Hobbes: CVS for Shared Memory

Derek R. Hower and Mark D. Hill
University of Wisconsin-Madison
Department of Computer Sciences
1210 W Dayton St
Madison, WI 53706
{drh5, markhill}@cs.wisc.edu

Abstract

We observe that many research proposals for enhanced multicore programmability share a common mechanism, namely, they limit how and when values in a shared memory program are updated. In this paper we present Hobbes, a generic mechanism to control shared memory updates that takes inspiration from software revision control systems like CVS. We show that two simple primitives, checkout and checkin, can be used to simplify programming and debugging, build deterministic software, and enable deterministic replay. Our preliminary evaluation indicates that the Hobbes programming model fits with many program's existing data sharing patterns and has the potential to support new, more intuitive patterns.

1 Introduction

Many of the research proposals introduced since the dawn of the multicore era that aim to make shared memory programming easier share a common mechanism. All in some way limit how and when values in a shared memory program are updated, and effectively reduce the potential to unexpectedly share data (e.g., through a data race). Deterministic hardware systems like DMP [12] and Calvin [17] delay update propagation until deterministic events occur. Languages such as X10 [11] and DPJ [7] constrain how memory from differing regions can communicate. Systems extensions like Grace [4] and Determinator [2] hide updated values from other threads until an explicit update event occurs. Many other examples exist in both hardware and software [1,10,16,20,23,24].

Given the above observation about the trends in multiprocessor programmability research, we aim to develop a generic framework for controlling how and when updates in a shared memory programming environment occur. We hope that if designed correctly, such a framework could be used to achieve the specific goals of all the proposals listed above using a single mechanism. Such a unified mechanism could simultaneously give hardware designers a single operation to optimize and software tools a single primitive to use.

Rather than reinvent the wheel, we take inspiration from a concurrency management abstraction most programmers are already familiar with: software revision control systems. Development teams have been successfully using revision control systems for well over two decades, and the model has proven to be simple enough to understand yet powerful enough to enable complex interactions. And importantly, like the proposals for shared memory above, revision control systems operate by limiting how and when updates are published.

In this paper we propose Hobbes, a software abstraction that takes inspiration from the CVS concurrency model [15]. Hobbes gives programmers the ability to checkout and checkin shared memory on a per-thread basis, much in the same way that revision control systems enable file checkout and checkin on a per-user basis.

We show that the Hobbes model can be used as a foundation to develop mechanisms that achieve a variety of features previously proposed for multiprocessor systems. In particular, Hobbes can be used to constrain a multithreaded program's interleaving, enable multithreaded deterministic replay, and allow for multithreaded deterministic execution. These are in addition to other benefits of the Hobbes model such as improved code readability and the ability to manage shared memory in new and intuitive ways. And importantly, Hobbes can achieve those goals with a single mechanism.

We discuss Hobbes here largely without regard to implementation considerations, and focus specifically on the merits and limitations of the model. Our current and future work, some of which is discussed below, will focus on how the model can be efficiently implemented in both conventional and specialized hardware.

We also discuss our experiences using an early prototype of Hobbes that can be used as a software library. We find

- that existing programs written for shared memory systems are already written in a way compatible with the checkout/checkin operations, at the very least showing that the operations are reasonable abstractions;

- that the checkout/checkin mechanisms are sufficient to implement multiprocessor determinism strategies previously proposed;
- the checkout/checkin mechanisms can be used by programmers to reason about multiprocessor communication;

In the following sections we will first outline the details of our current working definition of the Hobbes model. Then we will discuss how the Hobbes model can be used in a variety of ways to help programmers deal with multiprocessor machines, including by controlling and/or eliminating nondeterminism. We will also briefly discuss our experiences with Hobbes-U, a working prototype. Finally, we will outline our plans to investigate the Hobbes model further.

2 Hobbes Model

The Hobbes framework is used by threads in an application to manage data sharing through memory. When using Hobbes, all communication through shared memory is explicit and is controlled through an interface described below. This is in contrast to conventional shared memory in which threads may communicate between any two memory references.

