Runtime Detection of C-Style Errors in UPC Code

Peter Pirkelbauer
Lawrence Livermore National Laboratory
peter.pirkelbauer@llnl.gov

Chunhua Liao
Lawrence Livermore National Laboratory
liao6@llnl.gov

Thomas Panas
Microsoft
Parallel Data Warehousing
tpanas@microsoft.com

Dan Quinlan
Lawrence Livermore National Laboratory
dquinlan@llnl.gov

Abstract

Unified Parallel C (UPC) extends the C programming language (ISO C 99) with explicit parallel programming support for the partitioned global address space (PGAS), which provides a global memory space with localized partitions to each thread. Like its ancestor C, UPC is a low-level language that emphasizes code efficiency. The absence of dynamic (and static) safety checks allows programmer oversights and software flaws that can be hard to spot.

In this paper, we present an extension of a dynamic analysis tool, ROSE-Code Instrumentation and Runtime Monitor (ROSE-CIRM), for UPC to help programmers find C-style errors involving the global address space. Built on top of the ROSE source-to-source compiler infrastructure, the tool instruments source files with code that monitors operations and keeps track of changes to the system state. The resulting code is linked to a runtime monitor that observes the program execution and finds software defects.

We describe the extensions to ROSE-CIRM that were necessary to support UPC. We discuss complications that arise from parallel code and our solutions. We test ROSE-CIRM against a runtime error detection test suite, and present performance results obtained from running error-free codes. ROSE-CIRM is released as part of the ROSE compiler under a BSD-style open source license.

1. Introduction

A 2002 NIST study on the economic impact of software flaws (bugs) [18] reports that about half of a software project's budget is spent on testing. The same study estimates that software bugs cost the US economy 59.5 billion dollars (0.6% of 2002's GDP) annually. Software flaws cause programs to behave in undesirable ways. They originate from oversights and human misjudgments in the software development life cycle. Software flaws also arise from operations exhibiting undefined or unspecified behavior. Such behavior describes operations where the language designer deliberately did not specify the behavior.

Software frequently contains flaws, and the size and complexity of modern software makes the detection of such flaws difficult. Reducing the number of software flaws is an effort that permeates all stages in software development. For example, adopting rigid coding standards and restricting programming languages to a safe subset can eliminate error-prone coding techniques. Static and dynamic code analysis tools that operate on source or binary code are also used to reduce software flaws. Static analysis tools utilize source code (style) analysis or formal analysis techniques such as dataflow analysis, abstract interpretation, and model checking. Static analysis tools do not influence the running programs but often state space, implementation techniques, and language semantics make precise analysis intractable. Dynamic analysis tools find bugs by observing the behavior of running programs. This is typically accomplished by code instrumentation (source or binary) or by replacing (built-in) library functions (e.g., malloc) with custom implementations. Dynamic analysis operates with concrete values and is not prone to combinatorial state explosion. The downside of dynamic analysis tools is that monitoring running programs impacts performance. The quality of the results depends on the tests' input data and covered program paths.

Unified Parallel C (UPC) [22], an extension of the C programming language for parallel programming, is prone to vulnerabilities inherited from its C ancestor (e.g., array out of bounds accesses, dereferences of dangling pointers, the use of uninitialized variables, arithmetic overflow and underflow, string termination, etc.) and vulnerabilities introduced by parallel programming (e.g., race conditions, dead locks, etc.). While safer programming language, compiler, and tool support are desirable, economic, performance, compatibility, and portability reasons prevent safety mechanisms from receiving more attention.

In this paper we present an extension of an existing dynamic analysis tool, ROSE-CIRM, for UPC. The previous ROSE-CIRM tool [20] monitors ISO C and ISO C++ programs and detects runtime errors. In this work, we extend ROSE-CIRM to find similar C-style errors in UPC code. The presented tool is able to identify uninitialized memory accesses, dangling pointers, erroneous heap allocation/ deallocations, and out of bounds array accesses in parallel programs using UPC's shared memory space. We tested our tool for its accuracy and overhead using the RTED benchmark test suite [14] for UPC and a few error-free sample programs [7].

The contributions of this paper include 1) a first monitored execution environment for UPC, 2) compiler-based automatic code instrumentation of UPC source code, 3) a runtime library to record and check UPC programs' state, 4) evaluation based on a runtime

error benchmark suite, and 5) analysis of the runtime monitor's overhead and optimizations that reduce its overhead.

The rest of this paper is structured as follows: §2 provides background information on the Unified Parallel C (UPC) language and ROSE-CIRM for C/C++. §3 describes our extensions to ROSE-CIRM for UPC. §4 discusses the programs tested by our tool and obtained results. §5 gives an overview of other dynamic tools for detecting bugs. §6 concludes the paper and provides an outlook on future improvements.

2. Background

This section provides background information on the UPC language, on the ROSE compiler, and on our ROSE-based runtime error detection tool implementation for C and C++.

2.1 Unified Parallel C

The PGAS programming model offers a global memory space with localized partitions to each thread. UPC extends the ISO C99 programming language with parallel constructs for the partitioned global address space. Fig. 1 depicts a typical memory layout for a UPC program.

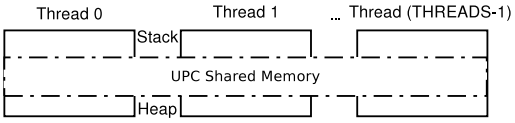

Figure 1. UPC memory layout

A UPC program consists of THREADS number of threads. Each thread contains its own stack and heap, where it has exclusive access. In addition, each UPC thread owns a portion of the global address space (shared memory). Any thread can access the entire shared heap using random reads and writes.

To control which data is private and which data can be accessed from other UPC threads, UPC introduces a new type qualifier (`shared`). For arrays, UPC allows the specification of a blocking factor, that determines how many consecutive elements are stored within the same shared region. Consider the following example:

```
1 shared int i; // a shared integer variable
  shared[2] int arr[4*THREADS]; // uses blocks of integers
```

The first declaration allocates a single integer variable in the shared space (within Thread 0). The second declaration defines an array of integers with 4*`THREADS` elements (The built-in constant `THREADS` returns the number of UPC threads). Each thread owns four elements of the array. The blocking factor is 2, thus the first and second element are located in Thread 0, the third and fourth in Thread 1, and so on.

UPC offers three kind of pointers: 1) C-style pointers (private to private) point to an address within an UPC's local address space, including its local portion of the shared space; 2) pointers to shared memory (private to shared) can address the entire global space (usable only by the owning thread); and 3) shared pointers to shared memory (shared to shared) can be used by any UPC thread. Example UPC pointers are shown below:

```
  int* private_to_private;
2 shared int* private_to_shared;
  shared int* shared shared_to_shared;
```

Three functions in UPC are available for dynamically allocating memory in the shared space. `upc_alloc` allocates all memory within the shared memory of a single thread, `upc_global_alloc` distributes the memory according to a specified blocking factor, and `upc_all_alloc` is a collective version of `upc_global_alloc`.

```
1 shared[2] int arr[4*THREADS];
  shared const int len = sizeof(arr)/sizeof(arr[0]);
3 // a[i] is initialized by the thread owning a[i]
5 upc_forall(int i = 0; i < len; ++i; &a[i]) a[i] = rand();
7 // a[i] is initialized by Thread (i % THREADS)
  upc_forall(int i = 0; i < len; ++i; i) a[i] = rand();
9 // all threads execute the loop body
11 upc_forall(int i = 0; i < len; i+=THREADS; ) a[i+MYTHREAD] = rand();
```

Figure 2. UPC parallel for loop

A locking mechanism is provided by UPC to protect critical regions. UPC defines the opaque type `upc_lock_t`, together with operations to enter (`upc_lock`, `upc_lock_attempt`), exit (`upc_unlock`), allocate (`upc_global_lock_alloc` and the collective version `upc_all_lock_alloc`), and deallocate (`upc_lock_free`).

