Colorama: Architectural Support for Data-Centric Synchronization *

Luis Ceze, **Pablo Montesinos**, **Christoph von Praun**† and **Josep Torrellas**

University of Illinois at Urbana-Champaign
{luisceze, pmontesi, torrellas}@cs.uiuc.edu
http://iacoma.cs.uiuc.edu

†IBM T.J. Watson Research Center
praun@us.ibm.com

ABSTRACT

With the advent of ubiquitous multi-core architectures, a major challenge is to simplify parallel programming. One way to tame one of the main sources of programming complexity, namely synchronization, is transactional memory (TM). However, we argue that TM does not go far enough, since the programmer still needs non-local reasoning to decide where to place transactions in the code. A significant improvement to the art is *Data-Centric* Synchronization (DCS), where the programmer uses local reasoning to assign synchronization constraints to data. Based on these, the system automatically infers critical sections and inserts synchronization operations.

This paper proposes novel architectural support to make DCS feasible, and describes its programming model and interface. The proposal, called *Colorama*, needs only modest hardware extensions, supports general-purpose, pointer-based languages such as C/C++ and, in our opinion, can substantially simplify the task of writing new parallel programs.

1. Introduction

As chip multiprocessors become widespread, there is growing pressure to substantially broaden their parallel application base. Unfortunately, the vast majority of current application programmers find parallel programming too complex. To effectively utilize the upcoming hardware, we need major breakthroughs that simplify parallel programming.

Developing a parallel application consists of four steps [15]: decomposing the problem, assigning the work to threads, orchestrating the threads, and mapping them to the machine. Orchestration is arguably the most challenging step, as it involves synchronizing the threads. It is in this area that innovations to simplify parallel programming are most urgently sought.

One such innovation is Transactional Memory (TM) [1, 7, 10, 16, 18]. In TM, the programmer specifies sequences of operations that should be executed atomically. TM simplifies parallel programming in two ways. First, the programmer does not need to worry about the intricacies of managing locks. Second, he does not need to fine-tune critical sections as much, since concurrency is only limited by dependences — not critical section length.

We claim, however, that TM is still complicated: it requires the programmer to reason *non-locally*. Specifically, when the programmer inserts a transaction annotation, he also needs to think about what other parts of the program may be accessing this same or related shared data, and potentially insert transaction annotations there as well. Intuitively, like inserting lock and unlock operations, inserting transaction annotations involves taking a *code-centric* approach.

To improve programmability further, we need a *data-centric* approach [20]. With *Data-Centric Synchronization* (DCS), the programmer associates synchronization constraints with the program's data structures. Such constraints indicate which sets of data structures should remain consistent with each other and, therefore, be accessed in the same critical section. From these constraints, the system automatically infers the critical sections and inserts thread synchronization operations in the code. DCS simplifies parallel programming because the programmer reasons *locally*, focusing only on what structures should be consistent with each other.

Existing DCS proposals [20] take user-provided, data-centric synchronization constraints and decide where to insert critical sections using software-only support. In particular, the compiler needs to analyze all the accesses in the code. This is unrealistic in most C/C++ environments, where pointer aliasing is common and, most importantly, dynamic linking denies the compiler access to the whole program.

To make DCS practical, this paper proposes the first design for Hardware DCS (H-DCS). Our proposal, called *Colorama*, relies on two hardware primitives: one that monitors all memory accesses to decide when to start a critical section, and one that flexibly triggers the exit of a critical section. Colorama is independent of the underlying synchronization mechanism. In this paper, we present a transaction-based implementation and also discuss the issues that appear in a lock-based implementation.

We describe Colorama's architecture, a simple implementation that extends a Mondrian Memory Protection (MMP) [22] system, its programming model and API, and its capacity to help debug conventional codes. We show that Colorama needs few hardware resources and has small overhead. It supports general-purpose, pointer-based languages such as C/C++ and, in our opinion, can substantially simplify the task of writing new parallel programs.

In the following, Section 2 introduces DCS; Sections 3, 4, 5 and 6 present Colorama's architecture, implementation, programming environment, and debugging issues respectively; Sections 7 and 8 evaluate Colorama; and Section 9 discusses related work.

2. Data-Centric Synchronization (DCS)

2.1. Basic Idea

In Data-Centric Synchronization (DCS) [20], the programmer associates synchronization constraints with data structures — typically

*This work was supported in part by the National Science Foundation under grants EIA-0072102, EIA-0103610, CHE-0121357, and CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; and gifts from IBM and Intel. Luis Ceze was supported by an IBM PhD Fellowship.

1-4244-0805-9/07/$25.00 ©2007 IEEE

when they are declared or allocated. These constraints specify which data structures are in the same "data consistency domain" and, therefore, should be kept consistent with each other. This means that when one structure is being modified, all the other structures in the same domain need to be protected from access by other threads. To support this model, when a thread accesses a structure of a domain, the thread automatically enters a critical section for that domain. No other thread can now access structures of that domain. When the thread finishes working on structures of that domain, the thread automatically exits the critical section.

DCS is in contrast to conventional Code-Centric Synchronization (CCS), where synchronization constraints are associated with code. In CCS, the programmer marks what code is inside which critical section.

We argue that DCS has a significant advantage over CCS in *programmability*. CCS requires the programmer to reason *non-locally* [20]: every time he inserts a transaction begin/end or a lock acquire/release annotation in the code, he also needs to think about what other locations in the program may be accessing this same or related data structures, and potentially insert synchronization annotations there as well. Instead, with DCS, the programmer reasons *locally*, focusing only on what data structures should be consistent with each other. The system automatically infers the critical sections.

The shortcoming of DCS stems from limited program knowledge. The system has to automatically infer when the code enters and exits a critical section, so that it can insert the appropriate synchronization operations around the section.

Identifying entry points to critical sections largely involves identifying accesses to data structures belonging to a domain. Identifying exit points is harder. It is typically impossible for the system to know when a thread has stopped working on structures of a given domain and, therefore, the critical section for that domain should terminate. Consequently, DCS schemes have an *Exit Policy*, which is a simple, clear algorithm for terminating a critical section. The exit policy used by the system is communicated to the programmer. This is because, to write correct code, the programmer *needs to know* the exit policy used, and write code in agreement with it. We believe that having a simple exit policy is an acceptable burden given the improvement in programmability provided by DCS.

2.2. Software DCS (S-DCS)

DCS has only been implemented in software, under limited environments. The main example of what we strictly consider Software DCS (S-DCS) is Vaziri *et al.*'s Atomic Sets [20]. This system includes a compiler and language extensions to Java. The programmer, when declaring Java classes, can group several fields into an Atomic Set. The elements of an Atomic Set are supposed to be manipulated atomically inside critical sections that are automatically created by the compiler.

The entry points of critical sections of an Atomic Set are inferred by the compiler by statically analyzing the code and identifying likely accesses to data belonging to the Set. Since Java is relatively analyzable due to type safety and the lack of pointer arithmetic, if the compiler has access to the whole program, then it can conservatively identify when data from Atomic Sets are accessed [20].

The exit policy used by Vaziri *et al.* is to insert the exit point of a critical section right before the return of the Java method that contains the corresponding entry point. This policy builds on the intuition that a method is a natural unit of work — a method is typically exited when the work is completed. Therefore, a single method includes both the entry and the exit points of a critical section.

2.3. Proposal for Hardware DCS (H-DCS): Colorama

S-DCS is unsuitable for popular languages such as C/C++, which allow pointer arithmetic and aliasing. Since the compiler cannot fully analyze the code due to lack of pointer information, it can only generate conservative critical section approximations of very limited use. Alternatively, if it inserts instructions to check the address of every pointer access dynamically, it induces intolerable overhead. More fundamentally, in environments with dynamic linking, deployment of S-DCS is impractical because the compiler may lack access to the whole program.