Memory is managed by Hobbes through the use of four main abstractions borrowed from CVS, namely modules, working copies, repositories and checkout/checkin operations. Though all four abstractions share a likeness to their counterparts in CVS, several differences exist that make them unique to shared memory programming.

2.1 Memory Repositories

Every Hobbes application contains exactly one memory repository, which in turn contains zero or more modules. Threads communicate by pushing/pulling updates to/from modules contained in the repository.

2.2 Modules

Modules are abstractions use to group related addresses together in an indivisible unit. A module consists of one or more contiguous ranges of addresses in a process address space. Ranges cannot overlap and an address may belong to at most one module. Any address that is not defined by a module is considered private; threads are guaranteed that any update made to such an address will not be seen by other threads, similarly to an address in a forked process.

Modules serve as the basis of sharing in Hobbes. In particular, threads communicate by passing different versions of modules between their working copies and the repository.

Address Space Layouts

The module/repository abstraction allows for many different address space layouts. Here, we discuss a few layouts that may be common in Hobbes applications and attempt to provide intuition on why the Hobbes model allows programmers to more effectively manage shared memory.

One of the simplest layouts uses a single module that contains the entire heap. In this layout, threads can share information through values on the heap but cannot communicate through the stack. This layout has the benefit of matching up with common intuition on how programs communicate, but may be overly inclusive as typically not all addresses on the heap are intended to be shared.

A modification of the entire-heap layout would be to use a single module strictly for shared heap allocations. This layout would require a modified memory allocator capable of distinguishing between a private or shared allocation (e.g., with pmalloc and smalloc). Using a separate module for private allocations could help prevent heisenbugs that occur from inadvertent sharing in private memory regions and, depending on implementation details, could result in performance benefits since the Hobbes system has fewer memory locations to track.

Other layouts could take advantage of multiple modules within the same application. For example, an application could use one module for the shared data of each subtask in an application.

2.3 Working Copies

A working copy is a thread's private representation of a module. Threads can be assured that any memory value in the working copy will not change spontaneously due to another thread's update as they could in conventional shared memory. Likewise, a thread can modify locations in a working copy at will with the assurance that other threads in the same process will not prematurely see those updates.

Working copies in Hobbes differ from those in CVS in at least one notable way. A Hobbes working copy cannot reflect any previous version in the history of a module as can a working copy in CVS. Programmers using CVS have access to different versions of project that they can use to track down bugs, revive dead code, and/or develop an understanding of the evolution of a project. While this model is useful for human programmers, threads in a shared memory program are not generally interested in finding and/or correcting work committed in the past, and so we chose to limit what Hobbes working copies can reflect. In particular, working copies in Hobbes always reflect the most recent version of a module at the time of the last checkout operation. In CVS terminology, this is analogous to a working copy that always points to the HEAD revision.

2.4 Checkout/Checkin

In the Hobbes model, threads can only communicate through memory by moving data between their respective working copies and the encompassing repository. Two operations, checkout and checkin, are used for this purpose. A checkout operation atomically pulls the most recent version of a module into a thread's working copy. Similarly, a checkin operation atomically pushes any changes made in a working copy back to the repository.

Checkout and checkin operations are totally ordered over the course of execution. That order is used to determine what version of a module at the repository a thread should see. Upon a checkout, a thread will see all updates from checkins that occurred before the checkout and none that occurred after.

2.4.1 Patches

The use of checkout and checkin operations create logical patches during execution. A patch consists of all updates made to a working copy since the last checkout. The checkin operation applies the current patch to the repository. Because corresponding checkout and checkin operations do not have to be consecutive in the total order of operations, patches are partially ordered during an execution. As a result, it is possible for a conflict to occur between concurrent patches.

2.4.2 Conflicts and Merging

A conflict occurs when two concurrent patches have overlapping updates. When a conflict occurs, it must somehow be resolved, and in Hobbes many possibilities exist. Below we will discuss two of these possibilities, though many more could be applied. We envision that future Hobbes systems may even allow programs to dynamically choose a conflict resolution policy.