UPC threads can be synchronized with `upc_barrier`. UPC also provides the pair `upc_notify` / `upc_wait` implementing a split phase barrier. Split phase barriers help writing code that overlaps computation and communication. Memory consistency can be controlled using `upc_fence`. In addition, UPC can control the memory consistency of files, scopes, and objects using pragmas and qualifiers.

UPC provides a parallel loop statement (`upc_forall`). `upc_forall`'s syntax extends C's `for` loop syntax with a fourth argument (the affinity expression) controlling which UPC thread will execute a specific loop iteration. Fig. 2 shows three versions of a `upc_forall` loop initializing all elements of a globally-declared shared array with random values. Each array element is written exactly once. In the first version, the thread whose shared memory owns `a[i]` executes the assignment. In the second version, the thread is determined by the result of the modulo division `i%THREADS`. In the last version, the loop counter is incremented in steps (`THREADS`). On the omission of an affinity expression, all threads execute each loop iteration. Note `MYTHREAD` returns the UPC thread number.

2.2 The ROSE Compiler Infrastructure

ROSE [19], developed at Lawrence Livermore National Laboratory, is an open source compiler infrastructure to build both source and binary program transformation and analysis tools for large-scale Fortran 77/95/2003, C, C++, OpenMP, and UPC applications.

As shown in Fig. 3, ROSE can process both source code and binary executables (x86, Power-PC, ARM instruction sets) as input with help from source parsers (EDG [6] and Open Fortran Parser [4]) and binary disassemblers (such as IDA Pro [10]). Leveraging the EDG C++ frontend, ROSE currently supports all UPC constructs as defined in UPC 1.2, and non-standard Berkeley UPC extensions (defined in the UCB bupc_extensions.h)

Internally, ROSE generates a uniform abstract syntax tree (AST) as its intermediate representation (IR) for both source and binary input. Sophisticated compiler analyses and transformations are developed on top of the AST and encapsulated as simple function calls, which can be readily reused by tool developers. Ex-

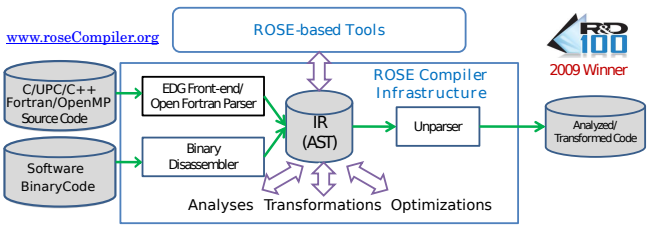

Figure 3. ROSE compiler infrastructure

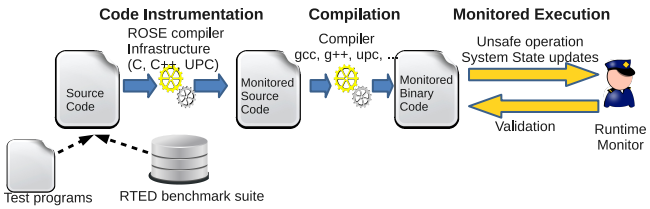

Figure 4. The ROSE-CIRM tool architecture

ample analyses include control flow analysis, data flow analysis, call graph analysis, and dependence analysis. Representative transformations include code instrumentation, inlining, outlining, loop transformation, and automatic parallelization.

ROSE is particularly well suited for building custom tools for static analysis, program transformation, and optimizations, etc. The ROSE team was awarded the prestigious R&D 100 Award in 2009, for significant contributions of making compiler techniques more accessible. ROSE is released under a BSD-style license.

2.3 ROSE-CIRM Tool

ROSE-CIRM [20] is a dynamic analysis tool that consists of a source code instrumentation tool built using ROSE and a runtime library. Fig. 4 illustrates the workflow of using ROSE-CIRM, including instrumentation, compilation, and monitored execution.

During code instrumentation, ROSE-CIRM identifies a program's source locations that require interaction with the runtime monitor and inserts calls to the runtime library. Such locations include statements declaring and accessing variables, places where scopes are entered and exited, locations allocating and deallocating memory, and places where functions are invoked, etc. At these locations, the runtime functions are inserted to record changes to the system state and/or check if there are any errors. Using source code instead of binary code allows the checking mechanism to utilize high level type information such as array bounds.

The code instrumentation phase is implemented through the transformation of an input program's AST. ROSE-CIRM uses the top-down AST traversal mechanism in ROSE to visit all nodes. Depending on a node's type and its context, source locations where instrumentation is needed will be identified and corresponding function calls will be inserted.

Fig. 5 illustrates the elementary mechanism: transformations are implemented by an `InstrumentationVisitor`. The example shows operations on ROSE basic blocks (i.e., a statement sequence enclosed by {}) and variable references. Basic block entries and exits are instrumented to keep track of local variables' lifetime. Variable references are instrumented to keep track of uninitialized variables. Hereby, the function `readWriteContext` determines whether a variable is used in a read or write context.

The instrumentation inserts calls to the runtime monitor. Table 1 shows a representative subset of the runtime monitor's interface

```
1 struct InstrumentationVisitor {
    ...
3   void handle(SgBasicBlock& block) {
      instrument_block_entry(block);
5     instrument_block_exit(block);
    }
7   void handle(SgVarRefExpr& var) {
9     Context ctx = readWriteContext(var);
      if (ctx) instrument_variable_access(var, ctx);
11  }
};
```

Figure 5. Instrumentation pass

Function	Action	Description
Dynamic Memory		
cirmAllocMem	U	Saves data about a heap allocation.
cirmFreeMemory	UC	Checks memory deallocations.
Stack Memory		
cirmCreateVariable	U	Keeps track of variables.
cirmInitVariable	U	Marks a memory location as initialized.
cirmAccessVariable	C	Checks that a memory location exists and is initialized.
cirmEnterScope	U	Record a new scope context (upon function or block entry).
cirmExitScope	UC	Deletes a scope context and all memory associated with it. Check for dangling pointers to disappeared memory.
Arrays and Pointers		
cirmCreateArray	U	Saves data about arrays allocated on the stack.
cirmAccessArray	C	Checks if the access is within array bounds. It also optionally checks if location has been initialized.
cirmMovePointer	UC	Validates a pointer update.
Others		
cirmRegisterTypeCall	U	Saves data about the structure of user defined data types.
cirmClose	C	Closes the runtime monitor.

