Final Report

Reporting Period 9/15/09-4/14/2013

Correctness Tools for Petascale Computing[*]

Rice University Subproject

Cooperative Agreement No. DE-SC0001776

John Mellor-Crummey

Principal Investigator at Rice University

Department of Computer Science, MS 132
Rice University
P.O. Box 1892
Houston, TX 77251-1892
Voice: 713-348-5179
FAX: 713-348-5930
Email: johnmc@rice.edu

[*]Multi-institutional center Project Director: Jeffrey Hollingsworth, University of Maryland, hollings@cs.umd.edu

Contents

1 Introduction

Grand challenge parallel science codes are enormously complex. Mapping them to petascale platforms is hard because of the multi-level parallelism needed to make efficient use of multicore nodes. In the course of code development, subtle programming errors often arise that are extremely difficult to diagnose without tools.

To meet this challenge, University of Maryland, the University of Wisconsin—Madison, and Rice University worked to develop lightweight tools to help code developers pinpoint a variety of program correctness errors that plague parallel scientific codes. The aim of this project was to develop software tools that help diagnose program errors including memory leaks, memory access errors, round-off errors, and data races.

The tools research involved using binary rewriting and function interface wrapping to augment applications to monitor memory allocation/deallocation, memory accesses, and floating point operations. Monitoring code used call stack unwinding of mixed-language programs to associate phenomena (data races, bytes allocated/deallocated, words read/written, floating-point roundoff error) with the calling contexts.

2 Summary of Research Performed

Research at Rice University focused on developing algorithms and data structures to support efficient monitoring of multithreaded programs for memory access errors and data races. In the following sections, we describe the work on each of these topics.

2.1 Memory Leak Detector

As part of this project, we implemented a Memory Leak Detector as an extension to HPCToolkit [8, 2]. The Memory Leak Detector counts the number of bytes for every call to malloc and free and reports the full call path and number of bytes for every malloc. By computing the number of bytes malloc'ed minus the number of bytes freed, we can identify the amount of memory that is leaked. The Memory Leak Detector is fully integrated with HPCToolkit and `hpcviewer` and the counts for malloc and free are displayed with the full call path from main to where the malloc occurred.

The Memory Leak Detector works by overriding the seven malloc functions (malloc, calloc, realloc, free, memalign, posix_memalign and valloc). For dynamically linked binaries, we use the `LD_PRELOAD` feature to override these functions. And for statically linked binaries, we use the loader's `--wrap` feature to define wrapped versions of the malloc functions. In both cases, we replace the original library call with a function that unwinds and records the current call path adds a small number of bytes for header information about the call and then calls the real malloc function. By storing a header with the malloc'ed data, when the program frees the same data, we are able to match the free to the original malloc. This allows reporting the free at the same context where the malloc occurred and thus allows us to identify which mallocs were never freed.

At the outset of this project, the HPCToolkit performance tools were able to unwind call stacks in response to asynchronous interrupts. We enhanced the unwinding capability in HPCToolkit to also support unwinding from within instrumented functions that synchronously trigger an unwind with a procedure call. A strength of HPCToolkit unwinder has been its ability to unwind call stacks for fully optimized code. This capability carries over from asynchronous unwinding to synchronous unwinding.

We used the new synchronous unwinding capability to write instrumented versions of the seven malloc functions: `malloc`, `calloc`, `realloc`, `free`, `memalign`, `posix_memalign` and `valloc`. Each of these functions (a) triggers a synchronous unwind to capture the full calling context of an allocation, (b) increments a count of the total number of bytes allocated in its allocation context, and (c) prepends a header to each allocated block that indicates the number of bytes allocated and the full calling context in which the allocation occurred. This makes it possible to identify exactly which malloc calls were never freed.

We implemented a a version of `free` that uses information recorded in a memory block's header to determine the number of bytes that are being freed and the context in which the bytes were previously allocated. When a block is freed, the `free` operation then attributes the count of bytes freed to the context *in which they were allocated* rather than the calling context of the free itself.