One of the simplest, yet still useful, policies is a last-writer-wins scheme. Under this policy, conflicting updates blindly clobber old values as determined by the checkin order. For example, if two concurrent patches both update an address A, only the value from the patch with the oldest checkin time will survive in the repository. In our experience using Hobbes so far, we find that under most situations the clobbered update is either tolerable (accounted for in the algorithm's design) or is an obvious indication of a bug. In the later case, it may be beneficial for a Hobbes system to provide an error notification, e.g., via an exception, when a conflict occurs.

Another possibility for resolving conflicts is to use a user-controlled resolution function, similar to a manual merge in CVS. Here, the user (program) can dynamically choose what to do when a conflict occurs, presumably through a callback function. This resolution policy is particularly interesting in its possibility to enable, for example, lock-free data structures through undo actions. Unfortunately, we have not yet evaluated this policy though hold high hopes for its utility.

3 Uses

Below we will discuss how a Hobbes framework can be used to achieve a variety of research goals, including deterministic execution, easier debugging, and support for hard to implement algorithms. Other possibilities may exist, such as supporting a software transactional memory implementation, but are not discussed here.

3.1 Code Annotation

Because the checkout/checkin mechanisms explicitly identify when threads communicate, they can serve as visual cues in the source code that state a programmer's intention. This can be especially helpful in large projects where more than one programmer works on the same software module. Unlike disciplined software engineering practices that have attempted to address this very problem (e.g., shared variable naming conventions), Hobbes forces, rather than suggests, the source code to be self-documenting, potentially leading to more readable and usable source code even in the face of shaky engineering practices.

Additionally, because of the self-documenting features of Hobbes, multithreaded debugging may be simplified. In particular, when debugging an application a programmer can be assured that if she observes a bug in a region of code that does not contain a checkout or checkin, that that bug is due to a local (i.e., single threaded) error.

One major obstacle to Hobbes' use as a code annotator is software composability. When checkout/checkin operations are hidden in a function call programmers may not be aware that communication is occurring. Hobbes could benefit from solutions to similar composability problems previously proposed, such as Java's insistence on listing throwable exceptions at a function's declaration.

3.2 Interleave Constrained Execution

A program that uses Hobbes can only communicate at checkout/checkin operations. This is in contrast to conventional shared memory programs that can potentially communicate between any two memory accesses. The reduced possibility for communication results in exponentially fewer potential thread interleavings, and consequently a program that is easier to develop, test, verify, and maintain. Once a program is verified using Hobbes, developers can have a higher confidence that released software will be free of

concurrency errors, potentially saving millions of dollars a year in development and support costs.

3.3 Deterministic Replay

While simply using Hobbes goes a long way towards making multithreaded executions more predictable, the resulting executions can still be nondeterministic. Sometimes this nondeterminism is even intentional, e.g. when designing a load balancing work queue [6]. However, even when nondeterminism is desired for algorithmic purposes there are times when replicating a previous interleaving can be helpful. Deterministic replay allows debuggers to hone in on rarely occurring bugs [27], enables software replication of multithreaded programs [8], gives security analysts insights into obscure attacks [13], and can even be useful to for application-specific tasks such as database queries [25]. Unlike many existing proposals to enable multithreaded deterministic replay [18,19,21,22,27], Hobbes can be used to build a replay system without requiring replay-specific hardware (though still taking advantage of any hardware acceleration present for Hobbes), as described below.

Recording the memory interleaving order, which is a key component of any multiprocessor deterministic replay mechanism, in Hobbes is nearly trivial. It requires only that the order of checkout/checkin operations are recorded. Replay is only slightly more complicated, and consists of a system capable of enforcing a predetermined checkout/checkin order (and, of course, a mechanism for replaying inputs). A Hobbes-based deterministic replay mechanism would be lightweight enough that it could be used online in production systems to collect detailed bug reports on concurrency errors or to forward ordering information to replica processes for fault tolerance [8].