Therefore, this paper proposes a novel architecture to support DCS in hardware. The resulting *Hardware DCS (H-DCS)* scheme is called *Colorama*. It supports any type of access pattern, has low overhead, and is usable in any language.

Colorama has two primitives, corresponding to the need to identify critical section entry and exit points. The first one is hardware to monitor all addresses issued by the processor with very low overhead. If a thread accesses a structure belonging to a consistency domain from outside of a critical section for that domain, Colorama starts a critical section.

The second primitive is hardware to support the exit of a critical section. Such primitive is very flexible and is driven by the compiler, so that different exit policies can be supported. At all times, however, it has to be clear to the programmer what exit policy will be used by the compiler as it generates the executable. In this first paper, however, we simply use the exit policy used by Vaziri *et al.* [20]. We use it because it is very intuitive. For example, Wang and Stoller [21] use the heuristic that methods execute atomically to identify potential atomicity violations in Java programs.

Note that the support for Colorama does not replicate (and is largely independent of) the support that the machine provides for synchronization. In this paper, we propose a Colorama implementation that relies on transactions as the underlying synchronization mechanism. We also discuss the issues that appear in an implementation based on locks.

2.4. Examples of Colorama Programming

In Colorama, a data consistency domain is called a *Color*, while a memory region with structures belonging to a consistency domain is referred to as *Colored*. In this section, we show three motivating examples.

Linked List. Consider a linked list that is manipulated by functions that insert a node, delete a node, and traverse the list (Figure 1). The programmer can color all the nodes in the list with the same color. This is done with the *color* and *colorprop* system calls shown. *Color* takes a starting address, a size, and a color ID; it colors the address range with color ID. *Colorprop* takes a starting address, a size, and a colored address; it propagates the color of the colored address to the address range.

With Colorama's support, the list manipulation functions in the figure are written without any transaction or lock annotation. The result is code as simple as in a sequential program.

Task Queue. Consider a task queue where each entry points to a bucket of shared data (Figure 2). A thread accesses the task queue to retrieve a bucket. Then, the thread operates on the bucket. Finally, it

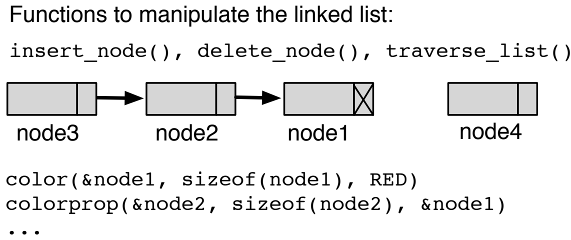


Figure 1. Example of linked-list manipulation.

accesses the task queue again to deposit new buckets. There are several variables associated with the task queue: head and tail pointers, a flag to check if the queue is empty, and a count of threads waiting on an empty task queue. The programmer can color the task queue, head, tail, empty and *num_waiters* structures with a single color, and each of the data buckets with a different color. Then, all the functions listed in the figure are written with no transaction or lock annotation.

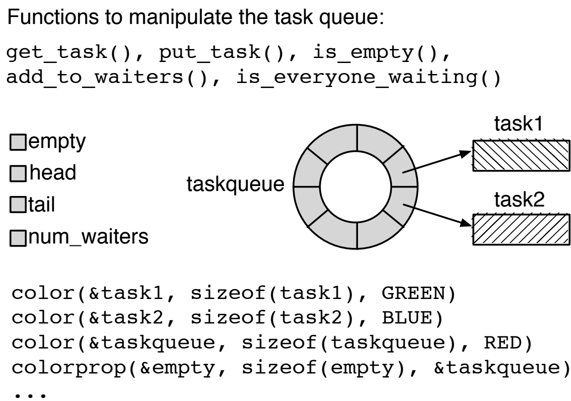


Figure 2. Example of task queue handling.

Sample MySQL Structure. Figure 3 shows a data structure from the MySQL database that is composed of many records. Each record has the *locks_in_mem* field and the *info* set of fields. A single global lock protects the *locks_in_mem* field in all records. Such lock is accessed from 29 sites in the MySQL code. Each record's *info* is protected by a per-record lock. Such lock is accessed from 14 sites. A Colorama programmer can color *locks_in_mem* in all records with the same color, and the per-record *info* fields with a per-record color. The records can now be accessed with no transaction or lock annotation.

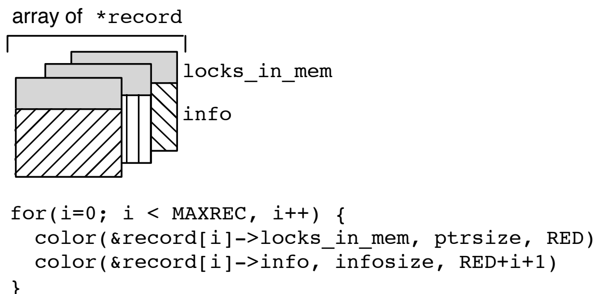


Figure 3. Sample structure from MySQL.

3. High-Level Architecture of Colorama

3.1. Overview

Colorama's architecture consists of a structure shared by all threads and a per-thread structure. The shared structure contains the current list of colored regions, while the per-thread one specifies what colors are currently owned by the thread. The per-thread structure also includes the mechanism to support the exit of a critical section.

At every load and store, Colorama leverages efficient hardware (Section 4) to check with very low overhead whether both the address is colored and the thread does not own the color. If so, Colorama triggers the entry to the color's critical section. Later, when certain events specified by the exit policy are detected, Colorama triggers the exit from the color's critical section.

The shared structure is called Color Map, or *Palette* (Figure 4). It is a software structure in shared memory that is partially cached in special hardware at each processor. The Palette lists, for each currently colored address region, the start and end addresses and its color (*ColorID*). Multiple address regions — and therefore multiple Palette entries — can have the same ColorID. However, a given address can only have a single ColorID and, therefore, appear in a single entry.

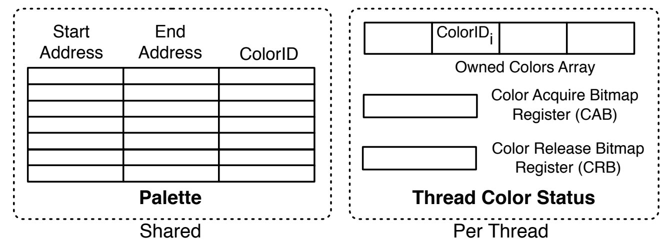


Figure 4. Architectural support for Colorama. While the Palette is conceptually a table, it has a hardware-software distributed implementation (Section 4.1).

The per-thread structure is the *Thread Color Status*. It contains the set of ColorIDs currently owned by the thread. These are the colors whose critical sections are currently being executed by the thread. They are listed in the Owned Colors Array.

The Thread Color Status also provides an efficient hardware primitive for the software to implement the exit policy. The primitive is built around the two Color Bitmap Registers: the read/write Color Acquire Bitmap (CAB) register and the write-only Color Release Bitmap (CRB) register (Figure 4). These registers have as many bits as entries in the Owned Colors Array (e.g., 64). Every time that a ColorID is inserted in location i of the Owned Colors Array, the corresponding bit in the CAB register is automatically set in hardware. In addition, when the software sets bit i of the CRB register, the hardware triggers a critical section exit for the ColorID in the corresponding entry of the Owned Colors Array.