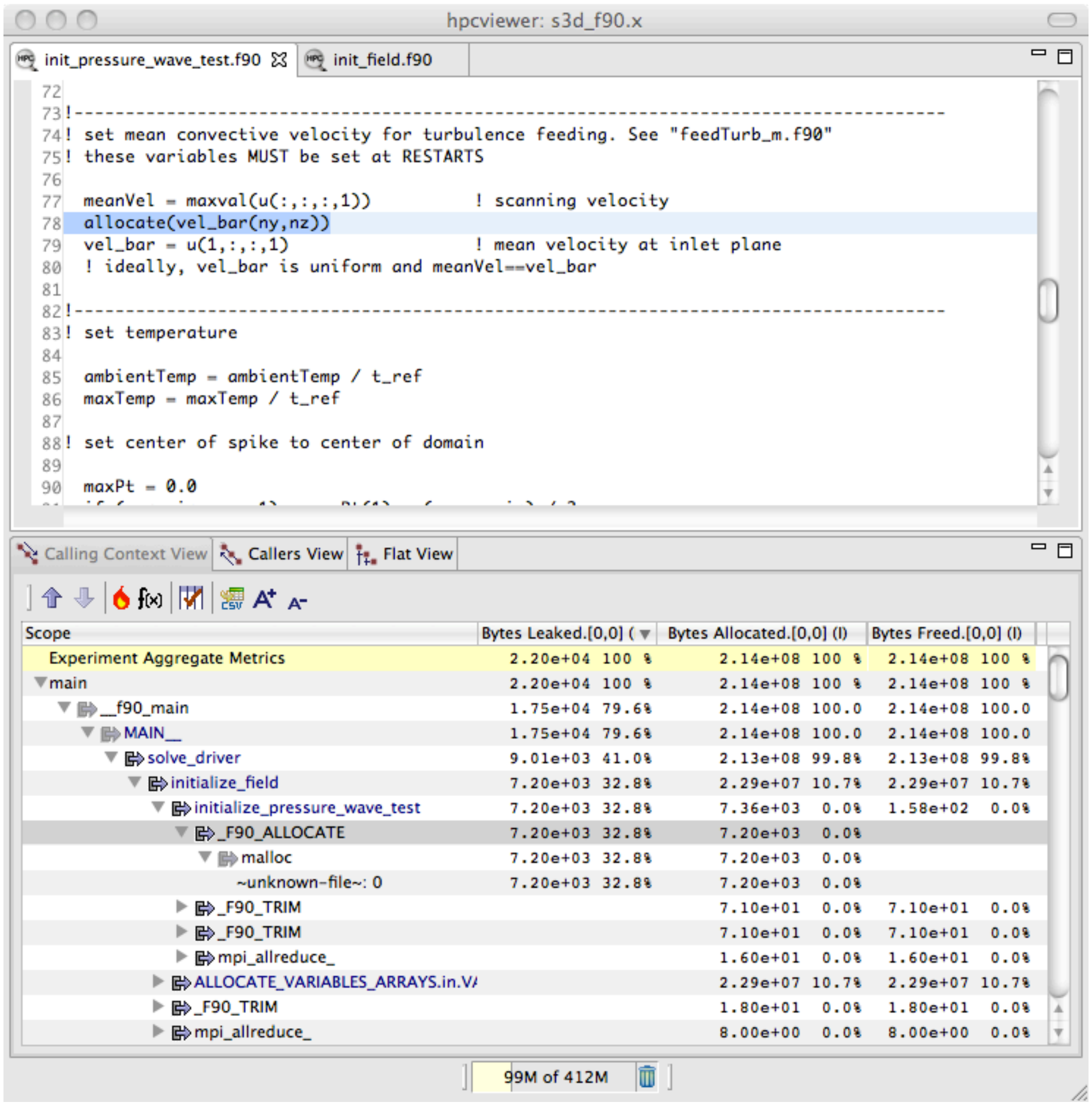
Besides the technical implementation problems, we also encountered a few problems that are not yet fully resolved. One problem is the Linux Out-of-Memory Killer (OOM). By default, Linux overcommits the amount of available memory. That is, requests to add more regions to a process's virtual address space will normally succeed, even if there is not sufficient backing store for the extra memory. Normally, HPCToolkit works by running a program, storing the collected data in memory and then writing out the data at the end of the program. However, if physical memory runs out, then programs can fail catastrophically with no opportunity to catch the failure and write out its data. This is especially a problem for a memory leak tool because programs that leak memory also tend to run out of memory. To cope with sudden death at the heads of the OOM killer, it would be possible for a tool to stream information about all allocates and frees to stable storage (e.g., disk); however, the cost of doing so would be prohibitive. The right solution is for the kernel to handle OOM situations differently, e.g., by signaling an out-of-memory application with a SIGSEGV and giving an embedded leak detection tool a chance to write out its state.

Another problem we encountered is that some programs bypass the normal malloc functions (malloc, calloc, realloc and free). Some programs use alternative malloc libraries, some programs write their own functions for malloc and free, and some programs use libraries that use non-standard, internal aliases for the libc malloc functions (eg, `__libc_malloc`). This makes it more difficult to track the memory behavior of these programs. At present, we have not extended the leak detector to track these other functions.

We used HPCToolkit's Memory Leak Detector to analyze the behavior of several programs and present a summary of the results for three applications here. Figure 1 shows a screenshot of `hpcviewer`'s interface showing bytes allocated, bytes freed, and bytes leaked attributed to each calling context in one MPI process from an 8-core MPI run of Sandia's S3D code on an Opteron-based Linux system. These results were collected on a conventional Linux system at Rice University for an execution of a dynamically-linked version of S3D.

