists of a compiler and a run-time monitor. The compiler selects monitoring points based on data identity, data type, and access type. The monitor examines compiler-selected data accesses in different precisions. Three techniques are vital. The first is combined compiler analysis of both content and location of data accesses. The second is the use of a high-level interface. Enabled by the compiler and the high-level interface, the last technique is the use of base address and hash-table search by the monitor. An important application of selective monitoring is automatic data off-loading on memory-limited devices.

Preliminary experiments on both an array-intensive and a pointer-intensive program on a workstation and a PC shows that selective monitoring provides a wide range of choices in terms of coverage and precision. The cost differs up to a factor of 49 by static count and a factor of over 80 by running time. The lowest monitoring cost is no more than 25% in quicksort and 11% in *Cheetah*. In addition, the high-level interface reduces the space cost of the monitor by orders of magnitude. These results suggest that selective monitoring can be efficient for practical use in applications such as data off-loading. We hope that this research will provide a new tool for managing complex and dynamic data in general-purpose programs.

Acknowledgment

We thank the students of Spring 2001 class of CS573, who used an earlier version of the monitoring system and provided valuable feedback. We also thank James Larus, whose comments helped us to significantly improve the presentation of this work. The SGI machine used in this work was purchased from an NSF infrastructure grant. The Pentium PC was donated by Intel.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, 2001.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, Florida, 1994.

[4] H. J. Boehm. Space efficient conservative garbage collection. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Analyzing and visualizing performance of memory hierarchies. In *Performance Instrumentation and Visualization*, pages 1–26. ACM Press, 1990.

[6] D.K. Chen, H.H. Su, and P.C. Yew. The impact of synchronization and granularity in parallel systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

[7] J. Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, 5(7):20–24, 1970.

[8] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[9] A. Diwan, K. McKinley, and E. Moss. Type-based alias analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, 1998.

[10] C. Fraser and D. Hanson. *A retargetable C compiler: design and implementation*. Benjamin/Cummings, 1995.

[11] R. Hastings and B. Joyce. Purify: fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, 1992.

[12] Paul Havlak. *Interprocedural Symbolic Analysis*. PhD thesis, Dept. of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.

[13] M. Hirzel and T. M. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proceedings of ACM Workshop on Feedback-Directed and Dynamic Optimization*, Dallas, Texas, 2001.

[14] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software - Practice and Experience (SPE)*, 20(12), 1990.

[15] J. Mellor-Crummey, R. Fowler, and D. B. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, 2001.

[16] J. L. Steffen. Adding run-time checking to the portable C compiler. *Software Practice and Experience*, 22(4), 1992.

[17] R. A. Sugumar and S. G. Abraham. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical report, University of Michigan, 1993.

[18] L. Xiao, X. Zhang, and S. A. Kubricht. Improving memory performance of sorting algorithms. *ACM Journal on Experimental Algorithmics*, 5:1–23, 2000.

[19] Y. Zhong, C. Ding, and K. Kennedy. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington DC, March 2002.