3.4 Deterministic Execution

While deterministic replay can be helpful, in many cases it may not be needed. Many parallel algorithms can efficiently be made deterministic with a Hobbes system, making the need to record a thread interleaving unnecessary. Eliminating the recording step can reduce bandwidth requirements in a software replication scheme or increase testing confidence in multithreaded programs to be on-par with that of single-threaded applications.

At the very least, Hobbes can be used as a mechanism to implement deterministic execution in a manner similar to current state-of-the-art [3,12,17]. Most current systems achieve determinism by stratifying (or quantizing) a multithreaded execution. After partitioning execution into a series of global strata, these systems ensure that threads communicate only at strata boundaries and do so in a deterministic order. Hobbes can be used to implement these systems by using an ordered barrier operation that, in addition to performing like a conventional barrier, also ensures that threads entering/leaving the barrier logically checkin/checkout in a deterministic order.

Programmers could manually insert calls to the ordered barrier to make their application deterministic. Alternatively, it is conceivable that a special runtime system could be constructed that transparently inserts ordered barriers during execution, making existing programs deterministic without modification.

In addition to being a mechanism for implementing the current state of the art in deterministic execution, Hobbes also has the potential to improve the state of the art in at least two ways. Both approaches attempt to tackle the problem of load imbalance that is one of the major performance limiters in strata-based systems [3,12,17].

First, Hobbes exposes the mechanism for achieving determinism (i.e., the ordered barrier) to the programmer. This presents the opportunity for high-level information about a program to be taken into consideration when creating strata and may prevent situations where strata are created at inopportune times, such as during periods where threads are not communicating. Current systems that create strata without any high level program information are not able to avoid these performance pitfalls.

Second, Hobbes allows programmers to use more precise instruments than barriers for controlling determinism. Generally, as long as the order of checkout/checkin operations is controlled in a repeatable way, then a deterministic execution results. This gives programmers the opportunity to encode a deterministic schedule that corresponds to the actual communication patterns of an application, which may not follow the barrier model. This gives programmers the flexibility to design complex deterministic programs without needlessly sacrificing performance due to a communication pattern mismatch.

4 Hobbes-U

Hobbes-U is the first prototype of a Hobbes model implementation. It is implemented entirely as a userspace library for C and C++. Hobbes-U is not particularly efficient or fast, but serves a key purpose by allowing software development with the Hobbes model to proceed. Hobbes-U also allows us to collect key statistics on program usage that will motivate future improvements in a Hobbes implementation. Hobbes-U was designed for functionality, not efficiency, and has a steep performance cost.

Table 1 – Hobbes-U Interface

`int hobbes_prepare()`
`void hobbes_finish()`
`hobbest_module_t create_module()`
`void* smalloc(size_t len, hobbes_module_t m)`
`int mcheckin(hobbes_module_t m)`
`int mcheckout(hobbes_module_t m)`
`void ordered_barrier(pthread_barrier_t b, hobbes_module_t m)`

Threads in Hobbes-U manage memory using the interface shown in Table 1. Before using any Hobbes abstractions, a program must first make a call to `hobbes_prepare()`, which initializes hidden library state. Users can manage modules through the combined use of `create_module()` and `smalloc(…)`. `create_module()` initializes a new, empty and module in a thread's working copy and return a handle to it. Subsequent calls to `smalloc(..)` that use that handle add memory regions to the module. `mcheckout(…)` and `mcheckin(…)` are used to pull or push, respectively modules from a working copy to the repository. Finally, the Hobbes-U interface also includes a special ordered barrier synchronization operation so that deterministic programs can be written.

Additionally, Hobbes-U serves as a drop-in replacement for the pthread library, enabling existing pthread software to use Hobbes memory management without substantial (if any) modification. The Hobbes implementation of pthread library calls includes implicit checkout/checkin operations when communication is expected (e.g., at barrier_wait).