Figure 2 shows a screenshot of `hpcviewer's interface` displaying leak detection results for one of 256 processes in an execution of a statically-linked version of the University of Chicago's FLASH code on Jaguar, a DOE leadership-Cray XT system running Compute Node Linux.

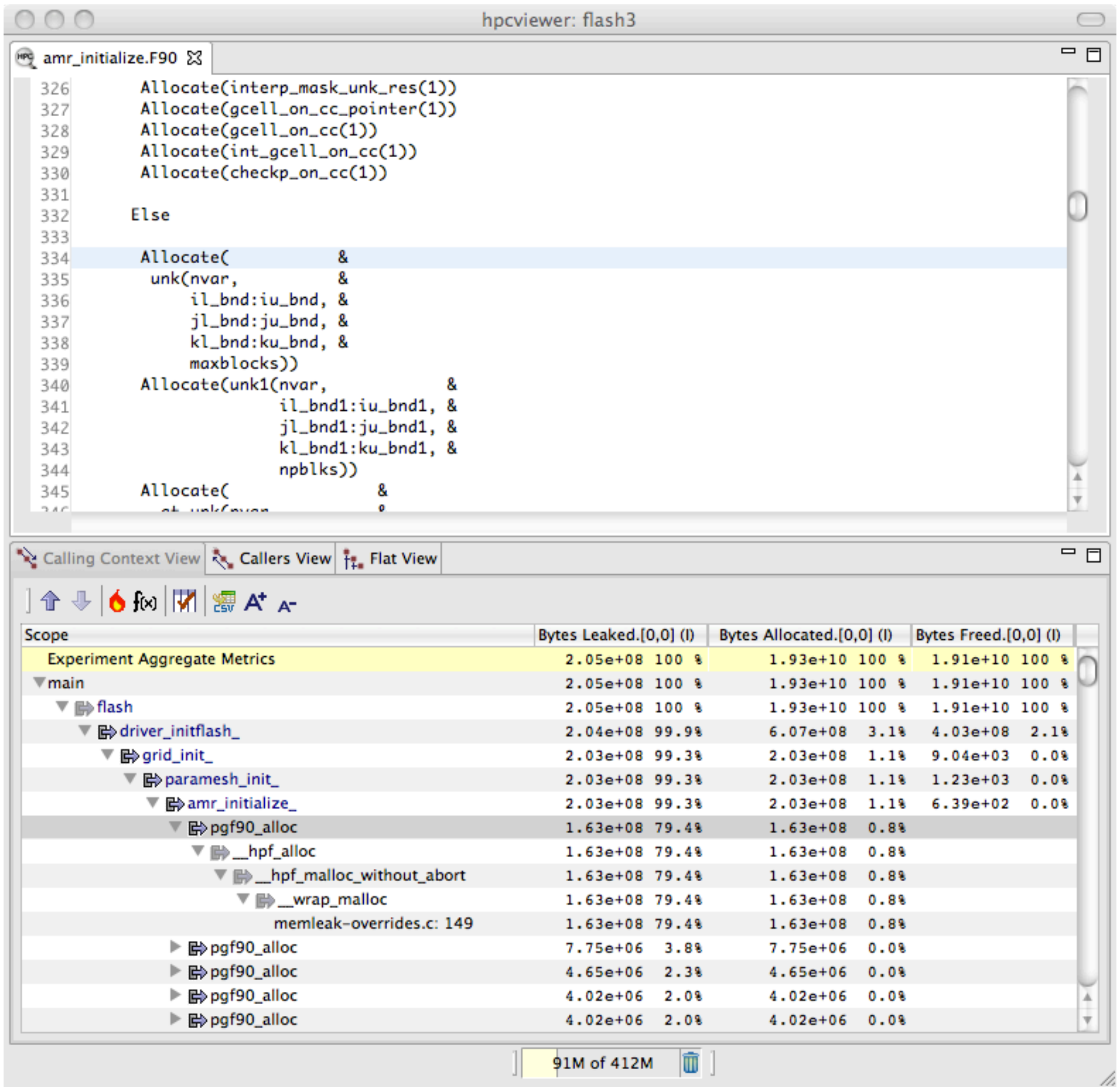
During the summer of 2011, we collaborated with Kenny Roche on the "DOE SC ASCR Software Metric" project. As part of this effort, we applied HPCtoolkit's performance tools, including the memory leak detector, to the codes of this project. We used the leak detector



```
!--------------------------------------------------------------------------------
! set mean convective velocity for turbulence feeding. See "feedTurb_m.f90"
! these variables MUST be set at RESTARTS

  meanVel = maxval(u(:,:,:,1))          ! scanning velocity
  allocate(vel_bar(ny,nz))
  vel_bar = u(1,:,:,1)                  ! mean velocity at inlet plane
  ! ideally, vel_bar is uniform and meanVel==vel_bar

!--------------------------------------------------------------------------------
! set temperature

  ambientTemp = ambientTemp / t_ref
  maxTemp = maxTemp / t_ref

! set center of spike to center of domain

  maxPt = 0.0
```