Table 1. Representative runtime-monitor interface for UPC

supporting dynamic analysis of dynamic memory, stack memory, arrays and pointers, etc. The first column of Table 1 shows function names and the last columns shows function descriptions. The second column indicates whether the function is a notification on a system state update ("U"), a runtime check ("C"), or both ("UC"). Calls to state-updating functions are inserted after the state update operation. Calls to runtime checking functions are inserted before the potentially harmful operation executes.

The code in Fig. 6 demonstrates how the code instrumentation embeds interaction with the runtime monitor (displayed in *italic*) in user code. For illustration purposes, the code omits some details such as how ROSE-CIRM tracks type information, location information, and loop variables. The code uses a function `compute` (at line 3) to calculate elements of a dynamically allocated array. Then the function `sum` (at line 14) calculates and prints the sum of all elements. Note that the implementation contains a bug in line 6: Variable `i` is initialized to `1` instead of `0`. The value of `values[0]` remains undefined. Function `sum` uses this undefined value later on.

Instrumenting `compute` adds calls to `cirm_EnterScope` and `cirm_ExitScope` to keep track of the creation and destruction of variables and parameters (e.g., at lines 4 and 11). The call of `cirm_CreateVariable` (at line 3) notifies the runtime monitor on the creation of an (initialized) parameter. The call to `cirm_AccessArray` in line 7 checks whether the index `i` is within array bounds (i.e., whether the locations `values[0]` and `values[i]` belong to the same allocated array). The call to `cirm_InitVariable` in line 9 notifies the runtime monitor that location `values[i]` has been initialized. `sum`'s instrumentation in line 18 keeps track of variable `res`, which was initialized. Before `res` is read, `cirm_AccessVariable` checks that it was initialized (line 20). Line 21 checks that the array access is within bounds and that the element was initialized (i.e., `cirmRead`). Line 31 allocates heap memory for `N` values. The call in Line 32 notifies the runtime system on the memory allocation. Line 33 notifies the runtime system about the declaration and initialization status of variable `values`. ROSE-CIRM aborts the program and

```
#define N 8
2 void compute(int (* values)[N]) { /* Compute values of array elements */
4   cirm_EnterScope();
    cirm_CreateVariable(&values, sizeof(values), cirmStack, cirmInitialized);
6   for (int i=1; i<N; ++i) { /*scope and variable instrumentation omitted*/
      cirm_AccessArray(&(*values)[0], &(*values)[i], cirmWrite);
8     (*values)[i] = ...;
      cirm_InitVariable(&(*values)[i], sizeof((*values)[i]));
10  }
    cirm_ExitScope();
12 }
14 void sum(int (* values)[N]) { /* Add all array elements' values */
    cirm_EnterScope();
16  cirm_CreateVariable(&values, sizeof(values), cirmStack, cirmInitialized);
    int res = 0;
18  cirm_CreateVariable(&res, sizeof(res), cirmStack, cirmInitialized);
    for (int i=0; i<N; ++i) { /*scope and variable instrumentation omitted*/
20    cirmAccessVariable(&res, sizeof(res));
      cirm_AccessArray(&(*values)[0], &(*values)[i], cirmRead);
22    res = res + values[i];

24    cirm_InitVariable(&res, sizeof(res));
    }
26  printf("sum = %i", res); /* print result */
    cirm_ExitScope();
28 }
30 int main() { /* scope and call instrumentation omitted */
    int (*values)[N] = (int (*)[N]) malloc(sizeof(*values));
32  cirm_AllocMem(values, sizeof(*values), akCHeap);
    cirm_CreateVariable(&values, sizeof(values), cirmInitialized);
34  compute(values);
    sum(values);
36 }
```

Figure 6. Instrumented sequential source code

reports an error message, when the uninitialized element `values[0]` is accessed (line 21).

Using the combined instrumentation and runtime support, ROSE-CIRM for C/C++ is able to detect errors including erroneous deallocations, accesses of unallocated memory, out of bound errors, accesses to uninitialized memory, dangling pointers, unreachable heap memory, mismatched function arguments, etc.

We have used the RTED benchmark [13, 14] to evaluate ROSE-CIRM for C and C++. ROSE-CIRM passes most (96%) test programs of the respective RTED benchmark (excluding floating point errors).

3. Extending ROSE-CIRM for UPC

In this section, we will discuss errors in UPC programs and the challenges that UPC's notion of a shared memory parallelism poses for a ROSE-CIRM extension.

3.1 Errors in UPC Programs

UPC inherits vulnerabilities from C and shared memory parallel programming. We group runtime errors according to their origin:

C style errors occurring in the private memory space of a UPC thread do not involve parallelism. The ROSE-CIRM functionality for C/C++ can handle such problems.

In UPC, similar *(C style) errors involving the global shared address space* can occur. This includes array subscripts where the index is out of bounds (e.g., `p[expr1]` when `p` is be a pointer to shared), reading through a dangling pointer (e.g., while another thread frees the same memory region), and reads of uninitialized values (potentially initialized by other threads) from the shared memory space. The presented extension for ROSE-CIRM addresses these problems.

Similar to the C library, the *UPC library functions* specify preconditions on arguments. Since UPC systems are not obliged to detect precondition violations, programmers must ensure that a specific argument tuple is valid. The current ROSE-CIRM implementation does not address these problems (although similar checks for C library calls could be extended to handle UPC library calls).

```
void compute(shared int (* values)[N]) {
2   cirm_EnterScope();
    cirm_CreateVariable(&values, sizeof(values), cirmStack, cirmInitialized);
4   upc_forall (int i = 1; i < N; ++i, &values[i]) {
      cirm_AccessArray(&(*values)[0], &(*values)[i], cirmWrite);
6     values[i] = ...;
      cirm_InitVariable(&(*values)[i], sizeof((*values)[i]));
8   }
    cirm_ExitScope();
10 }
```

Figure 7. Parallel compute function

Concurrency-induced vulnerabilities refer to problems such as deadlocks, livelocks, and races. Detecting these problems is beyond the scope of this paper.

3.2 State Updates in a Partitioned Global Address Space

The implementation of a monitored execution environment for UPC programs poses several new challenges. First, a runtime monitor's state update of the shared space must be observable by other threads' runtime monitors. For example, a shared variable initialized by one thread must be seen as initialized by other threads. Whether this is implemented by a single shared or multiple distributed data structure is a design choice. Our approach uses a separate runtime monitor for each UPC thread. The runtime monitors exchange messages on state updates.

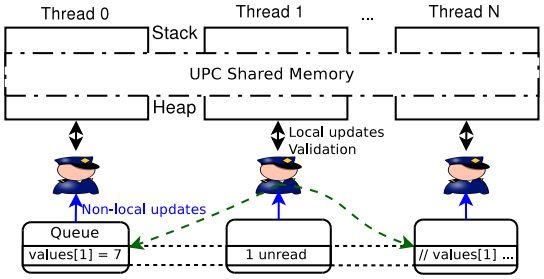

Figure 8. Runtime monitor communication

Fig. 8 shows the design. Each UPC thread has its own runtime monitor. The instrumented code notifies the runtime system on state updates and requests the validation of unsafe operations before they are carried out. Unsafe operations are tested using the information that the runtime system has available. Any update of the global memory space's state is communicated to the other runtime monitors through message queues. Each thread has its own message queue, where other threads can send their state updates.

