HANS BOEHM, BILL PUGH, AND DOUG LEA

standards: multithreading in C and C++

Hans Boehm is a researcher at HP Labs who is best known for his work on garbage collection. Recently he has focused on improving concurrent programming foundations, especially for C++.

hans.boehm@hp.com

William Pugh is a professor at the University of Maryland, College Park. His current research focus is on developing tools to improve software productivity, reliability, and education.

pugh@cs.umd.edu

Doug Lea, a professor of computer science at SUNY Oswego, is the author of several widely used software packages and components, as well as articles, reports, and standardization efforts dealing with object-oriented software development.

dl@cs.oswego.edu

MAINSTREAM DESKTOP and server machines increasingly require explicitly concurrent programs to achieve full performance, owing to the increasing prevalence of both single-chip multiprocessors and hardware support for multiple threads.

Currently a common way to write such programs is to program in C or C++, with the aid of a threads library, such as an implementation of the POSIX pthreads interfaces, to provide concurrency. This is also an established technique for handling multiple concurrent event streams, even on single-threaded single-processor machines.

Unfortunately, this approach has turned out not to be completely sound, primarily because reliable multithreaded execution requires certain guarantees about the language and compiler that cannot easily be provided by a library or library specification [1]. Some of the associated issues have been understood for many years. The second half of this paper briefly outlines a symptom of this issue that appears to not have been well recognized.

As a result, several of us have started an effort to address these problems by directly defining the meaning of multithreaded programs in the underlying programming language. Initially this is being done in the context of C++, building on some earlier work in the context of Java [2, 3].

There appears to be consensus that these issues should be addressed in the current ongoing revision of the C++ standard. In that context, we are addressing three somewhat separable issues:

1. Defining the meaning of existing programs in the presence of threads. Our current approach largely follows pthreads and leaves the semantics undefined if there is a data race, i.e., if a program modifies a location while another thread is accessing it. This approach appears to be the only plausible one for C and C++. However, it can only succeed if the definition of a data race is made precise enough for programmers, compiler writers, and hardware to know when data races occur and how to avoid them. Currently, it is not defined at all. Among other consequences, unexpected compiler transformations regularly break multithreaded programs (as illustrated in the example that follows).
2. Defining an atomic operations library to allow the construction of correct multithreaded programs without locks. This does not directly affect most existing application-level programs, though a significant number of them should be modified to use this library in order to ensure correctness. Such a library is necessary for development of portable core libraries and infrastructure code that increasingly use lock-free techniques to implement high-performance synchronization support. Defining an atomics library relies critically on the semantics of memory operations and data races.
3. Designing a threads API that meshes better with the rest of the C++ language.

We expect that the first two issues and their solutions also apply, with minor modifications, to C. And compatibility would be greatly desirable. We expect the last issue is mostly C++ specific,

though there are likely to be exceptions, such as support for thread-local storage.

As a result, we would like to encourage members of the C committee to follow our discussions and to provide input, particularly if they see aspects of our approach that would make it less palatable to the C committee, and hence lead to unnecessary divergence between C and C++.

A Simplified Example

We illustrate some of the problems addressed by this work with a simple case in which the current language specifications for C and C++ are clearly inadequate for multithreaded programs. This is only one among many possible examples. It helps demonstrate that the problems are in fact profound and must be addressed by the language specification and compilers. It also points out that the expected impact on compilers is likely to be nontrivial.

Consider the following declarations and function definition:

```
int global_positives = 0;
typedef struct list {
  struct list *next;
  double val;
} * list;

void count_positives(list l)
{
  list p;
  for (p = l; p; p = p -> next)
    if (p -> val > 0.0)
      ++global_positives;
}
```

Now consider the case in which thread A performs

```
count_positives(<list containing
only negative values>);
```

while thread B performs

```
++global_positives;
```

This should be perfectly correct, since count_positives in this specific case does not update global_positives, and hence the two threads operate on distinct global data and require no locking.

But some existing optimizing compilizers (including gcc, which tends to be relatively conservative) will "optimize" count_positives to something similar to

```
void count_positives(list l)
{
  list p;
  register int r;

r = global_positives;
  for (p = l; p; p = p -> next)
    if (p -> val > 0.0) ++r;
  global_positives = r;
}
```

This transformation is clearly consistent with the C language specification, which addresses only single-threaded execution. In a single-threaded environment, it is indistinguishable from the original.

The pthread specification also contains no clear prohibition against this kind of transformation. And since it is a library and not a language specification, it is not clear that it could.

However, in a multithreaded environment, the transformed version is quite different, in that it assigns to global_positives, even if the list contains only negative elements. Our original program is now broken, because the update of global_positives by thread B may be lost, as a result of thread A writing back an earlier value of global_positives. By pthread rules, a thread-unaware compiler has turned a perfectly legitimate program into one with undefined semantics.

This is a contrived example, but similar issues have been encountered in practice, and these are discussed in more detail in Boehm [1]. We hope this has served as a brief introduction to the kind of problems we are trying to address and will encourage others to follow the discussion.

REFERENCES AND FURTHER READING

[1] H.-J. Boehm, "Threads Cannot Be Implemented as a Library," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pp. 26–37, 2005. Also available at http://www.hpl.hp.com/techreports/2004/HPL-2004-209.html.

[2] JSR-133 Expert Group, JSR-133: Java Memory Model and Thread Specification: http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf, August 2004.

[3] J. Manson, W. Pugh, and S. V. Adve, "The Java Memory Model," in *Conference Record of the Thirty-Second Annual ACM Symposium on Principles of Programming Languages*, January 2005. Also available at http://www.cs.umd.edu/users/jmanson/java/popl05.pdf.