Scope	Bytes Leaked.[0,0] (		Bytes Allocated.[0,0] (I)		Bytes Freed.[0,0] (I)	
Experiment Aggregate Metrics	2.20e+04	100 %	2.14e+08	100 %	2.14e+08	100 %
main	2.20e+04	100 %	2.14e+08	100 %	2.14e+08	100 %
_f90_main	1.75e+04	79.6%	2.14e+08	100.0	2.14e+08	100.0
MAIN__	1.75e+04	79.6%	2.14e+08	100.0	2.14e+08	100.0
solve_driver	9.01e+03	41.0%	2.13e+08	99.8%	2.13e+08	99.8%
initialize_field	7.20e+03	32.8%	2.29e+07	10.7%	2.29e+07	10.7%
initialize_pressure_wave_test	7.20e+03	32.8%	7.36e+03	0.0%	1.58e+02	0.0%
_F90_ALLOCATE	7.20e+03	32.8%	7.20e+03	0.0%		
malloc	7.20e+03	32.8%	7.20e+03	0.0%		
~unknown-file~: 0	7.20e+03	32.8%	7.20e+03	0.0%		
_F90_TRIM			7.10e+01	0.0%	7.10e+01	0.0%
_F90_TRIM			7.10e+01	0.0%	7.10e+01	0.0%
mpi_allreduce_			1.60e+01	0.0%	1.60e+01	0.0%
ALLOCATE_VARIABLES_ARRAYS.in.V/			2.29e+07	10.7%	2.29e+07	10.7%
_F90_TRIM			1.80e+01	0.0%	1.80e+01	0.0%
mpi_allreduce_			8.00e+00	0.0%	8.00e+00	0.0%

Figure 1: `hpcviewer` presenting memory leak detector data for the S3D application. Leak data gathered for a dynamically-linked executable running on a conventional Linux system. Data shown represents leaks identified in 1 of 8 MPI processes.