3.2. Chosen Critical Section Exit Policy

As indicated in Section 2.3, in this paper we choose the exit policy used by Vaziri *et al.* [20]: trigger the exit of a color's critical section when the thread returns from the subroutine where the critical section was entered. We choose it because it is simple and intuitive: a subroutine is a natural unit of work; when the subroutine returns, the thread is likely to have finished the operation it was doing and, therefore, stopped working on that color's structures. Some evidence that programmers already follow this convention informally is presented

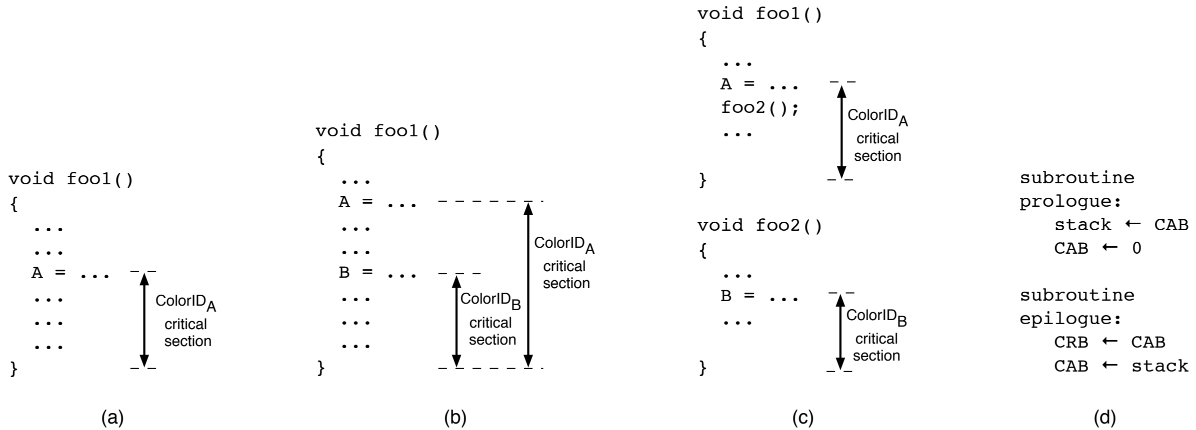


Figure 5. Illustration of the policy chosen in this paper to exit critical sections in Colorama and its implementation.

later (Section 8.1). Note, however, that in DCS, writing correct code *requires* that the programmer be aware of the exit policy supported by the system and follows it.

Figure 5 illustrates the policy. The figure assumes that variables *A* and *B* are colored with $ColorID_A$ and $ColorID_B$, respectively. Figure 5(a) shows an access to *A*, and how the resulting critical section runs until the end of the subroutine. Figures 5(b) and 5(c) show how critical sections nest. In both cases, a thread accesses *A* and, before it returns from the subroutine, it accesses *B*. As a result, the $ColorID_B$ critical section is nested inside the $ColorID_A$ one. The two figures, however, show different cases. In Figure 5(b), the accesses to *A* and *B* are in the same subroutine; as a result, both critical sections finish at the same time. In Figure 5(c), the accesses to *A* and *B* are in different subroutines, and the sections finish at different times.

This policy is implemented with the *compiler-inserted* instructions shown in Figure 5(d). At every subroutine entry, the compiler saves the CAB register in the stack and then clears it. This does not affect the Owned Colors Array (Figure 4). As the subroutine executes, if a *new* color becomes owned, the corresponding bit in the CAB register gets automatically set. Before the subroutine returns, the compiler copies the CAB to the CRB register, thereby triggering the exit of all the critical sections entered in this subroutine. Then, it restores the CAB register from the stack, leaving it in the state it had before the subroutine was called. This algorithm works with any nesting.

3.3. Detailed Colorama Operation

Based on the previous discussion, we now describe the operation of Colorama in detail. At every load and store, the cached Palette and the Thread Color Status are checked in hardware. If the address belongs to a colored region and the thread does not own that ColorID, a Colorama user-level software handler is automatically invoked with low overhead.

The handler adds ColorID to the Owned Colors Array. Then, if nested transactions are supported, the handler starts a new transaction for that color; if only flat transactions are supported, it starts a new transaction only if this is the only color owned by the thread. The handler then returns to the program. While these simple operations could be done in hardware, using a software handler is more flexible.

As per our exit policy, before every subroutine return, an instruction stores to the CRB register. For each set bit that gets written to the CRB register, if the same-offset entry in the Owned Colors Array has a valid ColorID, the hardware triggers a critical section exit for that ColorID.

A section exit for a set of ColorIDs starts with the automatic invocation of a Colorama user-level software handler. For each ColorID, the handler performs the following operations. First, the handler removes that ColorID from the Owned Colors Array. Then, if this was the last color in the structure, the handler initiates a transaction commit. If this was not the last color and the machine supports nested transactions, the handler initiates an inner-transaction commit for that ColorID. What an inner-transaction commit does is independent of Colorama. It could, for example, create a new checkpoint while keeping the thread speculative, in order to minimize the rollback distance in case of a collision. Finally, the handler returns.

When a transaction is squashed, its ColorID(s) are removed from the Owned Colors Array and its bit(s) in the CAB register are cleared.

3.4. Pointers as Subroutine Arguments

Sometimes, a critical section performs multiple operations on a structure, and invokes one subroutine per operation — passing as argument to each subroutine a pointer to the structure. This is common when handling complex structures such as hash tables. Figure 6(a) shows a lock-based example of a read and a write to a hash table. *htPtr* is a pointer to the hash table.

Figure 6(b) shows the corresponding Colorama code, where we assume that the hash table is colored. Colorama's hardware will detect accesses to the hash table only inside subroutines *readHash()* and *writeHash()*. As a result, it will create two separate critical sections, one inside each subroutine. This is not what the programmer intended.

Since we believe that this is a common style of programming, we would like Colorama to enclose the two subroutines inside a single critical section. Interestingly, Colorama would automatically do so if we accessed the hash table in subroutine *htUpdate()* before the call to *readHash()*: the exit policy would extend the critical section from that point till the end of *htUpdate()*.

To support this case, we extend Colorama with a primitive to potentially start a critical section. The mechanism is a new *colorcheck* instruction that performs a run-time address check. Colorcheck takes an address and checks whether it is colored and the color is not owned by the thread. If so, Colorama automatically triggers a criti-

```
void htUpdate()
{
 ...
 lock(L)
 value = readHash(htPtr,key)
 value++
 writeHash(htPtr, key, value)
 unlock(L)
 ...
}
```

(a) Lock-based code

```
void htUpdate()
{
 ...
 value = readHash(htPtr,key)
 value++
 writeHash(htPtr, key, value)
 ...
}
```

(b) Colorama code

```
void htUpdate()
{
 ...
 colorcheck htPtr                      - - -
 value = readHash(htPtr,key)             ^
 value++                                 |
 colorcheck htPtr                        | critical
 writeHash(htPtr, key, value)            | section
 ...                                     v
}                                      - - -
```

(c) Colorama code with colorcheck

Figure 6. Using the colorcheck instruction.

cal section entry as usual (Section 3.3). Colorcheck does not read or write the address, and cannot raise protection exceptions.

To use this primitive for our purposes, we extend the Colorama compiler to identify subroutine calls with arguments that are pointers. For every such argument, the compiler inserts a colorcheck instruction with that argument, right before the call — in the example, the argument is *htPtr*. The resulting code is shown in Figure 6(c). This change accomplishes what we need. At run time, colorcheck checks the contents of *htPtr* before *readHash()* and triggers the start of the critical section.

3.5. Why Use Multiple Colors

If the system supports nested transactions, having multiple colors provides an intuitive way to build transaction nests [17]: every time a new color is accessed inside a transaction, a new nesting level is created.

Irrespective of whether or not the system supports nested transactions, having multiple colors is also useful in three ways. First, it can help debug the code. Specifically, every time a processor attempts to commit a transaction, as it broadcasts the addresses that it wrote, we propose that it also broadcast the colors that the transaction owned. If a second processor that is executing a different-color transaction detects a collision with the committing one, the programmer is warned that a bug is likely — different-color transactions should not have collisions.

The second use is to help optimize the cross-thread dependence disambiguation that takes place at thread commit. If we are certain that the code has no bugs, we may decide to reduce overheads by not checking for collisions between concurrent transactions of different colors. This may save inter-processor traffic.

