Are Safe Concurrency Libraries Possible?

Peter A. Buhr
Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1
E-mail: pabuhr@uwaterloo.ca

September 11, 1995

There has been a significant effort over the past decade in creating concurrency libraries for a number of sequential programming languages, particularly for object-oriented ones such as C++ [BLL88], Eiffel [KB93], and Smalltalk [YT86]. While the goal is laudable, this discussion shows that obtaining concurrency solely through libraries is, in general, unattainable in both shared-memory and distributed environments. An informal proof shows that while it is possible to build a concurrency library and applications that use it, no guarantee of correctness can be given without an unrealistic requirement being placed on the library designer and user, even when the library features are used correctly in a well formed concurrent program. This unrealistic requirement is the need to examine the executable instructions after every compilation of the library or application for incorrectly generated code. Furthermore, if incorrect code is found, it may not be possible to fix the problem directly. The library or application may have to be restructured, possibly significantly, to produce a working program. Such a requirement is clearly beyond that of any reasonable, practical technique, tool, or programmer. In other words, if a concurrency library imposes such an inspection and repair requirement, the library feature cannot be safely implemented. Moreover, such requirements violate the abstraction and encapsulation of the concurrency library itself because of the need to examine the generated code.

The proof is based on the observation that any linguistic feature that affects code generation cannot be implemented in a safe and portable way outside of a translator (compiler, assembler, etc.). Without language-based constructs to indicate concurrent semantics, an underlying translator may generate code that is correct for sequential execution but incorrect for concurrent execution. Without examining the generated code after each translation, a programmer cannot assess and guarantee the correctness of even a well formed program. The proof does not assume a particular programming language or concurrency paradigm; it will be shown that all concurrency library approaches may suffer from the unrealistic requirement.

While this observation holds generally, three examples of code generation situations suffice to demonstrate problems with concurrency libraries. The three examples are: storage of values in registers, code re-ordering, and instruction substitution. Compilers and assemblers use these and other techniques to generate efficient code, particularly for RISC computers. In fact, most RISC computers rely heavily on these and other techniques to achieve their stated performance specifications. While these techniques are valid for sequential execution, the temporal nature of concurrent programs can result in problems. The following anecdotes, which have all affected this author, illustrate the problems that can occur with each code optimization technique.

When a compiler moves the location of a variable into a register, the current value of the variable becomes hidden from other tasks. For example, in C:

```
flag1 = flag2 = 0;
```

task$_1$	**task$_2$**
`while ( flag2 == 0 );`	`flag2 = 1;`
`...`	`while ( flag1 == 0 );`
`flag1 = 1;`	`...`

Communications of the ACM, 38(2), Feburary 1995, pp. 117-120

if the compiler loads variables flag1 and flag2 into registers, neither task can see the changes made to the shared variables and the tasks spin forever in their busy loops. This particular behaviour is clearly unexpected and cannot be debugged without examining the assembler code. Furthermore, all software solutions to mutual exclusion, such Dekker's and G. L. Petersons' algorithms, use this technique, so it is important. This problem also affects exception handling and is the main reason that the declaration qualifier volatile was added to C.

The surreptitious movement of code by translators to fill instruction pipelines causes problems for both the implementor and user of a concurrency library. For example, in:

```
acquire(lock)
critical section
release(lock)
```

the locking routines are often inlined for efficiency reasons. This optimization might be done explicitly by the lock designer or implicitly by the compiler. However, code reordering allows the machine instructions at the boundaries between the locking operations and the critical section to become intermixed. The critical section code can start execution before the lock is acquired, or the lock can be released before the critical section has completed execution. In the extreme case, the code to release the lock, which may be a single instruction, is moved *before* the critical section. Even if the code for the locking routines is not inlined, code in the critical section might still be moved before or after them if the critical section only accesses local variables on the grounds that the locking routines cannot see the local variables and have no aliases to them.

Some assemblers substitute multiple instructions for a single instruction generated by a compiler or assembler programmer. Unfortunately, this can cause errors in concurrent programs. For example, on a MIPS computer:

written	**substituted**
lw $sp,PrivateStack	lui $sp,4097
	lw $sp,-15288($sp)

the loading of the stack pointer register, $sp, with an absolute address, PrivateStack, has been changed into two loads, which load the absolute address in two parts, high and low order 16 bits, because there are 16-bit immediate values in instructions. However, if a context switch occurs between the substituted instructions because of an interrupt/signal, the stack pointer is invalid, which may cause a failure on systems that use the user stack to deliver an interrupt. This problem is not specific to the stack pointer; any transformation of an atomic action into a non-atomic action may cause problems. Without knowing which operations are truly atomic, a library cannot guarantee correctness.

One option is to forbid such optimizations by the translator because in certain situations they cause failures in concurrent programs. However, even if possible, this is too Draconian. Concurrent code situations that cause problems occur infrequently in most concurrent programs. Therefore, if optimizations occur, it is necessary to specify when they cannot be performed throughout the entire transformation hierarchy (compiler, assembler, linker, loader). There are three approaches:

1. provide some explicit language facilities to control optimizations (e.g. pragma, volatile, etc.),
2. provide some concurrency constructs that allow the translator to determine when to disable certain optimizations,
3. a combination of approaches one and two.

Notice that the programming language has been extended over its sequential counterpart in all three approaches. As an aside, there has been some suggestion in the C++ community and C POSIX standards committee for a standard concurrency-library interface, which the compiler is aware of, so that correct code can be generated. However, if the compiler *must* be aware of the concurrency library, this is the same as extending the language.