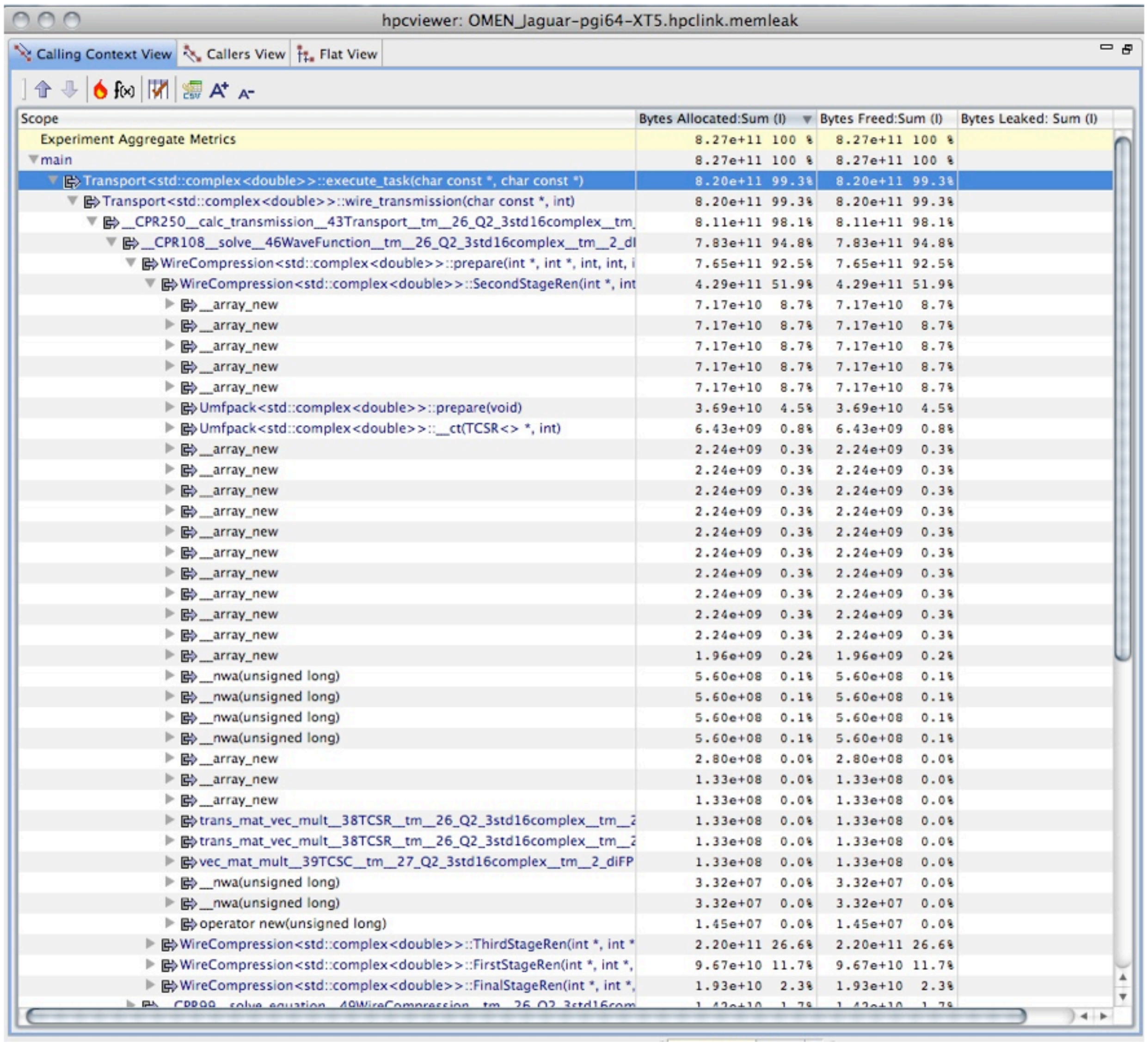
hpcviewer: flash3

amr_initialize.F90

```
        Allocate(interp_mask_unk_res(1))
        Allocate(gcell_on_cc_pointer(1))
        Allocate(gcell_on_cc(1))
        Allocate(int_gcell_on_cc(1))
        Allocate(checkp_on_cc(1))

       Else

        Allocate(           &
         unk(nvar,          &
             il_bnd:iu_bnd, &
             jl_bnd:ju_bnd, &
             kl_bnd:ku_bnd, &
             maxblocks))
        Allocate(unk1(nvar,              &
                      il_bnd1:iu_bnd1, &
                      jl_bnd1:ju_bnd1, &
                      kl_bnd1:ku_bnd1, &
                      npblks))
        Allocate(               &
```

Scope	Bytes Leaked.[0,0] (I)	Bytes Allocated.[0,0] (I)	Bytes Freed.[0,0] (I)
Experiment Aggregate Metrics	2.05e+08 100 %	1.93e+10 100 %	1.91e+10 100 %
main	2.05e+08 100 %	1.93e+10 100 %	1.91e+10 100 %
flash	2.05e+08 100 %	1.93e+10 100 %	1.91e+10 100 %
driver_initflash_	2.04e+08 99.9%	6.07e+08 3.1%	4.03e+08 2.1%
grid_init_	2.03e+08 99.3%	2.03e+08 1.1%	9.04e+03 0.0%
paramesh_init_	2.03e+08 99.3%	2.03e+08 1.1%	1.23e+03 0.0%
amr_initialize_	2.03e+08 99.3%	2.03e+08 1.1%	6.39e+02 0.0%
pgf90_alloc	1.63e+08 79.4%	1.63e+08 0.8%	
__hpf_alloc	1.63e+08 79.4%	1.63e+08 0.8%	
__hpf_malloc_without_abort	1.63e+08 79.4%	1.63e+08 0.8%	
__wrap_malloc	1.63e+08 79.4%	1.63e+08 0.8%	
memleak-overrides.c: 149	1.63e+08 79.4%	1.63e+08 0.8%	
pgf90_alloc	7.75e+06 3.8%	7.75e+06 0.0%	
pgf90_alloc	4.65e+06 2.3%	4.65e+06 0.0%	
pgf90_alloc	4.02e+06 2.0%	4.02e+06 0.0%	
pgf90_alloc	4.02e+06 2.0%	4.02e+06 0.0%	

Figure 2: `hpcviewer` presenting memory leak detector data for the FLASH application. Leak data gathered for a statically-linked executable running on 256 processors of ORNL's Cray XT (Jaguar). Data shown represents leaks identified in 1 of 256 MPI processes.



