Automated Verification of UPC Memory Consistency

Øystein Thorsen Charles Wallace

Michigan Technological University

1 Introduction

The *UPC* programming language [3] is a shared memory extension to ANSI C. Its memory consistency model is relaxed, allowing for a high degree of optimization, but also permitting behavior which may be surprising to the naïve programmer. To allow better understanding of this memory model, we present a tool [4] for analyzing the behavior of UPC programs. Given an execution trace, the tool determines whether the results are compatible with the UPC memory model. The tool is targeted at newcomers to UPC who want to learn about its memory model and at developers who want to verify possible behaviors of their programs.

This work was inspired by Yang's verification tool [5] for the Intel Itanium memory model. Based on the memory model definition in the UPC specification [3], we have devised a procedure for converting an execution trace into a propositional logic formula that is satisfiable if and only if the execution is compatible with the memory model's notion of consistency. We then use the SAT solver zChaff [6] to determine consistency.

2 UPC and its memory model

In UPC, multiple *threads* (computing agents) execute a common program concurrently. In keeping with the shared memory paradigm, threads access shared memory addresses concurrently through standard read and write instructions, rather than through explicit message passing. Program behavior is affected by the order in which memory operations are made visible to threads. By constraining the order of operations, a memory model determines which values may be returned by each read operation. UPC introduces

two *memory modes*, *strict* and *relaxed*. The mode of each shared memory access within a UPC program affects the visibility of those accesses during program execution. As the names suggest, program behavior under strict consistency is more constrained than that under relaxed consistency.

UPC includes support for global synchronization. The upc_barrier statement causes a thread to halt execution until all other threads have executed their corresponding barriers. Once each thread has reached its barrier, all threads may resume execution. From an implementation standpoint, a barrier is composed of two distinct operations. A thread first *notifies* all other threads that it has reached the barrier and then waits until all threads have reached the barrier. UPC also offers a *split-phase* barrier, consisting of the paired statements upc_notify and upc_wait. Separating a barrier into two separate instructions gives the programmer the opportunity to do local computations after notifying but before having to wait.

Every *UPC-consistent* execution is "explainable" in terms of a (global) precedence relation and thread-local precedence relations on the memory accesses in the execution. In this paper, we restrict attention to shared memory accesses. The global relation $<_{strict}$ linearly orders strict operations, including operations by different threads. This serves as a globally agreed-upon sequential order on strict accesses. This ensures sequential consistency for strict consistency mode. Also, for each strict access S by a given thread t, $<_{strict}$ orders S after t's preceding relaxed accesses and before t's following relaxed accesses. Thus a strict access serves to globally order certain relaxed accesses before others.

Each local relation $<_t$ is a linear order, on t's operations and all writes and strict operations, that represents thread t's view of the execution. This relation must agree with $<_{strict}$. Furthermore, it must agree with the observable behavior of the execution (the values returned by reads): each read operation must return the value written by the most recent write operation to the given location (a well-defined notion, since $<_t$ is linear).

Fence and barrier statements are defined as null strict operations. The difference between a fence and a barrier is that the latter is *collective*; that is, assumed to be executed by all threads. Hence a fence brings a *single* thread up to date in terms of memory consistency, while a barrier does the same for *all* threads.

3 Input and output

The input to the tool includes the "program order" of operations executed by each thread, the values returned by all read operations, and initial values for each memory location. For examples, see §A. In a valid execution trace, all threads call the same collective operations in the same order, each upc_wait is preceded by a unique upc_notify, and no collective calls are allowed between a upc_notify and a upc_wait.

Relaxed write operations have a special status. Each relaxed write is observed by threads other than the thread that issued it; however, threads may observe relaxed writes in different orders. The result in Example 1 is UPC-consistent precisely for this reason. One way to conceive of this is to define each relaxed write as a "family" of writes: the "real" write, observed by the thread that issued it, and for every other thread, a "virtual" write that it observes. In an explanatory order $<_t$ for a thread t, all of the virtual relaxed writes, issued by other threads, must be included. In contrast, a relaxed read is only included in the explanatory order of the thread that issued it, so there is no need for virtual copies of relaxed reads.

If the execution trace is UPC-consistent, we want to show a graph displaying the observed ordering for each thread, and if it is not UPC-consistent we want to show why. For this we use a graph drawing package [1] that displays the operations as nodes in a directed graph. Operations are color-coded and organized in columns based on the observing thread. Strict operations (including barriers) are displayed in black in a separate column, since they are part of the common strict order for all threads.

Two operations may have a directed edge between them showing which was observed before the other. Since the graph represents a temporal order of operations, the presence of a cycle indicates UPC-inconsistency. In the case of UPC consistency, we remove edges implied by transitivity from the displayed graph. For UPC-inconsistent execution traces, we know there must be a cycle preventing this temporal order, so we show all the edges involved in that cycle (or cycles).

4 Implementation

Each operation is encoded as a tuple containing all the relevant attributes (*e.g.,* issuing thread, memory mode, memory location, value read/written). The program reads the user input and converts it into a set of tuples, using a bison/flex parser. The tuples are then fed into the SAT generation pro-

gram. These tuples, along with constraints derived from the memory model, provide the input for generation of the SAT instance.