Hobbes-U utilizes existing virtual memory mechanisms and performs all operations under the hood at the page level. This is transparent to the user, however, who can still create and operate on modules of arbitrary size. Many techniques used by Hobbes-U have been borrowed from other systems that manage shared memory through paging mechanisms, such as Grace, Determinator, and various software DSM systems [23]. For this reason and space considerations we omit further details of the Hobbes-U design.

5 Preliminary Evaluation

Our preliminary evaluation of Hobbes-U is focused on the usability of the Hobbes model, and we purposely omit any performance evaluation at this time. Specifically, our evaluation seeks to determine if the Hobbes model a) is general enough to support a variety of programs, b) matches with programmer's existing intuition on how to manage share memory, and c) enables new, more organized and/or modular ways to manage shared memory.

To answer these questions, we used the Hobbes-U prototype to develop a suite of applications that include workloads from the popular PARSEC benchmark suite [5] as well as custom from-scratch implementations of well known algorithms like ocean and barnes-hut. We modified the applications from PARSEC as little as possible, and use them to test Hobbes' compatibility with existing techniques for managing shared memory. Our custom workloads take a different approach and are used to gain experience writing programs specifically for the Hobbes model.

5.1 Qualitative Findings

Of the six PARSEC workloads we tested, three of them worked by simply linking to Hobbes-U rather than pthreads. Of the remaining, two workloads only required small modifications, usually to eliminate any data sharing on the stack, which our Hobbes-U implementation does not support. We only needed to insert an explicit checkout/checkin in one instance in order to correctly synchronize a flag variable that was not protected with pthread library calls.

These findings indicate that the Hobbes model at the very least is compatible with current memory shared memory management strategies. Combined with the extra benefits like easier debugging and determinism, we believe that Hobbes has potential even if the checkout/checkin operations aren't used in "new" ways.

6 Related Work

Hobbes has many similarities to recently proposed work on parallel programming models. In general, Hobbes is distinguished from these prior proposals by 1) the flexibility the Hobbes model allows, and 2) Hobbes' independence from any specific programming language.

Several recent programming models for determinism, including Revsions [9], Worlds [26], Cilk Hyperobjects [14], Determinator [2], and Grace [4], use revision control systems for inspiration. Each of these proposals uses a fork/join parallelism model where threads can only communicate directly with their parent. The Hobbes model, on the other hand, is not limited strictly to a fork/join style. Of course, the generality of Hobbes has drawbacks, such as the composability problem discussed in Section 3.1 that the proposals above can easily avoid.

Other work has modified the parallel programming model at the programming language level in order to achieve data isolation and/or determinism. Examples include X10 [11] and DPJ [7]. Hobbes, of course, is not tied to a particular programming language.

7 Future Work

From a fundamental standpoint, we still need to determine if the Hobbes model is the correct representation of the CVS model. There were several design choices made in the model described in Section 2 that could be revisited, such as the decision to limit the working copy's history.

Perhaps the area that needs most work before Hobbes has a chance of becoming a practical model is in the implementation. The model necessitates some amount of data replication and movement that would presumably lead to a host of implementation challenges. However, we believe that many of those challenges can be addressed by leveraging inherent replication already present, though hidden from the program, in modern hardware. For example, it may be possible to utilize the data replication in hardware caches to avoid much of the replication now delegated to software. We may also investigate model changes that may allow easier implementations, such as limiting allowable module sizes.

Another area ripe for investigation is conflict resolution policies. As stated above, we have currently only used a last-writer-wins policy. Many others with more interesting properties may exist, including thread priority schemes or manual resolution through callback functions.