The final advantage of supporting multiple colors is that it enables the programmer to embed more information in the program on how shared data are used.

If the system uses locks, instead, supporting multiple colors directly translates into enabling more concurrency (Section 4.3).

4. Implementation of Colorama

4.1. Colorama Structures

The Colorama structures are the Palette and the Thread Color Status (Figure 4). The Palette is a distributed structure implemented part in hardware and part in software. It is accessed with a pattern similar to that of structures that contain address protection information — i.e., which address can be read or written by which thread. Indeed, protection information is also shared by all threads and is accessed at every memory request. Consequently, both types of information can share the same implementation. One difference is that the Palette contains per-word information, while current virtual memory systems associate protection information with pages. Consequently, to accommodate the Palette, we would need to redesign current TLB structures. In practice, there is already an efficient design that manages per-word protection information, namely the Mondrian Memory Protection (MMP) system [22]. Therefore, we implement the Palette as extra bits to be stored in the MMP structures.

The implementation of an MMP system is shown as the white structures of Figure 7(a). The Multilevel Permissions Table is a software table in shared memory that holds all the protection information. The table is hierarchically organized for space efficiency, with ranges of addresses expanded enough to keep the protection information at the available grain size (word, page, etc.). Processors transparently cache on demand sections of the table in a hardware buffer called Protection Lookaside Buffer (PLB). In addition, for faster access to protection information, architectural registers have sidecar registers, with recently-accessed protection information. Loads and stores automatically access the sidecars and PLB in hardware to check permissions. A PLB miss is like a TLB miss, and brings in the permissions transparently. OS-initiated PLB/sidecar updates propagate to memory and invalidate relevant entries in other processors's PLBs and sidecars.

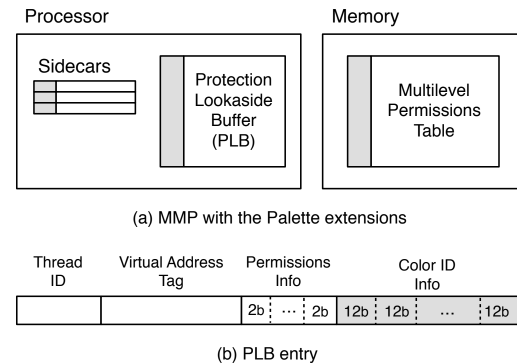


Figure 7. Implementation of the Palette on top of an MMP system. The shaded fields constitute the Palette.

The shaded fields in Figure 7(a) constitute the Palette. They simply add the ColorID bits to the three MMP structures. Figure 7(b) shows a PLB entry in detail. A PLB entry may correspond to a cache line. The Palette adds a ColorID (e.g., 12 bits) to every word contained in the PLB entry — e.g., 16×12 bits for a 16-word line. A load or store automatically checks the ColorID of the address accessed, which is typically in a sidecar register or in the PLB. When a thread changes the color of a range of addresses, the OS updates the PLB and the other structures as in the MMP system.

The Thread Color Status consists of three structures accessible in user mode: the read/write Owned Colors Array, the read/write CAB

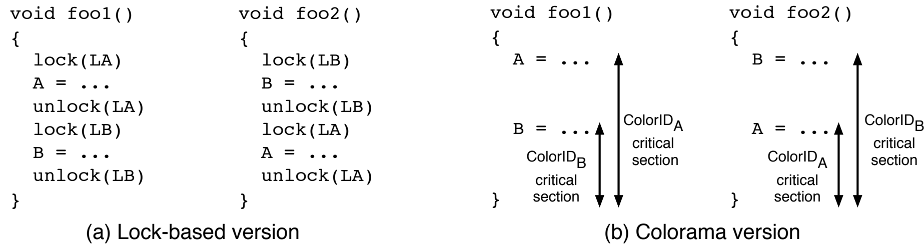


Figure 8. Example of how the chosen exit policy may cause a deadlock. Implementations with transactions do not have this problem.

register, and the write-only CRB register (Figure 4). They hold and manage the colors owned by the currently-running thread. These three structures all have the same number of entries (e.g., 64), although each entry is one bit in the registers and a ColorID in the Owned Colors Array. The Owned Colors Array and the CAB register are saved on a context switch. If a thread temporarily needs to own more colors than entries available, Colorama traps to software, which manages the extra state required.

The other key Colorama features are the colorcheck instruction, a related instruction called *getcolorid* (whose purpose is discussed later), and the low-overhead invocation of user-level handlers. The colorcheck and getcolorid instructions take an address. They are implemented like a load, in that the hardware accesses the sidecar, PLB entry, or the Permissions Table entry for the address. The getcolorid instruction simply returns the ColorID (if any) of the address. The colorcheck instruction, again like a load, if it finds that the address is colored and that the ColorID is not in the Owned Colors Array, it triggers a critical section entry. However, unlike a load, colorcheck stops right there, and does not access memory. Neither colorcheck nor getcolorid raises protection exceptions.

When a thread needs to enter or exit a critical section, the hardware invokes a Colorama user-level software handler. Using a software handler adds flexibility and simplicity, but it must be triggered with low overhead. Fortunately, the handler does not require any change in privilege mode. We can use support such as that of Informing Memory Operations [12].

The maximum number of colors supported is hardwired in several structures. While most programs need about 1K or fewer colors (Section 8.3), we size Colorama for a large number (4K). If the program needs more colors, Colorama hashes multiple colors into one. In this case, performance may be affected. Specifically, given the uses of multiple colors in a system that uses transactions (Section 3.5), we may end up combining two transactions that should be nested (and therefore squashing more work than necessary on a collision), potentially missing bug warnings, or generating more traffic than necessary to check for collisions.

4.2. Coloring at Page Granularity

An alternative implementation involves restricting color assignment such that all the structures in the same page share the same color. This policy can be enforced by specifying colors at memory allocation time and extending the memory allocator algorithm to keep pools of colored memory.

Such approach would need a simpler Palette implementation, since we could extend TLB and page table entries to include color information — the MMP system would not be needed. However, the resulting coloring support would be less flexible and possibly more complex for the software, since the color of the data structures would affect the memory layout.

4.3. Using Locks as the Underlying Synchronization Mechanism

This paper proposes an implementation of Colorama on a machine that uses transactions as the underlying synchronization mechanism. It is also possible to build Colorama on a system that uses locks. In this case, each distinct color is associated with a different implicit lock.

The Colorama user-level handler invoked at the entry point of a critical section, instead of starting a transaction, attempts to acquire the lock corresponding to the color. When it succeeds, it adds the ColorID to the Owned Colors Array and returns. Similarly, the handler invoked at the exit of a critical section releases the corresponding lock, removes the ColorID from the Owned Colors Array and returns. Note also that it is not possible to hash multiple colors into one because deadlocks may happen.

In a lock-based implementation, the specific exit policy that we have chosen in this paper may have two effects. The first one is that, since critical sections now run until the end of subroutines, they tend to have larger sizes and, therefore, may cause an increase in lock contention. In practice, we show in Section 8.2 that the average increase in critical section size is likely to be modest.

The second effect is that, depending on how the code is written, the exit policy chosen may cause deadlocks. As an example, Figure 8(a) shows a lock-based code and Figure 8(b) shows its corresponding Colorama code. In Figure 8(a), *foo1* acquires and releases lock *LA* and then acquires and releases lock *LB*, while *foo2* performs the same operations in opposite order. Suppose that, under Colorama, variables *A* and *B* have colors $ColorID_A$ and $ColorID_B$, respectively. Because of the exit policy, *foo1* will nest $ColorID_B$'s critical section inside $ColorID_A$'s, and *foo2* will do the opposite. If two threads executing *foo1* and *foo2*, respectively, perform the first assignment in *foo1* and *foo2* at the same time, they will deadlock.