Consider the `compute/sum` program from §2.3. Fig. 7 shows a parallelized version of function `compute`. Thread 1 executes line 6 (e.g., `values[1]=7`) which writes to the global address space. After the assignment, the instrumentation in line 7 notifies runtime monitor 1 about the state update (`cirm_InitVariable` in the code and black solid arrow in the diagram). Since the memory location was previously uninitialized, the change of state must be shared with the other runtime monitors. This is done by allocating memory for a message in the memory partition of each receiving thread (using `upc_global_alloc`). Then the sender (runtime monitor 1) copies the message (`values[1]=7` and green dotted arrows) to each receiver and enqueues the message into the receiver's queue (enqueue and dequeue use locks). The order of messages can differ in each queue. The sending thread's local portion of the allocated memory is uti-

lized as lock-protected counter keeping track of how many receiving threads have not processed the message (1 unread in Fig. 8).

We have modified ROSE-CIRM's interface functions (Table 1) to process the messages from the message queue whenever the runtime monitor gets called. After a thread has processed (blue solid arrow) the incoming message, it decrements the unread counter. The diagram shows a program state where all but Thread 0 have read the message, thus the counter is 1. The last thread frees the memory associated with the message.

In code prone to race conditions (where one thread writes and the other thread reads a variable), the reading thread might receive the notification too late. This data race scenario is discussed in §3.3.

The presented design assumes that state updates to global address space are rare among all operations. Runtime checks can be carried out without delays, because each runtime monitor has all information available. Having separate runtime monitors in each thread that communicate with each other also gives us control over the time when those messages are processed (i.e., when a PGAS state modifying operation becomes visible to a validation call). By delaying the processing of runtime messages until the next thread synchronization, some potential race conditions can be identified.

The downside of replication is that any global state modifying operation needs to be broadcast to all other UPC threads. Consider image convolution, where the pixel of a new image is calculated from a pixel and its eight neighbors. Assuming each write operation is a state update, each thread in our approach receives $\texttt{pixels} * \frac{\texttt{THREADS}-1}{\texttt{THREADS}}$ update messages (one for each pixel written by other threads) and sends $\texttt{pixels} * \frac{1}{\texttt{THREADS}}$ messages (one for each pixel that the thread writes).

3.3 Runtime Error Detection and Race Conditions

Another challenge for runtime monitoring of parallel programs is dealing with race conditions [16]. By race condition, we mean that the outcome of an operation depends on the runtime order with respect to another operation executed by a different thread.

We distinguish three kinds of race conditions:

- Early release problem: Some thread releases dynamic memory and another thread writes to that location.
- Data race: One thread writes to a memory location and another thread reads from or writes to the same location.
- Atomicity violation: Two operations, such as setting a value and checking it, must be carried out indivisibly.

While ROSE-CIRM does not detect these hazards, the runtime monitor has to prevent race conditions in the original code from erroneously allowing an unsafe operation. To execute such operation combinations safely, we impose a runtime order on them. In an effort to balance synchronization overhead and safety, we allow ROSE-CIRM to err conservatively. ROSE-CIRM never allows unsafe operations, but it may spuriously report an error, if the monitored program contains a race.

Early release problem: Consider the following code being executed by more than one thread:

```
shared int* p = upc_all_alloc(THREADS, sizeof(int));
2 if (!p) return;
4 p[MYTHREAD] = 0;
  /* useful work using p is omitted here*/
6 ...
  if (MYTHREAD == 0) upc_free(p);
```

Line 1 collectively allocates shared memory with a size of `THREADS*sizeof(int)` bytes. Each thread stores the beginning of the allocated memory in a private-to-shared pointer variable `p`. If the allocation was successful, line 4 zero-initializes each integer value for each thread. In line 7, Thread 0 frees this memory. The code is vulnerable to a race. Without synchronization between the memory initialization and the deallocation, Thread 0 might free the memory before all other threads have initialized and used their local values in useful work.

Such early memory releases pose problems for safe execution environments not relying on a garbage collector. Validating the operation in line 4 and executing the operation are not atomic. We consider the use of a reader-writer lock for every operation on dynamic shared memory as prohibitively expensive. Any concurrently executing thread must be prevented from interleaving a shared memory access test with a call to `upc_free`. The runtime monitor must provide some kind of synchronization between writes to dynamically allocated shared memory and memory deallocations. Our approach uses a *safe collaborative work zone*. Fig. 9 depicts the states and state transitions of threads in this approach.

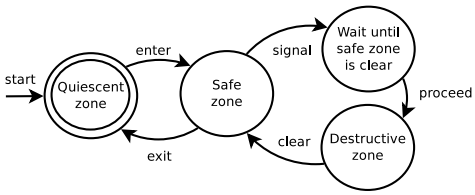

Figure 9. Work zone states and transitions

All threads start in the quiescent zone. In this state, they do not participate actively in any program operation (threads wait, or perform runtime monitor initialization/termination operations). In order to *enter* the safe zone, threads check that no other thread performs (or waits to perform) a destructive update (i.e., calls to `upc_free`). Once a thread enters the safe zone, all non-destructive operations are allowed. Before a thread can carry out a destructive heap operation it *signals* its intent to the other threads and waits until all other threads have *exited* the safe zone. At any point in time there can only be one thread that signals. Then, the thread *proceeds* to carry out the unsafe operation, broadcasts the state update, and *clears* the destructive zone. Threads can *enter* to the safe zone when there is no other thread in or waiting for the destructive zone. Before a thread enters a state where it cannot react to a signal, it also has to *exit* from the safe zone. This includes blocking operations (e.g., `upc_barrier`, `upc_lock`) and thread exits.

Checking whether another thread intends to conduct a destructive operation is performed upon entry of runtime monitor functions. Entering in the destructive zone is implemented by adding two functions (`cirmBeginDestructive`, `cirmEndDestructive`) to the runtime monitor interface. The instrumentation phase wraps destructive operations. The following code snippet shows the instrumented line 7 from the previous example.

```
1 if (MYTHREAD == 0) {
    cirmBeginDestructive();
3   upc_free(p);
    cirmEndDestructive();
5 }
```

In order to avoid deadlocks in our safe work zone scheme, all threads entering a potentially blocking state (e.g., all collective operations, `upc_lock`) have to `exit` the safe zone. We have implemented this approach by adding two runtime monitor functions `cirmEnterWorkzone` and `cirmExitWorkzone`. The instrumentation phases makes sure that threads exit and enter the work zone when executing a blocking operation.

```
1 cirmExitWorkzone();
  upc_lock(some_lock);
3 cirmEnterWorkzone();
```

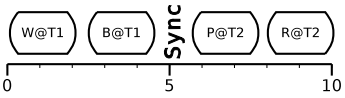

Figure 10. System state update and synchronization

We opted for this implementation (as opposed to using reader-writer locks) because we assume memory deallocations and blocking operations occur rarely compared to shared memory accesses. Consequently, this approach incurs little lock/unlock overhead in loops accessing the global address space.

