Programming and Debugging Shared Memory Programs with Data Coloring

Luis Ceze[†], Christoph von Praun[‡], Călin Caşcaval[‡]
Pablo Montesinos[#] and Josep Torrellas[#]

[†] Department of Computer Science and Engineering, [‡] IBM T.J. Watson Research Center
University of Washington {cascaval}@us.ibm.com
luisceze@cs.uiuc.edu

[#] Department of Computer Science,
University of Illinois at Urbana-Champaign
{pmontesi, torrella}@cs.uiuc.edu

Abstract. Concurrency control is one of the main sources of error and complexity in shared memory parallel programming. While there are several techniques to handle concurrency control such as locks and transactional memory, simplifying concurrency control has proved elusive.
In this paper we introduce the *Data Coloring* programming model, based on the principles of our previous work on architecture support for data-centric synchronization. The main idea is to group data structures into consistency domains and mark places in the control flow where data should be consistent. Based on these annotations, the system dynamically infers transaction boundaries. An important aspect of data coloring is that the occurrence of a synchronization defect is typically determinate and leads to a violation of liveness rather than to a safety violation. Finally, this paper includes empirical data that shows that most of the critical sections in large applications are used in a data-centric manner.

1 Introduction

While critical sections are probably the most popular form of concurrency control in shared memory programs, their use is a complex and error-prone task, especially for programmers with little experience in parallel programming. It can be argued that a major reason why critical sections are error-prone is that they require *non-local reasoning* [1]: shared data structures that need to be accessed atomically must be protected in *all* code locations where they may be referenced. Failure to do so may result in data races that are often hard to detect and debug. This problem is independent of the underlying critical section implementation, whether based on locks or transactions. Based on empirical evidence from large

[0] This work was supported by the National Science Foundation under grants CCR-0325603 and CNS-0720593.
Christoph von Praun is now affiliated with Georg-Simon-Ohm University, Nuremberg, Germany.

applications, we recognize the fact that the main reason for concurrency control is to protect shared data. Therefore, we believe that programmers should annotate the *data* that must be kept consistent.

In this paper we introduce the *Data Coloring* programming model, which is based on the principles of our previous work on architecture support for data-centric synchronization [2]. In this model, programmers associate consistency constraints (called *Colors*) with shared data structures. This is a data-centric approach to synchronization, as opposed to the conventional operation-centric approach. We argue that data-centric synchronization simplifies concurrency control because it mostly needs local reasoning – the programmer annotates the data structures with consistency constraints without worrying about where in the code the structures are being accessed.

In our programming model, in addition to assigning colors to data, the programmer also marks places in the control flow where data should be made consistent (*Color Steps*). Based on all these annotations, the system dynamically infers transaction boundaries. Finally, to support situations where largely-unrelated data structures need to be consistently operated on by a set of actions, we also support operation-centric synchronization with explicit transactions.

2 Consistency and Concurrency Control

Concurrency control prevents threads from performing concurrent, conflicting accesses to shared data, thus maintaining data consistency. *Consistency* defines how a thread observes the state of a *set of memory locations* with respect to updates by other threads.

It is common practice —and a widely accepted model of shared memory parallel programming— that concurrency control is specified explicitly. For example, programmers delimit critical sections, implemented by locks or transactions. Consequently, data consistency is a result of the synchronization structure, rather than something explicitly defined. As a result, data consistency can be easily compromised, either by the absence or the wrong use of synchronization annotations.

Data-centric synchronization [2, 1] turns this conventional model upside down: consistency constraints are tied to the shared data and specified explicitly; the system dynamically synthesizes actions for concurrency control according to the consistency specifications. Hence, data consistency is guaranteed as specified by the programmer. In the following, we define consistency in the context of data-centric synchronization, and elaborate on how actions for concurrency control are synthesized from the memory access stream in each thread.

2.1 Domain Consistency

Consistency properties are associated with a particular data domain. As an example, consider the bits of a byte as the domain. *Byte consistency* means that a thread observes all bits in a byte as updated previously by another thread.

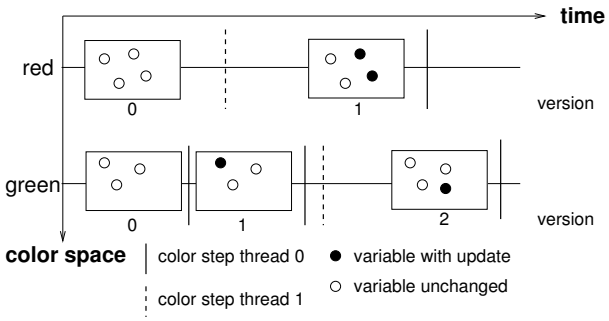

Fig. 1. Domain consistency.

Conceptually, every access to a byte constitutes a transaction with isolation and atomicity properties. The granularity of the *consistency domain* is one byte, and the *synchronization epoch* at the boundaries of which consistency is established is a single memory access. In current machines, byte consistency is automatically enforced by hardware.

Domain consistency extends this model to regions of memory and sequences of operations. The extent of a *consistency domain* is explicitly specified through *coloring*: data with the same color belong to the same domain. Coloring happens at the time when data is allocated and initialized. Each address maps to at most one color.

Synchronization epochs are specified through a *color step*. Color steps mark positions in the code where the programmer considers that the accesses to a consistency domain by a thread are complete, or where a thread should have the opportunity to observe other threads' updates to the domain. A synchronization epoch starts implicitly at the first data access to a domain (following a color step). Once inside a synchronization epoch, a thread observes the state of memory in isolation. Updates are made visible atomically when a color step is encountered. When a thread executes inside a synchronization epoch, we say that the thread 'holds the color' corresponding to the epoch.