This scenario must be rare in practice, since our experiments of Section 8.2 on conventional code have been unable to detect even a single instance of subroutine pairs that *could* deadlock in Colorama. Consequently, it may be acceptable to use this exit policy and, rather than trying to avoid deadlocks, detect them and break them if they occur. Alternatively, we can use a different exit policy that is not subject to this problem. We are currently working on this issue.

Deadlocks can be detected with a software table in memory that lists, for each color, the current owner thread and the spinning threads. When the Colorama user-level handler that attempts to acquire the lock for a color fails to do so, it registers its thread ID as spinning on the lock. It then checks for a cycle in owner and spinning thread IDs across multiple locks in the table. If it finds one, a

deadlock has occurred. Then, the handler informs the user of where the deadlock happened.

We consider this support to be a debugging aid. We expect that, as programmers become familiar with Colorama's programming model and whatever exit policy is used, they will write code that executes fast and reliably.

Note that deadlocks do not exist in a transaction-based implementation of Colorama. Transactions are known to be susceptible to livelocks, but they are easily avoided.

5. Programming with Colorama

The goal of Colorama is to simplify parallel programming. One of the ways in which Transactional Memory (TM) simplifies the programmer's job is by not requiring so much fine-tuning of the critical sections — concurrency is limited by dependences, not critical section length. With Colorama, the programmer's job is further simplified beyond TM because he does not even need to mark critical sections — the system automatically infers them. The result is highly programmable and maintainable code. In this section, we examine several programming issues in Colorama.

5.1. Correctness

At a minimum, Colorama guarantees that all executions of critical sections of the same color by different threads are serializable. Consequently, if the programmer colors all the shared data structures that should be accessed in an exclusive manner, Colorama produces a data-race free program. All conflicting accesses will be separated by transaction boundaries or lock operations.

The extent and granularity of coloring typically matter relatively little in a transaction-based implementation of Colorama, since concurrency is only limited by data dependences — although long transactions with resulting cache overflow are slow. However, they matter substantially more in a lock-based implementation. In this case, if the programmer colors structures for which the accesses do not need to be constrained (e.g., thread-private variables), the resulting superfluous critical sections or longer-than-necessary ones may limit concurrency and lower performance. Conversely, a programmer can enable more concurrency if variables that do not have mutual consistency constraints are assigned different colors. This may improve performance.

If the programmer fails to color a structure that should be accessed in an exclusive manner, the program may have data races. Likewise, if he assigns different colors to structures that have mutual consistency constraints, or if he does not respect the exit policy of the system — in our case, by continuing to manipulate an exclusive structure past the corresponding subroutine return — the program may function incorrectly.

5.2. Code Compatibility Issues

A program written for Colorama may be linked with libraries that do not use Colorama's Application Binary Interface (ABI) — for example, they use explicit transactions or locks. In this case, no special action needs to be taken. The legacy library will use transactions or locks to protect its own data structures, not program data. For library-accessed program data, Colorama will continue to trigger critical section entries on access and (if the library executes program code through a callback) critical section exits on subroutine returns.

In certain exceptional cases, applications may require the absence of Colorama's default exit policy. For example, consider an infinite loop where a consumer thread reads data from a shared buffer that is filled by a producer thread. If programmed with transactions, every access to the buffer would be a transaction. In Colorama, if the shared buffer is colored, the whole infinite loop would become a single critical section. To avoid this case, the programmer (or compiler) has to explicitly release the buffer's color at every iteration. As another example, to implement a wait on condition variables, the programmer (or compiler) will want to be able to temporarily release a color and then re-acquire it.

These operations are available through a Colorama library as follows. First, consider releasing the color associated with an address. The library first uses a Colorama instruction called *getcolorid* (Section 4.1). Such instruction simply returns the ColorID of the address. Then, the library searches the Owned Colors Array (Figure 4) to find the array offset where that ColorID is stored. If found, the library writes to the CRB register a set bit at the same offset, which triggers the release of ColorID. Note also that we can release all colors by writing all ones to the CRB register.

Releasing a color *temporarily* involves releasing the color as before and saving the address. Re-acquiring a color involves using the colorcheck instruction on the saved address.

5.3. Colorama's Complete API

Colorama's complete API is shown in Table 1. It contains five instructions, three system calls, and four library calls. The instructions are colorcheck, getcolorid, and moves to/from CAB or CRB. The system calls color or decolor addresses. The reason why these operations are system calls is that they update the PLB, which also contains protection information (Section 4.1). These system calls are typically issued when data structures are allocated or deallocated — they are rarely issued otherwise. Possibly, the two coloring system calls could be inserted directly by the compiler, based on language syntax extensions that specify colors when data structures are declared. Moreover, the decolor system call could be inside *free()*. Finally, the rationale for the four library calls in Table 1 was presented in Section 5.2. Typically, only experienced programmers would use the library calls.

5.4. Example: Prevention of an Atomicity Violation

Finally, to showcase the advantages of Colorama's programming simplicity, we show one example where Colorama helps prevent a subtle synchronization defect. Figure 9 shows Java method *append*, which appends one string to another. It calls methods *length* to get the length of a string and *getChars* to copy the string. The figure also shows a call to append string *sb* to string *sa*.

Method *append* is annotated as *synchronized*, which means that it executes under mutual exclusion with other *synchronized* methods invoked on *sa*. Methods *length* and *getChars* are also *synchronized*. However, when they are called from within *append* in the example, they are *synchronized* with other methods invoked on *sb*. As a result, although the individual interactions of *length* and *getChars* on *sb* are atomic, the sequence of interactions is not: it can happen that string *sb* is altered by another thread in-between the *length* and *getChars* calls — resulting in a stale value of *len* at the point of calling *getChars*.

In Colorama, defects such as this one are prevented. If string *sb* is colored, as soon as it is first accessed inside *append*, a critical section starts. With the exit policy used, the critical section extends to the end of the method — therefore encompassing the calls to *length*

Instructions (Typically inserted by the compiler)	
`colorcheck Addr`	Check if (Addr is colored and its color is not owned by the thread). If true, enter critical section
`getcolorid Addr, reg`	Save the ColorID of Addr in a register
`mov reg, CAB`	Update the CAB register
`mov CAB, reg`	Read the CAB register
`mov reg, CRB`	Update the CRB register
System Calls (They change the Palette. Inserted by the programmer or the compiler)	
`color (StartAddr, Size, ColorID)`	Color this address range with ColorID
`colorprop(StartAddr,Size,ColoredAddr)`	Propagate the color of ColoredAddr to this address range
`decolor (Addr)`	Remove the color from the structure at Addr
Library Calls (They change the Thread Color Status. Used in exceptional circumstances)	
`color_release ()`	Thread releases ownership of all its colors
`color_release (Addr)`	Thread releases ownership of the color of the structure at Addr
`color_temp_release (Addr)`	Thread *temporarily* releases ownership of the color of the structure at Addr
`color_reacquire ()`	Thread re-acquires ownership of all the colors that it temporarily released

Table 1. Colorama's complete API.

```
class StringBuffer {
   public synchronized StringBuffer append(StringBuffer sb)
   {
     ...
     int len = sb.length();
     ...
     sb.getChars(len,...); // len may be stale
     ...
   }
   public synchronized int length() { ... }
   public synchronized void getChars(...) { ... }
}

StringBuffer sa;
StringBuffer sb;
...
sa.append(sb);
```

Figure 9. Example where Colorama prevents an atomicity violation.

and *getChars* and avoiding the problem. No code annotations are necessary beyond coloring. Also, note that, if *sb* is not shared, we avoid any synchronization overhead by simply not coloring it.

6. Code Debugging Issues

While we argue that programming in Colorama is simpler and less error-prone than in the conventional CCS approach, it is still possible to have bugs. In this section, we examine how to debug Colorama code. In addition, we also consider a related question, namely leveraging the Colorama hardware to debug conventional CCS code.