Data races: Reads are tested before the actual memory access occurs. A read never causes a state update. When a write operations changes the system state (the memory location went from uninitialized to initialized), the state update is communicated after the write operation has finished. ROSE-CIRM processes state update messages after synchronization statements (e.g., `upc_barrier`, `upc_notify`). This suffices to properly check read and write operations as long as the original code is free of race conditions. In Fig. 10, Thread 1 writes a value (W@T1) which triggers a state update (the memory was initialized). The update is broadcast (B@T1) to all other threads before the synchronization statement. If they have not done so earlier, all threads process the state update messages after the synchronization (before they proceed). Thus, any following read operation (R@T2) is carried out after all messages that were sent before the synchronization have been processed (P@T2).

ROSE-CIRM works with both the relaxed and strict memory consistency models. Synchronization mechanisms relying on strict memory consistency imply memory fences before and after the strict operation. Thus checking operates the same way as with built-in synchronization mechanisms. Only if there is a race between W@T1 and R@T2, we permit ROSE-CIRM to spuriously report an uninitialized value. Under the relaxed memory model an implementation must guarantee that a state update message (P@T2) cannot be processed before the corresponding write operation (W@T1) has been observed. This is guaranteed by an (implicit) memory fence when the update message (U@T1) is enqueued in T2's message buffer.

ROSE-CIRM's runtime system validates pointers after they have been assigned new memory locations. For UPC's shared pointers to shared memory, this is problematic as other threads can read the pointer value before it has been validated by the pointer modifying thread. To avoid *atomicity problems*, ROSE-CIRM performs shared pointer modification/validation and dereferences in a critical region.

```
1 // shared int* shared p = ...
3 cirmEnterSharedPointerOp();
  p+=2;
5 cirmExitSharedPointerOp();
```

3.4 Address Handling

To access the global address space, UPC provides the notion of shared variables and pointers. *Pointers to shared memory* encode the thread number, the address in the global address space, and the phase (the offset within the current block). In addition, the local partition of the global address space can also be accessed using regular C pointers. Shared memory addresses can be cast to C-style pointers, but not vice versa.

To deal with local and global addresses uniformly, the ROSE-CIRM runtime system uses an address abstraction with two fields in forms of <thread_id, local_address> or <thread_id, address_offset>. The runtime monitor uses the former to locate the state information and the latter to communicate state updates. The *thread_id* indicates the owning UPC thread. The *local_address* points to a valid address within the address space of the current thread. When the *thread_id* indicates a remote address, the local address field denotes the location in the remote thread projected on the current thread's global address space range. The *address_offset* denotes an address offset within the current thread's global space range.

Consider two UPC threads, Thread 1 and Thread 2. In their own address space, the global shared space starts at address `0x40` (local_global_address_base) and `0x60` respectively. A shared array that starts at local address `0x50` in Thread 1 would start at local address `0x70` in Thread 2. To keep track of the memory state in a uniform way, addresses in the global address space are communicated in terms of offset from the shared memory base address. When Thread 1 receives an update notification (e.g., `cirmInitVariable`) on location <Thread 1, `0x50`>, the communication system sends the location as <Thread 1, `0x10`> (addr_offset = global_addr - local_global_addr_base) to Thread 2. Thread 2 translates the address offset back to its own local address space by adding back its shared memory base. The address becomes <Thread 1, `0x70`>.

For pointer checking, the runtime monitor has to dereference an address. If the address resides in the shared space, our address representation has to be converted into a shared-to-shared pointer in order to utilize UPC's dereference implementation. This conversion depends on the particular shared pointer format in a UPC compiler and runtime system.

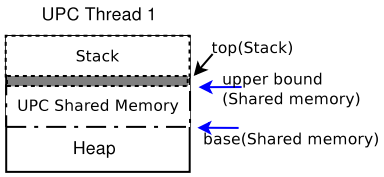

Figure 11. UPC thread memory layout

The fact that C-style pointers can point to locations into the local shared address space makes it impossible to use type information to decide whether an operation modifies the shared heap. As illustrated in Fig. 11, we test instead whether an address falls within the shared heap boundary. This technique also depends on the particular memory layout used by a UPC compiler and runtime system.

3.5 Bounds Checking

For regular C code, bounds checking is trivial. A memory access is valid if a memory location lies within a specific address range. In UPC, the order of two relative addresses depends on the specified blocking factor. We illustrate the problem with an array layout for a two-dimensional array depicted in Fig. 12:

The diagram shows the memory layout for a two-dimensional character array in an UPC program with four threads with bounds four (the number of UPC threads) and eight respectively. Each innermost array component is drawn in a different color. The number in each cell denotes the linearized offset within the defined array. (The four white cells at offset 32 to 35 belong to padding bytes). The array `chararr[0]` ranges from address (actually, the offset in each threads shared memory) `0x50` to address `0x51` in Thread 2. The blocking factor (3) determines how many elements are consecutively allocated to a thread. Thus address `0x52` in Thread 0 is within `chararr[0]` and its relative order is before address `0x51` in Thread 2's. Since, in this example, the blocking factor is not a divisor of the total array size, the allocated space is larger than the array. The first two threads host nine elements, Thread 2 eight elements, and Thread 4 six elements. The unused space (white cells 32-35) are padding bytes.

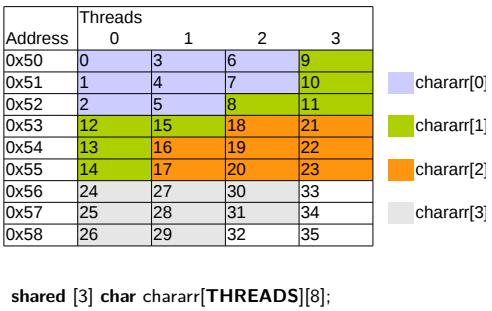

Figure 12. Array memory layout

Other blocking factors produce different mappings from addresses to linearized offsets within an distributed array. For example, with a blocking factor of eight, every thread would own all elements of a first dimension (e.g., Thread 0 hosts `chararr[0]`, and Thread 1 `chararr[1]`, etc.). In this case, address `0x52` in Thread 0 would linearize before address `0x51` in Thread 1. If the blocking factor is one, the characters are distributed in round robin fashion. All elements of the first array dimension (`chararr[0]`) would reside at addresses `0x50` and `0x51` of all threads. Address `0x52` in Thread 0 would linearize after address `0x51` in Thread 1.

The blocking factor is an integral component for array bounds checking. ROSE-CIRM derives the blocking factor at memory allocation time and stores this information together with the allocated block. The runtime monitor uses the blocking factor to derive linearized offsets for an address. Bounds checking tests whether the linearized base offset and the linearized location belong to the same array component. For multi-dimensional arrays, each subscript operation is checked separately. Subsequently there exists a single valid mapping from an array index to a linear address. Note UPC allows pointers to shared's blocking factor differ from the blocking factor specified at allocation time. Such pointers are type checked according to the block size given at allocation time, and not with their actual block size.

3.6 Implementation

We have extended ROSE-CIRM's instrumentation to handle UPC specific constructs, such as `upc_forall`, blocking factors, shared memory allocations/deallocations. We also added the instrumentation to protect the code from race conditions (e.g., `upc_free` is protected by `cirmBeginDestructive()` and `cirmEndDestructive()`). The instrumentation follows the ROSE-CIRM approach to separate the instrumentation from the original code. The shortcoming of this approach is that memory accesses depending on function calls (e.g., `arr[index()]`) cannot be instrumented without side effects.