In the first approach, the user is responsible for explicitly controlling all optimizations that could affect correctness. For example, it may be necessary to declare the variables, flag1 and flag2, used for synchronization in the previous example, as shared (or volatile in C parlance) so that the program works correctly. A user would only know that an additional declaration qualifier is needed through an understanding of potential problems with code optimization. If a user does not deactivate a particular optimization for a particular architecture or system, the program may fail. An overcautious user might deactivate too many optimizations and pay an unnecessary runtime performance penalty. Therefore, this approach can be rejected because it requires extensive knowledge of each translator's implementation to know what and when certain optimizations are applied, which constitutes another unrealistic requirement for the library designer and user. This unrealistic requirement applies to each translator and must be rechecked if the translator changes (possibly by examining the generated code, leading back to the first unrealistic requirement). Furthermore, whatever facility is provided, it must be completely open-ended because new code optimizations can appear at any time. Finally, optimizations vary from translator to translator and from system to system, which makes writing portable code extremely difficult.

In the second approach, the translator is responsible for detecting *all* temporal situations and ensuring correct and efficient code generation. To accomplish this task, the translator must be aware that a program is concurrent through specific concurrency constructs. The exact form of these constructs and the concurrency paradigm that results from them is not important in this discussion. It is sufficient to point out that the programming language is extended. However, the form of the concurrency constructs must be extensive enough so that the translator can adequately determine when specific optimizations can and cannot be applied. (It is beyond the scope of this short discussion to talk about a "minimal set" of concurrency constructs that allow a compiler to determine this fact.) Notice, that the translator must not only handle all usages of its specific concurrency constructs correctly, but also situations like the shared synchronization flags (flag1, flag2), even if such cases are outside of the concurrency paradigm supplied by the concurrency constructs.

The last approach is a combination of the previous two. In this approach, the translator ensures correct code generation for its supplied concurrency constructs, but allows incorrect generation of code for temporal situations outside of these constructs, such as the shared synchronization flags. To make synchronization work using flag variables would require capabilities from approach one to ensure correct code generation, which implies some understanding of general implementation/optimization details. However, with a reasonable complement of concurrency constructs in a language, a user would never have to use constructs that require knowledge of optimizations. Special situations would arise only for users performing unusual concurrent operations or possibly for teaching purposes. Essentially, this last approach admits that outside of the specific concurrency constructs, the translator may not be able to automatically detect all possible concurrent situations that could appear in a program or that it may be too expensive to analyze an entire program to detect them all.

Virtually all concurrency libraries, including the ones build by this author [BS90, BDS$^+$92], are built on an unsafe foundation that can result in unexpected failure both inside the library or in user programs. To ensure correct operation, most of these libraries rely on the fact that many compiler optimizations do not cross a routine-call boundary. However, with the advent of more powerful global optimizations and the desire to inline small routines for efficiency reasons, this is no longer a valid assumption. While the above proof is informal, its conclusion, that safe concurrent programs require extensions to the programming language they are written in, follows directly from the observation that concurrency is a linguistic feature that affects code generation. It also seems clear that it is impractical to just control optimizations independently of concurrency. Translators need to know what is going on in a program to generate correct code and still be able to perform adequate optimizations. Therefore, programming languages without concurrency constructs *must* be extended with sufficient concurrency capabilities to ensure their correct operation and to allow some reasonable degree of portability.

Finally, while this discussion has been independent of the concurrency constructs added to a language, it is important to point out that there are other high-level language issues that can interact with concurrency. Language features like exception handling, polymorphism, persistence, and distributed programming all interact with concurrency in both obvious and subtle ways. In many cases, it is not possible for concurrency libraries to deal with these interactions, leading to imposed restrictions or coding conventions on library users that increase the potential for runtime errors (see [BDZ89, BD92] for those relating to C++). Again, the only way the translator can statically check for incompatible interactions among language features, so that it can print appropriate warnings and errors, are concurrency extensions.

Acknowledgment

I would like to thank Glen Ditchfield and Doug Lea for taking the time to read and comment on the ideas presented here.

References

[BD92] Peter A. Buhr and Glen Ditchfield. Adding Concurrency to a Programming Language. In *USENIX C++ Technical Conference Proceedings*, pages 207–224, Portland, Oregon, U.S.A., August 1992. USENIX Association.

[BDS+92] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μC++: Concurrency in the Object-Oriented Language C++. *Software—Practice and Experience*, 22(2):137–172, February 1992.

[BDZ89] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Adding Concurrency to a Statically Type-Safe Object-Oriented Programming Language. *SIGPLAN Notices*, 24(4):18–21, April 1989. Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, Sept. 26–27, 1988, San Diego, California, U.S.A.

[BLL88] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A System for Object-oriented Parallel Programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[BS90] Peter A. Buhr and Richard A. Stroobosscher. The μSystem: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software—Practice and Experience*, 20(9):929–963, September 1990.

[KB93] Murat Karaorman and John Bruno. Introducing Concurrency to a Sequential Language. *Communications of the ACM*, 36(9):103–116, September 1993.

[YT86] Yasuhiko Yokote and Mario Tokoro. The Design and Implementation of ConcurrentSmalltalk. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings*, pages 331–340, Portland, Oregon, September 29 1986. Association for Computing Machinery, SIGPLAN Notices 21(11).