6.1. Debugging Colorama Code

We classify Colorama bugs into three classes: (i) failing to color a structure that should be colored; (ii) coloring two structures from the same consistency domain with two different colors; and (iii) violating the exit policy. The bugs in class (i) can lead to data races, which can be detected with conventional data-race detection tools. They can also lead to collisions between critical sections of different colors, which are easily detected by Colorama (Section 3.5).

The bugs in classes (ii) and (iii) cause atomicity violations. They can be debugged with conventional tools that use heuristics to detect atomicity violations [6, 21].

The bugs in class (iii) are unique to DCS. For the exit policy used in this paper, they occur when the programmer assumes that a critical section extends past its corresponding subroutine return. The exit policy, of course, triggers a critical section exit at that particular return. Fortunately, we can use simple heuristics to identify possible instances of these bugs. The procedure is to record the colors of the critical sections that exit at a given subroutine return *i*. Then, we check if the thread accesses any of these colors again before the next *N* dynamic subroutine returns — where *N* can be 1. If it does, the programmer is warned, as he may have expected that the color's critical section had extended beyond the return *i*. Note that this procedure only relies on single-thread information — not on information dependent on the access interleaving of multiple threads. As a result, the bug manifests deterministically.

6.2. Debugging CCS Code with Colorama Hardware

A programmer who writes conventional CCS code on a machine with Colorama hardware can benefit from additionally annotating the data structures with colors as in DCS. Such annotations, if they drive the Colorama hardware without actually starting critical sections, can help debug the CCS code. As an illustration, assume that the programmer has written the CCS code with transactions. In this case, the Colorama hardware can detect when the following rules are violated, which is a strong indication of a bug.

1. Colored data should only be accessed inside transactions. Accesses from outside are typically bugs.

2. As indicated in Section 3.5, transactions of different colors should not collide. The Colorama hardware records the colors accessed by each transaction. A collision between two transactions of different colors likely suggests that the programmer was unaware of some data sharing.

3. A non-nested transaction should typically access only one color. If a transaction accesses multiple colors, there may be an opportunity for transaction nesting that could be flagged to the programmer. More than a bug, this is possibly a missed optimization opportunity.

4. A subroutine should not typically contain two transactions of the same color. As pointed out in [21], functions that manipulate shared data in parallel programs are often intended to be atomic. Therefore, having two transactions of the same color in the same subroutine rather than one may be a bug.

7. Experimental Setup

Since there are no programs written for Colorama, our evaluation consists of analyzing existing lock-based applications and estimating Colorama's potential and overheads. We analyze a variety of large, open-source, realistic multithreaded applications written in C

or C++. Among them are the AOL web server, the Firefox web browser, the MySQL database server, and others. Table 2 lists the applications along with their number of dynamic instructions, critical sections (static and dynamic) and peak memory footprint, as they run natively on a Xeon-based multiprocessor with 8 hardware contexts.

Name	Description	# Inst (10^9)	# Critical Sec		Peak Footp (MB)
			Sta	Dyn (10^3)	
aolserver	Web server (v4.0.10)	19.5	116	1169.4	11.2
barnes	SPLASH-2 application	11.8	22	69.1	34.8
firefox	Browser (v1.5.0.1)	7.1	485	832.8	172.2
gaim	Instant msg (v2.0.0b2)	3.2	6	9.9	138.5
gftp	FTP client (v2.0.18)	1.4	173	882.0	52.9
mysql	MySQL DB (v5.0.18)	32.7	147	3302.7	545.5
tuxracer	Game (v0.5a)	10.5	74	15.7	91.7
Avg	—	12.3	146.1	897.4	149.5

Table 2. Multithreaded applications evaluated.

We developed a Pin-based [14] tool that profiles our applications running natively with multiple threads. The tool tracks synchronization operations and collects information such as lock acquire and release sites, lock addresses, and critical section executions and sizes. It also collects other events such as instruction counts and memory allocations and deallocations. The tool is also connected to a simulator that models a Multilevel Permissions Table for MMP [22] with Palette extensions (Figure 7(a)).

Synchronization operations are typically calls to multithreading libraries such as Pthreads. Many times, however, applications synchronize with indirections to pthread functions or with actual application code. An example is Tcl_MutexLock and Tcl_MutexUnlock, part of the TCL library used by *aolserver*. Our profiler can handle such cases as well.

8. Evaluation

We evaluate the suitability and impact of our chosen Colorama exit policy, and then examine Colorama's structure sizes and overheads.

8.1. Suitability of Colorama's Exit Policy

This section presents experimental evidence showing that the exit policy that we choose for Colorama in this paper is already an informal convention largely followed by programmers of CCS code. Consequently, requiring its compliance for correct DCS code would likely be a light burden. For this experiment, we determine, for each critical section executed by the applications, whether the lock acquire and release are in the same subroutine. If they are, the section is *matched*; otherwise, it is *unmatched*.

Figure 10 shows the percentage of dynamic (D) and static (S) critical sections that are matched or unmatched. Recall from Table 2 that individual applications have 10K-3303K dynamic critical sections and 6-485 static ones. From the figure, we see that matched critical sections account for practically all the dynamic sections, and for 95% of the static ones. This supports our choice of exit policy. It shows that programmers already tend to initiate and conclude a critical section in the same subroutine.

The few unmatched cases are either special cases or are in code that is very fine-tuned for concurrency, especially in libraries. For example, in *firefox*, *gaim*, and *gftp*, all unmatched critical sections are inside the fine-tuned GTK library.

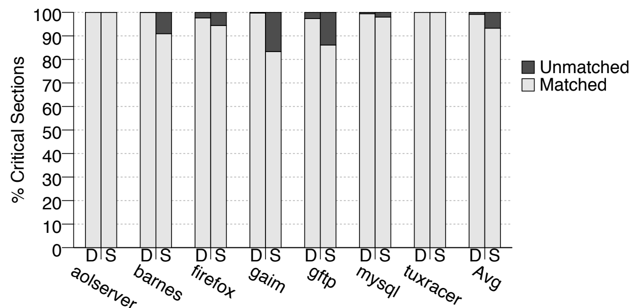


Figure 10. Percentage of dynamic (D) and static (S) critical sections that are matched or unmatched.

Figure 11 shows a representative unmatched critical section from GTK. In the figure, subroutine *g_main_dispatch* assumes that it holds lock *context*. Inside the subroutine, before the invocation of callback function *dispatch*, the code releases the lock; after the invocation, the code acquires the lock back. This structure would not be compatible with our exit policy. In this particular case, however, Colorama can handle this code without any changes because it is library code (Section 5.2).

```
/* thread holds "context" lock */
g_main_dispatch (GMainContext *context)
{
  ...
  UNLOCK (context);
  ...
  need_destroy=!dispatch(src,callback,usr_data);
  ...
  LOCK (context);
  ...
}
```

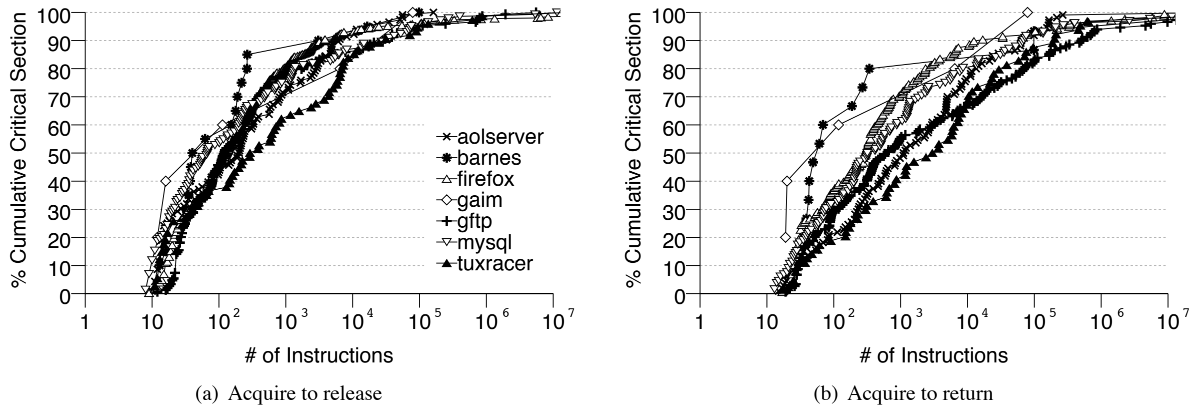
Figure 11. Example of code from the GTK library with an unmatched critical section.