The runtime monitor has been implemented to work with GCCUPC [8, 12], but the approach can be ported to other UPC systems. We rely on two implementation specific techniques: (1) We use information of GCCUPC shared pointer representation to convert our own address abstraction back into UPC shared pointers. (2) the shared memory base is read from a GCCUPC variable. Since the address of the upper bound is not available, our implementation uses the current top of the stack. Note that UPC does not require the shared heap to be contiguous. Such a case can be handled by including a shared heap descriptor in the address abstraction.

We added six synchronization functions (shown in Table 2) to the runtime monitor interface. S indicates that these functions only synchronize. Many existing runtime monitor interface functions (shown in Table 1) have been extended to support UPC. For example, the following functions have been extended to update the system state and trigger communication among runtime monitors: `cirmAllocMem` for `upc_alloc` and `upc_global_alloc` (note, as a collective operation `upc_all_alloc` does not need communication), `cirmFreeMemory` for `upc_free`, `cirmInitVariable` for writes to uninitialized shared memory, and `cirmMovePointer` for changes to pointers (both shared and local) that point to shared memory.

Function	Action	Description
Synchronization Functions		
cirmExitWorkzone	S	Stops operating in safe mode
cirmEnterWorkzone	S	Starts operating in safe mode
cirmBeginDestructive	S	Waits until other threads have left the safe workzone
cirmEndDestructive	S	Ends the destructive operation
cirmEnterSharedPtrOp	S	Starts a shared pointer to shared operation.
cirmExitSharedPtrOp	S	Ends a shared pointer to shared operation.

Table 2. UPC related functions added to the runtime monitor

Another concern of this project was *code reuse* as the existing ROSE-CIRM consists of about 7500 lines of C++ code (including comments and empty lines), which makes extensive use of data structures (i.e., arrays, maps, and hash tables) of C++ STL and its extensions. However, C++ STL and its extensions do not support the concept of PGAS. The lack of similar data structures that can be placed in the global address space prompted us to pursue an implementation where each thread owns a local (possibly partial) copy of the system state. Note the C++ STL designers provided for tailoring their data structures to specific memory allocation requirements. This is typically accomplished by instantiating data structures with customized allocators. While, in principle, a shared memory allocator could acquire memory from the global address space, this approach remains insufficient due to the inability to make pointers to the global shared space work within ISO C++.

4. Evaluation

This section evaluates our tool for its accuracy and overhead using the RTED benchmark suite [13, 14] and a few error free codes. Table 3 summarizes the system configuration of our evaluation.

Hardware	Intel X5680 (6x2 cores) clocked at 3.3Ghz
Memory	24GByte
OS	Red Hat Enterprise Linux Client 5.6
C++ compiler	g++ (4.1.2-50) Red Hat
UPC compiler	GCCUPC (4.5.1.2)

Table 3. Test system configuration

We compiled the test codes without optimizations and linked them to the compiler optimized ROSE-CIRM runtime library. UPC (like C99 and C++) zero initializes data that has static lifetime. To assess the overhead of the runtime monitor communication, we test with ROSE-CIRM's default setting that treats global data as uninitialized. (To allow informed programmers rely on the standard behavior, ROSE-CIRM offers a switch to change the initialization status of global variables.) Depending on the test, we used between 2 and 12 UPC threads. We report the average run-time of ten runs in seconds.

4.1 Accuracy of Dynamic Analysis

The RTED benchmark suites [13, 14] test whether certain defects of C, C++, Fortran, and UPC programs are detected and properly reported by compilers, runtime systems or third party tools. Each test suite contains programs exhibiting a specific software defect. For each test, the suite defines an example of a good error message.

Category	Number of tests	Correctly Identified (in percent)	
Out of bounds accesses (indices)	726	685	(94%)
Out of bounds accesses (pointers)	160	150	(94%)
Uninitialized memory reads	64	62	(97%)
Dynamic memory handling related	10	10	(100%)

Table 4. RTED UPC benchmark

The RTED suite is organized in several error categories. For example, the RTED C suite tests the following runtime error categories: memory allocation/ deallocation, array index out of bounds, floating point, IO related, memory leaks, pointer, string, function invocation, uninitialized variables, and C99 specific defects. The RTED UPC test suite contains the following additional error categories: *deadlocks*, *races*, out of bounds indices, out of bounds pointer accesses, *out of bounds accesses involving UPC functions*, *bad arguments to UPC functions*, uninitialized memory reads, dynamic memory handling related, and *wrong order of UPC function invocation*. In this paper, we focused on porting the existing ROSE-CIRM detection mechanism to support the global address space. The RTED error categories, printed in italics, are related to parallelism or UPC build-in functions. These categories are not within the scope of this work. The RTED for UPC benchmark suite is designed to work with four UPC threads.

```
1 shared double arr_x[THREADS*4]; /*DECLARE1*/
  shared double *ptr_x;
3 int main() {
5   upc_forall(int i=0;i<THREADS*4;i++;i) arr_x[i] = THREADS + i;
    upc_barrier;
7   if(MYTHREAD == THREADS - 1){
9     if( zero() ) ptr_x = (shared double *) &arr_x[0];
11    double var_res = (*ptr_x) + MYTHREAD*THREADS; /* RTERR */
    }
13 }
```

Figure 13. RTED benchmark c_G_2_c_D.upc

Fig. 13 shows an example from the RTED benchmark suite. In line 11, `ptr_x` has not been set, thus `*ptr_x` dereferences NULL. The program declares a shared array (line 1) and a pointer to a shared `double` (line 2). The `upc_forall` loop in line 5 initializes the array. One of the threads enters the following `if` statement. In line 9, `if`'s condition (call to `zero`) always returns false. Note `zero` is implemented via a function call to prevent it from being easily analyzed. Thus, the error cannot be flagged with certainty at compile time. At run-time, `ptr_x` remains set to NULL and reading its value should be reported.

While testing against the RTED benchmark suite, ROSE-CIRM found a few unplanned bugs in the test programs. For example, the expected source line number mismatches the line number where the error occurs, or some tests unintentionally freed stack memory. Table 4 shows the results obtained for tests we regard as correct before the marked error occurs. To assess our results, we compare the line number of the reported error against the location defined in the benchmark suite. We do not judge the conformance of the produced error message.

ROSE-CIRM detects most errors in the benchmark programs. Programs where ROSE-CIRM does not report the runtime error correctly involve subtle out of bounds accesses in dynamically allocated multi-dimensional arrays (§2.3) and code portions that are currently not instrumented (i.e., array subscripts depending on function calls).

```
1 long res = 0;
  cirmCreateVariable(&res, sizeof(res), cirmInitialized, cirmStack);
3 for (long i = 0; i < 100000000; ++i) {
    res = i + res;
5   cirmInitVariable(&res, sizeof(res));
  }
```

Figure 14. Overhead of instrumentation

4.2 Overhead of Dynamic Analysis

To determine the overhead of runtime monitoring, we conducted tests with a simple (Fig. 14) loop that only updates a single variable in the private space. We have removed other access checks such as cirmAccessVariable. The runtime overhead ratio is a factor of 230. Compiling the test programs with optimizations increases the overhead of the safe execution environment, as the instrumented calls prevent aggressive compiler optimizations.