hpcviewer: OMEN_Jaguar-pgi64-XT5.hpclink.memleak

Scope	Bytes Allocated:Sum (I)	Bytes Freed:Sum (I)	Bytes Leaked: Sum (I)
Experiment Aggregate Metrics	8.27e+11 100 %	8.27e+11 100 %	
main	8.27e+11 100 %	8.27e+11 100 %	
Transport<std::complex<double>>::execute_task(char const *, char const *)	8.20e+11 99.3%	8.20e+11 99.3%	
Transport<std::complex<double>>::wire_transmission(char const *, int)	8.20e+11 99.3%	8.20e+11 99.3%	
__CPR250__calc_transmission__43Transport__tm__26_Q2_3std16complex__tm_	8.11e+11 98.1%	8.11e+11 98.1%	
__CPR108__solve__46WaveFunction__tm__26_Q2_3std16complex__tm__2_dl	7.83e+11 94.8%	7.83e+11 94.8%	
WireCompression<std::complex<double>>::prepare(int *, int *, int, int, i	7.65e+11 92.5%	7.65e+11 92.5%	
WireCompression<std::complex<double>>::SecondStageRen(int *, int	4.29e+11 51.9%	4.29e+11 51.9%	
__array_new	7.17e+10 8.7%	7.17e+10 8.7%	
__array_new	7.17e+10 8.7%	7.17e+10 8.7%	
__array_new	7.17e+10 8.7%	7.17e+10 8.7%	
__array_new	7.17e+10 8.7%	7.17e+10 8.7%	
__array_new	7.17e+10 8.7%	7.17e+10 8.7%	
Umfpack<std::complex<double>>::prepare(void)	3.69e+10 4.5%	3.69e+10 4.5%	
Umfpack<std::complex<double>>::__ct(TCSR<> *, int)	6.43e+09 0.8%	6.43e+09 0.8%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	2.24e+09 0.3%	2.24e+09 0.3%	
__array_new	1.96e+09 0.2%	1.96e+09 0.2%	
__nwa(unsigned long)	5.60e+08 0.1%	5.60e+08 0.1%	
__nwa(unsigned long)	5.60e+08 0.1%	5.60e+08 0.1%	
__nwa(unsigned long)	5.60e+08 0.1%	5.60e+08 0.1%	
__nwa(unsigned long)	5.60e+08 0.1%	5.60e+08 0.1%	
__array_new	2.80e+08 0.0%	2.80e+08 0.0%	
__array_new	1.33e+08 0.0%	1.33e+08 0.0%	
__array_new	1.33e+08 0.0%	1.33e+08 0.0%	
trans_mat_vec_mult__38TCSR__tm__26_Q2_3std16complex__tm__2	1.33e+08 0.0%	1.33e+08 0.0%	
trans_mat_vec_mult__38TCSR__tm__26_Q2_3std16complex__tm__2	1.33e+08 0.0%	1.33e+08 0.0%	
vec_mat_mult__39TCSC__tm__27_Q2_3std16complex__tm__2_diFP	1.33e+08 0.0%	1.33e+08 0.0%	
__nwa(unsigned long)	3.32e+07 0.0%	3.32e+07 0.0%	
__nwa(unsigned long)	3.32e+07 0.0%	3.32e+07 0.0%	
operator new(unsigned long)	1.45e+07 0.0%	1.45e+07 0.0%	
WireCompression<std::complex<double>>::ThirdStageRen(int *, int *	2.20e+11 26.6%	2.20e+11 26.6%	
WireCompression<std::complex<double>>::FirstStageRen(int *, int *,	9.67e+10 11.7%	9.67e+10 11.7%	
WireCompression<std::complex<double>>::FinalStageRen(int *, int *,	1.93e+10 2.3%	1.93e+10 2.3%	
__CPR99__solve_equation__49WireCompression__tm__26_Q2_3std16com	1.42e+10 1.7%	1.42e+10 1.7%	

Figure 3: HPCToolkit's `hpcviewer` showing no leaks in a 9-core run of OMEN.

to verify that OMEN does not leak memory. Figure 3 shows a screenshot of a small 9-core run of OMEN used to verify the absence of leaks.

The Memory Leak Detector is part of the main trunk of the HPCToolkit project hosted at Google Code at `http://code.google.com/p/hpctoolkit/` and is available via anonymous subversion checkout.

2.2 Data Race Detector

Data races may arise between computation threads or through asynchronous communication. For scientific parallel programs, which typically use domain decomposition for parallelization, the key challenge is efficiently monitoring happens-before relationships between synchronizing threads and analyzing apparent races to weed out harmless conflicting accesses, such as when atomic operations provide sufficient atomicity guarantees.

In this project, we implemented a prototype of an on-the-fly data race detection tool for OpenMP programs with ordered sections and locks. There were two parts to this work: developing an on-the-fly protocol for reasoning about data races in an execution of an OpenMP program, and developing a tool that instruments a program to monitor accesses for data races. We describe these two aspects of our race detector in the sections below.

2.2.1 A Protocol for Detecting Races in OpenMP Programs

Our protocol for detecting data races in OpenMP programs is an extension of the *offset-span* labeling algorithm for detecting data races in nested fork-join programs by Mellor-Crummey [6] to handle ordered sections. In fork-join programs, a single thread may split (fork) into multiple threads which then run in parallel and later rejoin to a single thread. There is no synchronization between threads, so two references in different threads to the same memory location, one of which is a write, will always be a data race. Although some data races can be benign (eg, a permutation of the node numbers in a graph algorithm), most data races represent unexpected behavior and thus are a sign of an incorrect program.