8.2. Impact of Colorama's Exit Policy

The exit policy that we have chosen has two potential implications: the critical section size increases and independent critical sections may get combined in a nest. These issues typically have little or no impact in our proposed transaction-based implementation of Colorama. However, in a lock-based implementation, the first issue could increase lock contention and the second one could, under certain conditions, cause deadlock (Section 4.3).

To assess the first issue, we measure the average *dynamic* size of each critical section in its lock version (from acquire to release) and in what would be its Colorama version (from acquire to subroutine return). The resulting cumulative distribution is shown in Figures 12(a) and (b), respectively.

While the dynamic sizes of critical sections do increase, the average increase is not excessive. In some applications, there are a few critical sections that increase in size substantially. For example, this occurs for the sound thread in *tuxracer*. The thread acquires and releases a lock at the beginning of the game, and then runs for the duration of the game without returning from the subroutine. However, we believe that, since the Colorama programmer is required to know the system's exit policy, he will write the code to avoid lengthy critical sections.



(a) Acquire to release

(b) Acquire to return

Figure 12. Cumulative distribution of dynamic critical section size from acquire to release (a) and from acquire to subroutine return (b).

To assess the case of independent critical sections being combined into a nest of critical sections, we measure how often multiple, independent critical sections have their entry points inside the same subroutine. These are the ones that would be combined into a nest. Figure 13 shows the percentage of dynamic (D) and static (S) critical sections that, because of Colorama's exit policy, would end up combining with an independent second critical section, by nesting it inside. Such instances are called *Combined*.

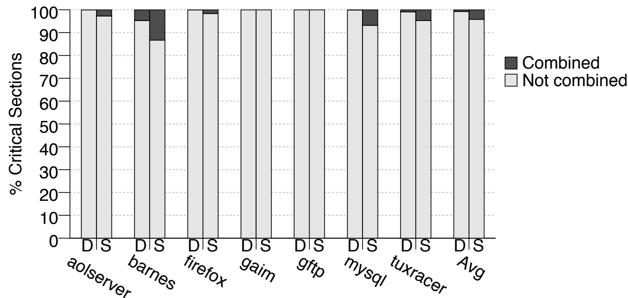


Figure 13. Percentage of dynamic (D) or static (S) critical sections that end up nesting a second critical section inside them.

From the figure, we see that on average only about 1% of the dynamic critical sections and 4% of the static ones end up nesting a second critical section in. A detailed analysis of these (few) cases shows that the resulting order of any pair of nested locks is always the same — which eliminates the possibility of getting a deadlock. Consequently, we conjecture that the possibility of deadlock will be rare.

8.3. Colorama Structure Sizes

To estimate the sizes of the Colorama structures in Figure 4, we perform several measurements on the applications. We conservatively assume that every time an application allocates or deallocates memory, it adds or deletes, respectively, a colored region. Consequently, the number of "live" allocated regions plus the number of static data objects in the binary gives the total number of colored regions at a time. This number is shown in Column 2 of Table 3, and corresponds to the number of rows in the Palette. Such number ranges from 100 to nearly 1M.

App	Colorama Structure Sizes				Colorama Overheads	
	# of Palette Rows (10^3)	# of Colors	# of ColorID Bits	# of OCA Entries	# of Subr Calls (% Inst)	# of Inst per Col Syscall (10^3)
aolserver	0.6	141	8	39	1.9	28346.8
barnes	0.1	155	8	15	0.7	287775.4
firefox	960.1	3992	12	11	1.2	3.6
gaim	743.0	1151	11	4	1.9	1.9
gftp	15.2	874	10	6	2.5	1.9
mysql	40.7	1936	11	10	2.7	129.5
tuxracer	10.3	73	7	6	0.3	160.6
Choice	—	4096	12	64	—	—

Table 3. Characterization of Colorama.

We also measure the number of distinct lock addresses in each program. Such number estimates the number of different colors needed. The number is shown in Column 3. We see that programs need 100-4000 colors. From this number, we compute the number of bits in ColorID. As shown in Column 4, we need 7-12 bits in the ColorID field.

Finally, to determine the number of entries in the Owned Colors Array (OCA) in Figure 4 (or the number of bits in the CAB and CRB registers), we need to measure the maximum number of locks held by a thread at a time. To be conservative, we measure the maximum number of locks held at a time by *all* threads combined. Such number is shown in Column 5, and ranges from 4 to 39.

The last row of Table 3 shows the parameters we choose for Colorama: 4K colors, 12-bit ColorIDs, and a maximum of 64 owned colors per thread. Moreover, following [22], we set the PLB to 128 entries, where each entry maps 16 words.

8.4. Colorama Overheads

Finally, we measure the two main Colorama overheads, namely additional instructions and additional memory space. One more overhead is the extra references to memory due to PLB misses, but these are largely the same as in the base MMP design (without Colorama) — around 8%, as quantified in [22].

Additional Instructions. For every subroutine invoked, the compiler inserts about six instructions to perform the operations shown in Figure 5(d). In addition, for each pointer that the subroutine takes

as argument, the compiler adds one colorcheck instruction. Overall, we could assume that, on average, Colorama adds about seven instructions per subroutine invocation. As a reference, Column 6 of Table 3 shows that, on average, about 1.6% of the dynamic instructions are subroutine calls.

In reality, the resulting overhead is likely to be very small. First, the added instructions are mostly register moves and loads/stores that hit in the cache — since they access the stack; they can easily fill the many unused execution slots in superscalars. Moreover, the compiler does not need to add these additional instructions for the subroutines that it can prove do not access colored data. Finally, applications often execute library code, which is not subject to this overhead.

A second source of overhead is the execution of the user-level handlers to enter and exit critical sections. However, the contribution of these instructions is very small, given the low frequency of critical section entry and exit. Such frequency is given by two times the numbers in Column 5 of Table 2 over the numbers in Column 3 of the same table.

Finally, Colorama also executes coloring system calls. We conservatively assume that every time the application allocates or deallocates memory, it issues one such call to add or delete a colored region, respectively. Column 7 of Table 3 shows the frequency of these system calls. For four applications, they are issued on average only once every 129K-288M instructions. In this case, the overhead is negligible. In three other applications, they are issued once every 2K-4K instructions. In these applications, the frequent memory allocation/deallocation is already very costly in itself. We can eliminate most of the additional cost of coloring by having the memory allocator keep pools of colored memory. As a result, there is no need to issue a system call at each of these operations.

Additional Memory Space. The large majority of Colorama's memory overhead is due to the Palette. To compute the Palette's overhead, we model in detail the MMP's Multilevel Permissions Table of Figure 7(a) in our simulator. We use the Mini-SST format of the entries, as suggested in [22]. We measure two memory space overheads: the one for the base MMP with permissions information (white part of the Permissions Table in Figure 7(a)), and the one for the Palette state only (shaded part in Figure 7(a)). Figure 14 shows these two memory space overheads as a fraction of the application footprint. For a given bar, both these two overheads and the application footprint are the peak values for the whole application execution. For additional information, the figure models ColorID fields that range from 8 to 32 bits.