We also tested ROSE-CIRM against two error free programs, N-Queens code and heat conduction, provided by El-Ghazawi et. al [7]. Both programs have a relatively small core that determines performance. This makes the codes easy to analyze.

The *N-Queens* program [7, pp 67–71] is a parallel implementation that determines the number of solutions to the N-Queens problem that exist for an NxN board. The program can be parallelized without extensive use of the partitioned global address space. The problem space is split into THREADS partitions which are solved independently. The only use of shared memory is to store the number of solutions each thread finds. Thread 0 waits until all threads have finished and sums the results. We tested the program with a 13x13 board. We varied the number of threads from 1 to 12. Table 5 summarizes the results. The timing results are stated in seconds.

Approach / Threads	1	4	8	12
Un-monitored (s)	0.29	0.08	0.04	0.02
ROSE-CIRM orig. (s)	141.0	41.4	22.5	14.4
ROSE-CIRM opt. (s)	10.9	3.2	1.7	1.1
Overhead ratio - orig.	484.4	516.9	561.4	625.3
Overhead ratio - opt.	37.6	40.2	43.4	48.1

Table 5. N-Queens: runtime monitoring overhead

We found the instrumented code runs about 550 times slower. The performance hit is primarily caused by the amount of collected data and performed tests. The major factor impacting performance in this example is recursion: The N-Queens code uses recursion to count the number of solutions. For each function call, ROSE-CIRM keeps track of parameters and local variables. Thus each recursion level adds overhead for automatic storage management.

We used the Oracle studio profiler to pinpoint the runtime overhead when four UPC threads are used. Table 6 lists how much overhead is incurred by each runtime monitor function and how often these functions are called. The number in percent indicates the share of the total runtime overhead. We omit functions that have less than a 5% share.

ROSE/CIRM function	Share	times called
Function Call		
cirmCreateVariable	22.51%	24965801
cirmEnterScope	5.79%	78759963
cirmExitScope	16.62%	78759963
Variable Access		
cirmInitVariable	27.70%	78760699
cirmAccessVariable	21.83%	162674801

Table 6. N-Queens : profile

The entries in the first section are roughly related to N-Queens' recursive function implementation. More than 22.5% of the overhead is related to variable creations. `cirmExitScope` cleans up all variables associated with the scope, thus its share on the overhead is larger than `cirmEnterScope`. Approximately half of the overhead is spent on updating and checking the status of memory (`cirmInitVariable` and `cirmAccessVariable`).

Since the overhead is significant, we assessed by how much simple (non UPC specific) intraprocedural static analysis can improve performance. We manually performed the following optimizations:

- reaching definitions: If the use of a local variable has a reaching definition, we can eliminate calls to `cirmAccessVariable`.
- simple escape analysis: If the address of a variable (or parameter) was not taken, any accesses are local. If there is no `cirmAccessVariable` call on the variable, we can eliminate `cirmInitVariable` and `cirmCreateVariable` calls. Moreover, if a scope does not track any variables we can eliminate the calls to `cirmEnterScope` and `cirmExitScope`.
- array bounds checking: ROSE-CIRM's bounds checks are expensive because arrays are treated like pointers. Each call to `cirmAccessArray` retrieves the information based on the base address. Then the two addresses are linearized §3.4 and checked whether they belong to the same innermost array. If the array bounds are statically known (e.g., the subscript operator operates not on a pointer), we can check all but the last array subscript operation with a simple value comparison. Assume an array declaration `int arr[4][3]` and an array access `arr[x][y]`. Here we can replace the call to `cirmArrayAccess(&arr[0], &arr[x])` with a simple bounds check `0 <= x && x < 4`.

In the N-Queens code, these optimizations eliminate the checking of all local variables. Table 5 shows that, on average, runtime overhead decreases from 547 to 42.

The *heat conduction* program [7, pp 54–62] models heat transfer. Values are stored in a globally defined four dimensional grid.

```
shared [BLOCKSIZE] double grids [2][N][N][N];
```

The first dimension, containing two elements, stores the initial values and the results for each time step. In odd-numbered time steps, grid[1] is computed from grid[0], and vice versa for even-numbered time steps. Each element is calculated by considering its six neighboring elements. In addition, the heat conduction code globally defines a shared array where each processor stores the local transferred heat maximum. The iteration stops when the global maximum falls under a predetermined limit. We tested the code using a block size of one, a grid with 80 elements per dimension, and the code running on eight UPC threads. We varied the number of iterations from 10 to 40. Table 7 shows the obtained results. For each test we timed the grid initialization, the first iteration, the last iteration, and the total time. The time is stated in seconds.

Grid initialization and the first loop iteration is the only code portion that incurs overhead due to runtime monitor synchronization. There is no message exchanged after the first iteration has completed. `grid[0]` has been initialized upfront; `grid[1]` and the array storing the locally transferred maximum have been computed during the first iteration. Every subsequent iteration runs faster (see the last iteration data).

The overall performance penalty of the runtime monitor is a factor of 146 for 10 iterations. The performance penalty decreases as the number of iterations increases because the synchronization overhead can be amortized over more iterations. The lower bound for the program overhead can be derived from the overhead of the last iteration (about a factor of 120 and 50 for the unoptimized and optimized code respectively). The two major performance impacting factors are: (1) synchronization overhead, as each thread replicates the initialization state for the entire grid. This overhead becomes more dominant in the optimized code. (2) bounds checking, as the inner loop of each iteration, contains 32 bounds checks (eight array accesses times four dimensions). The overhead in column labeled Last can be almost exclusively attributed to bounds checking.

	Init	**First**	**Last**	**Total**
10 iterations				
Un-monitored (s)	0.007	0.023	0.026	0.23
ROSE-CIRM orig. (s)	2.57	3.78	2.95	33.48
ROSE-CIRM opt. (s)	1.26	4.09	1.25	19.23
Overhead ratio orig.	367.0	164.2	113.5	146.8
Overhead ratio opt.	179.3	177.7	48.1	84.3
20 iterations				
Un-monitored (s)	0.01	0.020	0.022	0.45
ROSE-CIRM orig. (s)	2.54	3.81	2.96	63.19
ROSE-CIRM opt. (s)	2.46	3.41	1.24	32.24
Overhead ratio orig.	254.3	190.5	134.7	140.4
Overhead ratio opt.	246.3	170.3	56.4	71.7
40 iterations				
Un-monitored (s)	0.006	0.024	0.024	0.89
ROSE-CIRM orig. (s)	2.61	3.82	2.99	123.71
ROSE-CIRM opt. (s)	2.05	3.58	1.23	56.20
Overhead ratio orig.	434.3	159.3	124.8	139.0
Overhead ratio opt.	341.3	149.0	51.4	63.1

Table 7. Heat transfer: runtime monitoring overhead

Table 8 gives an overview of how much of the overhead can be allocated to each function. Functions with an overhead share of less than 5% are not displayed. The overhead to synchronize data with the other runtime monitors is shown at the bottom of the table. Note that sending overhead is incurred across all state updating functions. The receive overhead is incurred across all functions, as each of them processes incoming messages.