The fork-join model focuses on *logical* parallelism instead of *physical* parallelism. That is, all branches of a fork are run concurrently in different threads, regardless of how many threads there are. For example, an OpenMP program might use an `omp parallel` region to compute the values of 100 million elements in an array, but the program could be run on a machine with 16 physical (hardware) threads. In this case, the program has 100 million logical threads, but only 16 physical threads. In the fork-join model, two writes to the same location in different logical threads is always a data race and this is what the offset-span labeling algorithm [6] computes. Any logical data race can become a physical data race if the two logical threads happen to run on different hardware threads. By contrast, a tool like Helgrind [7] reports physical data races and could miss a logical data race if the two racing threads happen to run on the same physical thread.

Although OpenMP mostly follows the fork-join model, it has some significant extensions that go beyond the fork-join model. One of the most important extensions is that of ordered critical sections. An ordered section within an OpenMP parallel loop requires that the section in thread 1 must finish before the section in thread 2 begins which must finish before the section in thread 3 begins, and so on. With ordered sections, it is possible to have two writes in different threads that are never a data race because one must always occur before the other from the rules of ordered sections. This cannot happen within a pure fork-join program, and the original fork-join algorithm would report many false positives in this case.

Our contribution is to extend the offset-span labeling algorithm to the case of ordered sections in OpenMP programs. We extended offset-span labeling to add a *phase* number that is incremented on entry and exit to an ordered section. The offset-span-phase triple for a memory reference is sufficient to determine the "happens before" relation in ordered sections in the same way that offset-span is used in pure fork-join programs. We then proved that an arbitrary set of reads from a specific location can be condensed to two representative reads (technically, the right most and most deep reads) such that if there is a race with the full set, then there must be a race with one of the two representatives. This gives a practical on-the-fly algorithm for detecting data races by maintaining three values (two reads and one write) for each memory location. Both this and the original fork-join algorithm are *exact* in the sense that they have no false positives and no false negatives for the models that they represent.

2.2.2 A Tool for Detecting Data Races in OpenMP Programs

Using the protocol described in the previous section, we constructed a prototype Data Race Detector for the x86-64 architecture using Intel's Pin tool [1, 5]— a leading tool for dynamic instrumentation of a program binary on x86 and x86-64 platforms. Pin supports instrumenting a running program by replacing arbitrary machine instructions with a call to a wrapped (instrumented) version of that instruction; this enables very fine-grained monitoring of the program's execution. Pin is the basis for many tools for both correctness and performance.

For pinpointing races, a key feature missing from Pin is the ability to associate call paths with instructions as they execute. That is, instead of reporting what happened with a machine instruction at hexadecimal address 0x402fa8, we prefer to report this as line number 142 inside the `for` loop on the path `main` calls `foo` calls `bar` calls `baz`. Specifically, we want to know not only where in the source code some action took place, but we also want to know how the program got there. Programs are built with subroutines calling other subroutines, and it is far more useful to know how some function was called instead of just the single location in the code.

To enable data races detected with Pin to be attributed to call paths, we developed CCTLib [3]—a library for recording call paths in Pin programs. CCTLib collects accurate call paths through dynamically loaded libraries, stripped libraries and handles binary code for which the compiler provided incorrect or incomplete information about function bounds. CCTLib provides an API for recording a call path at an arbitrary point in a program execution and can be used by any Pin tool. CCTLib works by maintaining a shadow stack of the program's call stack. A shadow stack involves instrumenting function entry and exit points, either at compile time or by using binary rewriting. The advantage of a shadow stack is that a full call path is known at every point in the program's execution. The main disadvantage is that it adds overhead to every function call.

Our prototype data race detection tool for OpenMP works with the GNU OpenMP (GOMP) runtime, but it is straightforward to extend it to another OpenMP runtime. With Pin, we intercept each load and store instruction as well as calls to key OpenMP runtime APIs. Our approach works at the binary level and needs no modifications to the application source code.

To maintain access history, we employ shadow memory techniques described in [4]. Providing source-level mapping along with the call paths leading to the two conflicting accesses

```
#define N 2

char  a[N];
int main() {
    int i = 0;
#pragma omp parallel for schedule(dynamic,1) ordered
    for (i = 0; i < N; i++) {
#pragma omp ordered
{
        a[0] = i;
}
#pragma omp ordered
{
        a[0] = i;
}
    }
    return 0;
}
```