The figure shows that the Palette adds only a very modest overhead over that of the base MMP. On average, for the range of ColorIDs used, the Palette only adds 1-2.5% more space to the footprint of the application.

9. Related Work

Section 2.2 described the work that we strictly consider S-DCS, and how it differs from Colorama. To that discussion, we add that Atomic Sets [20] are what we call colors, and that Vaziri *et al.* also allow the programmer to explicitly associate external methods to an Atomic Set, which arguably breaks the pure data-centric approach.

Other systems that support a less flexible form of DCS are languages [2, 3, 8] with concurrency control based on Monitors [11]. In such languages, it is possible to specify a shared data structure and the set of procedures that are allowed to access it. The compiler will then add the necessary synchronization operations to make these procedures execute in a mutually exclusive way. The key difference is that, in Colorama, the programmer does not have to specify the procedures that touch the shared data structure. Synchronization is inferred dynamically by the hardware — an approach that is efficient, flexible, and often the only alternative when the code is hard to analyze statically or simply not available to the compiler.

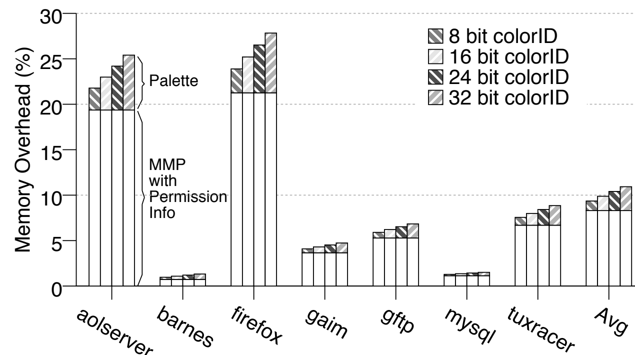


Figure 14. Space overhead of the base MMP and of the Palette for different ColorID sizes.

Several works have associated data objects to synchronization information for a variety of purposes. For example, in Entry Consistency (EC) [4], the association is done to enforce memory consistency in a distributed shared-memory system. The programmer explicitly associates shared locations with locks. When a processor enters a critical section by acquiring a lock, the associated shared locations are made consistent. An important difference with Colorama is that in EC, the programmer explicitly marks the critical sections in the code. This makes EC code-centric, with some data-centric annotations.

Having to explicitly list the shared data associated with a critical section is a burden to the programmer. As a result, Scope Consistency [13] improves on EC by having the software system automatically infer the shared data accessed in the scope of each critical section. Still, the programmer has to mark the critical sections.

Like Colorama, Xu *et al.* [23] try to infer critical sections, although the approach and environment is very different. They examine a post-mortem trace of memory references after a bug has been detected, and propose heuristics to infer the code that should be in critical sections. They use this information to estimate if a synchronization was missing. The Colorama hardware cannot directly use their heuristics to decide when to enter/exit a critical section because their scheme requires access to future references and to references from other processors. Moreover, their heuristics can have false positives and false negatives. However, their scheme could be usable in other DCS designs.

Other related works include: (i) programmer-specified association between code and data for static or dynamic validation of parallel programs (e.g., [19]); (ii) programmer-specified "transactional" variables in composable memory transactions [9] that provide stronger atomicity guarantees; and (iii) the lock bits associated with memory regions in the IBM 801 [5], used to support transactions on memory-mapped I/O.

10. Conclusions and Future Work

To reduce the complexity of parallel programming, this paper has proposed *Colorama*, the first design of Hardware DCS (H-DCS).

Colorama relies on two nimble hardware primitives to make DCS practical: one that monitors all memory accesses and one that can flexibly trigger the exit of a critical section based on a mechanism programmed in software. We have described Colorama's operation with transactions as the underlying synchronization mechanism. Moreover, we have presented Colorama's simple implementation based on MMP, its programming model and API, and its capacity to help debug conventional CCS codes. Finally, we have discussed the issues that appear in a lock-based implementation.

The evaluation assessed the policy chosen in this paper to exit a critical section at the return from the subroutine where the critical section was entered. We showed that this exit policy is already an informal convention largely followed by programmers of CCS code. Consequently, requiring its compliance for correct DCS code will likely be a light burden at most. We also showed that the policy increases critical sections modestly on average, and rarely combines critical sections — issues largely relevant to a lock-based implementation. The evaluation also showed that, by building on top of an MMP system, Colorama requires only modest hardware resources and induces small overheads.

Overall, Colorama effectively supports general-purpose, pointer-based languages such as C/C++ and, in our opinion, can substantially simplify writing *new* parallel programs beyond transactions. Our future work involves: (i) developing and evaluating new exit policies, (ii) writing and evaluating large Colorama programs and (iii) combining S-DCS and H-DCS into a hybrid system.

Acknowledgments

We thank the anonymous reviewers and the members of the I-ACOMA group at the University of Illinois for their invaluable comments. Special thanks go to Karin Strauss, Paul Sack, Brian Greskamp, Calin Cascaval and Mark Oskin for their feedback on the paper.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *International Symposium on High Performance Computer Architecture*, February 2005.

[2] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, vol. 18, no. 3, 1992.

[3] J. Barnes, "Introducing Ada 9X," *ACM Ada Letters*, 1993.

[4] B. Bershad, M. Zekauskas, and W. Sawdon, "The Midway Distributed Shared Memory System," in *IEEE Int'l Computer Conference (COMPCON)*, February 1993.

[5] A. Chang and M. F. Mergen, "801 Storage: Architecture and Programming," *ACM Transactions Computer Systems*, vol. 6, no. 1, 1988.

[6] C. Flanagan and S. Qadeer, "A Type and Effect System for Atomicity," in *Conference on Programming Language Design and Implementation*, June 2003.

[7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *International Symposium on Computer Architecture*, June 2004.

[8] P. B. Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, 1975.

[9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable Memory Transactions," in *International Symposium on Principles and Practice of Parallel Programming*, June 2005.

[10] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *International Symposium on Computer Architecture*, 1993.

[11] C. Hoare, "Monitors - An Operating System Structuring Concept," *Communications of ACM*, vol. 17(10), 1974.

[12] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors," in *International Symposium on Computer Architecture*, June 1996.

[13] L. Iftode, J. Singh, and K. Li, "Scope Consistency: A Bridge between Release Consistency and Entry Consistency," in *Symposium on Parallel Algorithms and Architectures*, June 1996.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Conference on Programming Language Design and Implementation*, June 2005.

[15] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*. Addison Wesley, 2005.

[16] K. Moore, J. Bobba, M. J. Moravan, M. Hill, and D. Wood, "LogTM: Log-Based Transactional Memory," in *International Symposium on High Performance Computer Architecture*, February 2006.

[17] E. Moss and T. Hosking, "Nested Transactional Memory: Model and Preliminary Architecture Sketches," in *Workshop on Synchronization and Concurrency in Object-Oriented Languages*, October 2005.

[18] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," in *International Symposium on Computer Architecture*, June 2005.

[19] D. F. Sutherland, A. Greenhouse, and W. L. Scherlis, "The Code of Many Colors: Relating Threads to Code and Shared State," in *Workshop on Program Analysis for Software Tools and Engineering*, November 2002.

[20] M. Vaziri, F. Tip, and J. Dolby, "Associating Synchronization Constraints with Data in an Object-Oriented Language," in *Symposium on Principles of Programming Languages*, February 2006.

[21] L. Wang and S. D. Stoller, "Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs," in *International Symposium on Principles and Practice of Parallel Programming*, March 2006.

[22] E. Witchel, J. Cates, and K. Asanović, "Mondrian Memory Protection," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[23] M. Xu, R. Bodik, and M. D. Hill, "A Serializability Violation Detector for Shared-Memory Server Programs," in *Conference on Programming Language Design and Implementation*, June 2005.