8 Bibliography

1. Allen, M.D., Sridharan, S., and Sohi, G.S. Serialization sets: A dynamic dependence-based parallel execution model. *ACM SIGPLAN Notices 44*, 4 (2009), 85–96.
2. Aviram, A., Weng, S., Hu, S., and Ford, B. Efficient System-Enforced Deterministic Parallelism. *9th USENIX Symposium on Operating Systems Design and Implementation*.
3. Bergan, T., Anderson, O., Devietti, J., Ceze, L., and Grossman, D. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *SIGPLAN Not. 45*, 3 (2010), 53-64.
4. Berger, E.D., Yang, T., Liu, T., and Novark, G. Grace: safe multithreaded programming for C/C++. *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, (2009), 81–96.
5. Bienia, C. and Li, K. Parsec 2.0: A new benchmark suite for chipmultiprocessors. *MoBS, June*, (2009).
6. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices 30*, 8 (1995), 216.
7. Bocchino Jr, R.L., Adve, V.S., Dig, D., et al. A type and effect system for deterministic parallel Java. *ACM SIGPLAN Notices 44*, 10 (2009), 97–116.
8. Bressoud, T.C. and Schneider, F.B. Hypervisor-based Fault Tolerance. *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating Systems Principles*, (1995).
9. Burckhardt, S., Baldassin, A., and Leijen, D. Concurrent programming with revisions and isolation types. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, (2010).
10. Ceze, L., Tuck, J., Cascaval, C., and Torrellas, J. Bulk Disambiguation of Speculative Threads in Multiprocessors. .
11. Charles, P., Grothoff, C., Saraswat, V., et al. X10: an object-oriented approach to non-uniform cluster computing. *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (2005), 538.
12. Devietti, J., Lucia, B., Ceze, L., and Oskin, M. DMP: Determinisitc Shared Memory Multiprocessing. *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, (2009), 85--96.
13. Dunlap, G.W., King, S.T., Cinar, S., Basrai, M., and Chen, P.M. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, (2002), 211-224.
14. Frigo, M., Halpern, P., Leiserson, C.E., and Lewin-Berlin, S. Reducers and other Cilk++ hyperobjects. *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, (2009), 79–90.
15. Grune, D. Concurrent Versions System, A Method for Independent Cooperation. .
16. Hammond, L., Carlstrom, B.D., Wong, V., Chen, M., Kozyrakis, C., and Olukotun, K. Transactional Coherence and Consistency: Simplifying Parallel Hardware and Software. .
17. Hower, D., Dudnik, P., Hill, M.D., and Wood, D.A. Calvin: Deterministic or Not? Free Will to Choose. *The 17th IEEE International Symposium on High Performance Computer Architecture*, (2011).
18. Hower, D.R. and Hill, M.D. Rerun: Exploiting Episodes for Lightweight Race Recording. *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, (2008), 265-276.
19. Montesinos, P., Ceze, L., and Torrellas, J. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. .
20. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., and Wood, D.A. LogTM: Log-Based Transactional Memory. *Twelfth IEEE Symposium on High-Performance Computer Architecture*, (2006), 258-269.
21. Narayanasamy, S., Pereira, C., and Calder, B. Recording Shared Memory Dependencies Using Strata. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (2006), 229-240.
22. Narayanasamy, S., Pokam, G., and Calder, B. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. *Proceedings of the 32nd annual international symposium on Computer Architecture*, (2005), 284-295.
23. Protić, J., Tomašević, M., Tomasevic, M., and Milutinović, V. *Distributed shared memory: concepts and systems*. Wiley-IEEE Computer Society Pr, 1998.
24. Saha, B., Adl-Tabatabai, A., Hudson, R.L., Minh, C.C., and Hertzberg, B. McRT-STM: a High Performance Software Transactional Memory System for a Multi-Core Runtime. .
25. Thomson, A. and Abadi, D.J. The Case for Determinism in Database Systems. *Proceedings of the VLDB Endowment*, (2010).
26. Warth, A. and Kay, A. Worlds: Controlling the scope of side effects. *VPRI Research Note RN-2008-001, Viewpoints Research Institute*, (2008).
27. Xu, M., Bodik, R., and Hill, M.D. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. *Proceedings of the 30th annual international symposium on Computer architecture*, (2003), 122-133.