ROSE/CIRM function	**Share** 10 iterations	**Share** 20 iterations	**Share** 40 iterations
Variable Access			
cirmAccessArray	75.8%	78.2%	79.4%
cirmAccessVariable	10.0%	10.13%	10.0%
cirmInitVariable	9.6%	7.1%	5.7%
Communication overhead			
send	4.2%	2.4%	$< 1\%$
receive	2.9%	1.5%	$< 1\%$

Table 8. Heat transfer : profile

Again, since the overhead is significant, we manually performed optimizations relating to reaching definitions, simple escape analysis, and array bounds checking. The code benefits mainly by reducing bounds checking overhead. For example, the total overhead ratio is reduced from 140 to 72 for 20 iterations.

5. Related Work

UPC compilers and runtime systems offer support for runtime error detection to a varying degree. In our tests with the Berkeley UPC system, we found that it can report invalid pointers to shared memory locations. The developers of the RTED UPC test suite [14] report that the tested compilers and runtime systems (Berkeley, HP, GCCUPC, to some degree Cray) do little to prevent harmful operations and report the responsible source code location. We are not aware of other dynamic bug finding tools for UPC.

A number of dynamic analysis tools exist that work on source code level. ParaSofts Insure++ [17] is a runtime analysis and memory error detection for multi-threaded C and C++. IBMs Purify [21] is a dynamic software analysis tool to help memory debugging and

memory leak detection. Both tools are proprietary software and performed well on the RTED benchmark suites for C and C++ [15]. Intel Inspector [11] (formerly Intel Thread Checker) is a runtime analysis tool for sequential and multi-threaded Windows and Linux applications. Dmalloc [1] is a open source memory debugger C library for finding memory allocation errors. Programs to be analyzed must be recompiled to use Dmalloc memory functions.

Dynamic analysis can also be operate on binary codes. For example, DynInst [2] is a runtime code-patching library that is useful in developing dynamic programming analysis tools. Valgrind [3] is an open source instrumentation framework for building dynamic analysis tools. Pin [5] is a dynamic instrumentation tool for binary executables. Compared to source level tools, binary tools can see the whole program but they often lack of high level type information, which makes it hard to find all errors.

6. Conclusion and Future Work

In this paper, we have presented ROSE-CIRM for UPC, a dynamic analysis tool that reports C-style runtime errors in UPC code. ROSE-CIRM for UPC uses the combined compiler instrumentation and runtime support to monitor the execution of UPC programs and identify C-style errors within a partitioned global address space. Our evaluation indicates that our tool can spot 95% to 100% of UPC runtime errors for the covered categories of the RTED benchmark suite. Our tests also indicate that the runtime overhead introduced by the monitored execution environment can be prohibitive for larger size software. However, even simple static analysis techniques may dramatically reduce the runtime overhead. ROSE-CIRM's source code is distributed together with ROSE with a BSD style license. It can be downloaded from `rosecompiler.org`.

ROSE-CIRM for UPC can be improved to find more bugs related to concurrency, such as race and dead lock detection. The instrumentation and runtime monitor can be improved to handle a larger variety of UPC programs. In particular, this includes programs relying on UPC's library functions, code that accesses shared memory regions with varying blocksizes (by means of pointer casts), and complex array subscript expressions.

Performance issues in C/UPC/C++ can be addressed by integrating with ROSE's existing static analysis tools (e.g., Compass [19]). Large scale parallel systems will benefit from improvements to the runtime monitor design. The runtime monitor relies on a simple queue implementation, thus concurrent PGAS state modifying operations serialize at the communication infrastructure. Broadcasting and replicating the information in each thread diminishes parallelism. Two alternative approaches may be used to maintain the system state across runtime monitors consistent: query/response style communication mechanism and a single shared data structure.

With a *query/response* approach, each thread would query other threads for information that is not locally available. While such an approach is more complex to implement, it would scale better. On massively parallel systems, programmers minimize remote shared memory accesses by storing memory that is frequently accessed together in the same address space partition (blocking). These optimizations would also benefit the query/response based approach because reducing remote shared memory accesses automatically reduces the number of remote state queries.

Using a *shared data structure* to store the (global) state would dispense with explicit message exchanges. An implementation based on shared maps (i.e., red-black trees) and hash tables requires protection mechanisms to ensure data consistency. Using a coarse-grain lock to control access to the data structure would serialize all update operations (reader- writer locks would allow at least concurrent reads). Integrating fine-grained locks that protect only portions of the shared data structure or relying on non-blocking synchronization [9] is a non-trivial effort.

Acknowledgments

We thank Michael Driscoll, Rajesh Vanka, and the anonymous referees for their comments and suggestions. This work was funded by the department of defense and used elements at the Extreme Scale Systems Center, located at Oak Ridge National Laboratory. This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] Dmalloc. `http://dmalloc.com/`.

[2] Dynist: An Application Program Interface (API) for Runtime Code Generation. `http://www.dyninst.org/`.

[3] Valgrind. `http://valgrind.org/`.

[4] Open Fortran Parser. `http://fortran-parser.sourceforge.net/`

[5] Pin - A Dynamic Binary Instrumentation Tool. `http://www.pintool.org/`.

[6] Edison Design Group. C++ front-end. http://www.edg.com.

[7] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003. ISBN 0471220485.

[8] Gary Funck. GPC/UPC 4.0, "flexible heap" design overview. Technical report, Intrepid Technology Inc., Sep 2006.

[9] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *16th International Conference on Distributed Computing*, DISC '02, pages 265–279, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-00073-9.

[10] Hex-Rays SA. IDA Pro Disassembler and Debugger. `http://www.hex-rays.com/idapro/`.

[11] Intel Corporation. Inspector XE 2011 from Intel. `http://software.intel.com/en-us/articles/intel-inspector-xe/`.

[12] Intrepid Technology Inc. GCC Unified Parallel C (GCC UPC). http://www.gccupc.org/. retrieved on 10th June 2011.

[13] Iowa State University. Runtime error detection test suites. http://rted.public.iastate.edu, 2009.

[14] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Elizabeth Kleiman, and Olga Weiss. Evaluating error detection capabilities of UPC run-time systems. In *Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 7.1 – 7.4, New York, NY, USA, 2009. ACM.

[15] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Mi-Young Park, Elizabeth Kleiman, Olga Weiss, Andrew Wehe, and Melissa Yahya. The importance of run-time error detection. In *Third Parallel Tools Workshop*, 2009.

[16] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1 (1):74–88, March 1992. ISSN 1057-4514.

[17] Parasoft. Automatic C/C++ runtime error detection with Parasoft Insure++. White Paper, 2005.

[18] Program Office Strategic Planning and Economic Analysis Group. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. National Institute of Standards & Technology, May 2002.

[19] Dan Quinlan et al. ROSE Compiler Infrastructure. `http://www.rosecompiler.org/`.

[20] Daniel Quinlan and Thomas Panas. Run time error detection (RTED). Project Report. Lawrence Livermore National Laboratory, 2010.

[21] Rational Software. Releasing better software faster with Rational Purify and Rational Quantify. Rational Software White Paper, 2000.

[22] The UPC Consortium. UPC language specification v1.2, June 2005.