Listing 1: OpenMP Ordered Section with data race.

```
W->W race
----------------------------------------------------------------------------
LABEL: [0,1,0][0,2,3]
movb  %al, 0x200471(%rip):main._omp_fn.0:Ordered_Race.c:14
 callq  0x4006e0:main._omp_fn.0:Ordered_Race.c:10
  callq  0x4006f0:main._omp_fn.0:Ordered_Race.c:10
   callq  0x4006e0:main._omp_fn.0:Ordered_Race.c:6
    callq  0x7f0ffe7686e0:GOMP_loop_ordered_dynamic_start:libgomp/loop.c:105
     callq  0x4006a0:main._omp_fn.0:Ordered_Race.c:6
      callq  0x400855:main:Ordered_Race.c:6
       callq  0x18(%rsp):__libc_start_main::0
        callq  0x4006d0:_start::0
        THREAD[0]
******************************** RACES WITH *********************************
LABEL: [0,1,0][1,2,1]
movb  %al, 0x200484(%rip):main._omp_fn.0:Ordered_Race.c:10
 callq  0x4006e0:main._omp_fn.0:Ordered_Race.c:6
  callq  0x4006a0:main._omp_fn.0:Ordered_Race.c:6
   callq  %rbp:gomp_thread_start:libgomp/team.c:124
    callq  %fs:0x408:start_thread::0
     callq  %rax:__clone::0
     THREAD[1]
      popq  %rbx:pthread_create::0
       callq  0x7f0ffe7664d0:gomp_team_start:ibgomp/team.c:460
        callq  0x400690:main:Ordered_Race.c:6
         callq  0x18(%rsp):__libc_start_main::0
          callq  0x4006d0:_start::0
          THREAD[0]
----------------------------------------------------------------------------
```

Figure 4: Output of the Data Race Detector showing the stack traces for two conflicting accesses in the above program.

requires maintaining the call paths of all previous accesses; we accomplish this efficiently both in space and time by constructing a calling context tree as described in [3].

Listing 1 is a sample OpenMP code that uses an OpenMP `parallel for` loop construct in conjunction with OpenMP ordered sections. The code exhibits a data race between two accesses to the array element a[0] on line numbers 10 and 14 and our tool identifies the conflicting accesses as a write-write data race along with the source mapping and call paths leading to the data race as shown in Figure 4.

The Data Race Detector is currently hosted on a subversion repository at Rice University and is available on request.

3 Findings and Results

In brief, significant findings and results of the project were the following:

- Implemented a Memory Leak Detector that reports the number of bytes a program mallocs and where they are malloc'ed but never freed. The tool is fully integrated with HPCToolkit and `hpcviewer` and reports the full call paths where the mallocs occured.
- The Memory Leak Detector can sometimes have a high overhead (as high as 10-20x) for programs that continually malloc and free many small regions. This overhead is due to overriding the library malloc functions (malloc, calloc, realloc, free, etc.). To address this problem, we also implemented a probabilistic mode for leak detection that finds frequent leaks with reasonable probability but with substantially less overhead.
- Some programs bypass the normal malloc functions (malloc, calloc, realloc and free) and either use a different separate malloc library or write their own custom library or use internal libc functions that bypass the normal interface. This makes tracking these programs much more difficult.
- The Linux Out-of-Memory Killer (OOM) remains a problem for the Memory Leak Detector and for tools in general. By default, Linux overcommits the amount of available memory. If physical memory runs out, then programs can fail catastrophically with no opportunity to catch the failure and write out its data. This is especially a problem for a memory leak tool because programs that leak memory also tend to run out of memory.
- We extended the Mellor-Crummey's offset-span labeling strategy for detecting data races in fork-join programs to handle ordered critical sections. We proved that a set of only three access per location (two reads and one write) are sufficient to verify data races for an entire sequence of reads and writes. This model is exact for logical data races in the sense that it has no false positives and no false negatives.
- We implemented a prototype Data Race Detector for OpenMP programs with ordered sections based on the extended fork-join model and using the Intel Pin tool.
- We showed how to extend the ordered fork-join model to the case of lock sets, but we did not implement the extension for locks.

- We identified that the model breaks down for programs that use and depend on physical (hardware) threads to avoid data races.

References

[1] *Pin 2.10 user guide*, `http://www.pintool.org/docs/41150/Pin/html/`.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, *HPCToolkit: Tools for performance analysis of optimized parallel programs*, Concurrency and Computation: Practice and Experience **22** (2010), no. 6, 685–701.

[3] Milind Chabbi, Xu Liu, and John Mellor-Crummey, *Call paths for pin tools*, Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (New York, NY, USA), CGO '14, ACM, 2014, pp. 76:76–76:86.

[4] Milind Chabbi and John Mellor-Crummey, *Deadspy: A tool to pinpoint program inefficiencies*, Proceedings of the Tenth International Symposium on Code Generation and Optimization (New York, NY, USA), CGO '12, ACM, 2012, pp. 124–134.

[5] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, *Pin: Building customized program analysis tools with dynamic instrumentation*, Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '05, ACM, 2005, pp. 190–200.

[6] John Mellor-Crummey, *On-the-fly detection of data races for programs with nested fork-join parallelism*, Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (New York, NY, USA), Supercomputing '91, ACM, 1991, pp. 24–33.

[7] Nicholas Nethercote and Julian Seward, *Valgrind: A framework for heavyweight dynamic binary instrumentation*, Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '07, ACM, 2007, pp. 89–100.

[8] Rice University, *HPCToolkit performance tools*, `http://hpctoolkit.org/`.