For human consumption, we give a high-level characterization of the UPC memory model [2], closely following the definition in the UPC specification [3]. We use two predicates: **order** i j to indicate that operation i is ordered before operation j, and **reads** i j to indicate that read operation i gets its value from write operation j. The predicates **order** and **reads** are each encoded as sets of variables in the SAT encoding.

When all the clauses are generated the SAT solver tries to solve the problem. If a solution is found, the program prints a graph showing a possible ordering. If a solution is not found, we extract the cycle from the SAT encoding and output a graph showing the cycle. Finding a satisfying variable assignment is a state-space search where a good heuristic function and good pruning can drastically reduce the time it takes to find a solution. This approach has shown great promise in the last few years [7].

5 Future Work

Currently, the tool is in a rather primitive state. The input requires a manual conversion from actual UPC source code to an idealized execution trace. For programs of significant size, this process is unwieldy and results in rather opaque results. Thus an imperative for us is to derive execution traces automatically from UPC code. We feel that this initial work shows promise for further development, both as a verification tool for UPC developers and as a pedagogical tool for students.

References

[1] GraphViz home page. http://www.graphviz.org.

[2] Øystein Thorsen. Automated verification of UPC memory consistency. Master's thesis, Michigan Technological University, 2006. Available at http://www.upc.mtu.edu/papers/UPCVerifier.pdf.

[3] The UPC Consortium. *UPC Language Specifications V.1.2*, May 2005.

[4] UPCVerifier software. http://www.upc.mtu.edu/applications/UPCVerifier.html.

[5] Yue Yang. *Formalizing Shared Memory Consistency Models for Program Analysis.* PhD thesis, University of Utah, 2004.

[6] zChaff home page. http://www.princeton.edu/~chaff/zchaff.html.

[7] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Computer Aided Verification*, pages 582–595, 2002.

A Examples

The following examples are derived from §B5 of the UPC Language Specification [3], which provides examples of UPC-consistent and UPC-inconsistent behavior. The examples below correspond to examples 1, 6 and 8 in the Language Specification.

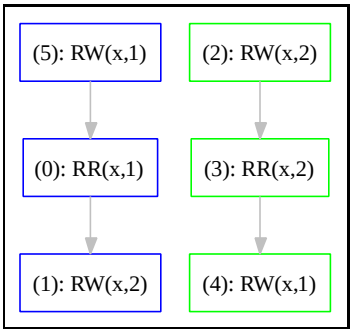
Example 1

```
startvalues
x = 0

numthreads = 2

thread
RR(x,1)
RW(x,2)

thread
RR(x,2)
RW(x,1)
```



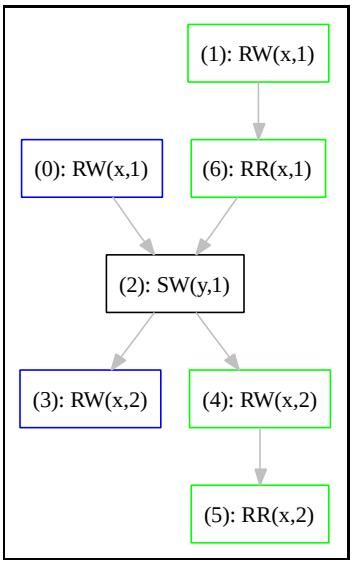
This is an example of UPC-consistent behavior. As shown in the tool input, each thread performs a relaxed read followed by a relaxed write. The result may be counterintuitive to programmers accustomed to sequential consistency, since no global order of all operations can explain the values read. However, the tool output shows explanatory orderings for each thread. Note that the operations labeled (1) and (2) are copies of the same relaxed write (similarly, (4) and (5)). The threads order the two relaxed writes differently. Moreover, each thread's explanatory ordering includes only the relaxed reads issued by that thread. Informally speaking, it is not the responsiblity of a thread to explain the relaxed read values of *other* threads.

Example 2

```
numthreads = 2

thread
RW(x,1)
SW(y,1)
RW(x,2)

thread
RR(x,2)
RR(x,1)
```



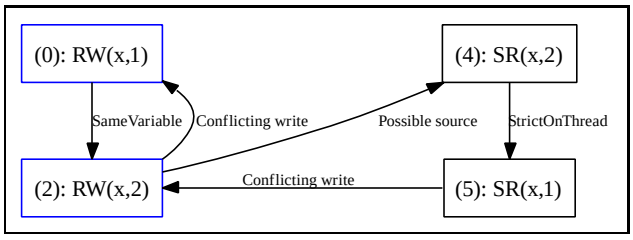
This is another UPC-consistent example. In the first thread's execution, the presence of the intervening strict write implies that all threads must observe the two relaxed writes in program order. The (potentially counter-intuitive) read results are due to the fact that the second thread may observe its own reads out of order, as indicated in the tool output.

Example 3

```
numthreads = 2

thread
RW(x,1)
RW(x,2)

thread
SR(x,2)
SR(x,1)
```



This behavior is UPC-inconsistent. The relaxed writes must be observed in program order since they operate on the same location, and the reads must be observed in program order since they are strict. The first read result of 2 creates a write-read dependency that forms a cycle in the output graph.