Figure 1 illustrates domain consistency. A thread observes the variables of a domain at any point in time in a consistent state. A color step specifies that a thread is willing to proceed to a more recent version of a domain; if there are no concurrent updates to the domain, the version is unchanged. Domains can grow and shrink in size as data is allocated and deleted. Consecutive versions of a domain differ in the value of at least one variable. When concurrent threads update a domain in a conflicting manner and race for the corresponding color step, the underlying transaction mechanism lets one thread successfully commit its version and rolls back other threads racing for the step on the same color.

A color step differs from a memory fence. First, a step pertains only to a specific domain and does not order accesses to memory in other domains. Secondly, a memory fence merely orders the memory access operations, while a color step may trigger a transaction to complete.

Not all shared variables need to participate in domain consistency, i.e., there can be 'uncolored' shared memory. We assume that a color step orders preceding accesses of uncolored memory with respect to the atomic update of the domain (if any). Moreover, color steps are observed in a total global order, irrespective of the domain they operate on. Finally, a color step on a variable whose color is not held or is not colored at all is a no-op.

2.2 Programming Model

Data-centric Synchronization The synchronization model that uses coloring and color steps is *data-centric*: The color is an immutable property of shared variables and domain consistency is guaranteed at any time during the execution of a program. For example, the consistency of a linked list data type may be enforced by declaring all nodes in the list to have the same color and let list operations (insert, remove, ...) be followed by a color step annotation.

Notice that consistency domains that encompass only a single variable are also useful: the memory isolation and atomicity properties provided by data coloring facilitate that a sequence of accesses to a colored variable occurs without interference from other threads. For example, the following code achieves the conditional initialization of a shared variable. We use a pseudo Java notation for illustration.

```
static Object singleton color() = null;
Object getUniqueInstance() {
   Object ret;
   if (singleton == null)
     ret = singleton = new Object();
   else
     ret = singleton;
   color_step(&singleton);
   return ret;
}
```

The declaration of the variable `singleton` specifies that it should be colored; in the example, the color is chosen by the system when the variable is allocated. Domain consistency prevents multiple threads from finding that the variable `singleton` is `null` and creating multiple object instances. The example illustrates that the programmer is solely concerned with specifying the point where data in a domain is consistent. Note that the newly created object is not colored, but the reference to it is. The color step guarantees that any thread that finds variable `singleton` initialized will also observe a completely initialized instance of the object. The color at the **color_step** operation is specified through an object reference or variable address. Also, a **color_step** operation may take multiple arguments that refer to different domains; in all cases, the updates to all domains occur (collectively) in an atomic step.

Domain consistency breaks up the flat model of memory consistency commonly assumed in computer architecture into smaller domains: data-centric syn-

chronization provides strong, transactional consistency guarantees among locations in the same domain but provides no consistency guarantees across domains. The latter can be achieved with operation-centric synchronization. In the data coloring model, domain consistency is the default consistency model.

Operation-centric Synchronization An application may temporarily request consistency that is stronger than domain consistency, e.g., during the execution of a composite operation. We call synchronization that serves to achieve guarantees beyond domain consistency *operation-centric*. For example, an accounting system has individual accounts that each are protected in their own consistency domain. For the purpose of a money transfer operation, a temporary consistency domain is established to encompass multiple account records that are involved in the transfer.

```
void transfer(Account a, Account b, int n) {
   atomic(ALL_COLORS) { // ⟨defer_color_steps(1,ALL_COLORS)⟩
     a.withdraw(n);
     b.add(n);
  }
}                          // ⟨1:color_step(ALL_COLORS)⟩
```

Fig. 2. Operation-centric synchronization with color steps.

Operation-centric synchronization is commonly specified through the start (**tx_start**) and end (**tx_end**) of a transaction. In the model of domain consistency, the boundaries of a transactional operation specify that all color steps encountered after the start of the transaction are coalesced and occur atomically at the end of the transaction. If several domains are involved in the transaction, then the updates of all domains must occur in an atomic step. In Figure 2, the programmer specifies that the updates to both bank accounts must be done atomically. This is accomplished with the **atomic** annotation, which takes as argument the colors of all the domains that need to be temporarily coalesced. The compiler then generates the primitives **defer_color_step** and **color_step**, ensuring that all color steps executed between the two are aggregated. The compiler-generated code is specified in angle brackets ⟨ ⟩.

2.3 Functional Composition

A synchronization mechanism has to be compatible with functional composition, i.e., the combination of synchronization in the caller and callee must maintain the semantics expected by the programmer. For programs with critical sections, this is achieved through nesting — various nesting semantics have been proposed for memory transactions [3–5].

When code with color step synchronization is combined in the call hierarchy, special care has to be taken to prevent a color step in a callee from inadvertently disrupting the consistency window that a programmer expects in the caller. Figure 3 illustrates this on a concurrent container implementation. Initially, the compiler would flag method `getAndRemove` as potentially 'non-atomic', since it invokes other methods (`find`, `get` and `remove`) that may execute color steps.

```
int find(Key key) {
   int index;
   // search, access key, initialize index
   color_step(key, this);
   return index;
}
Value get(int index) {
   Value value;
   // initialize value from index
   color_step(this);
   return value;
}
void remove(int index) {
   // remove value at index
   color_step(this);
}
Value getAndRemove(Key key) {
   atomic(this) {      // ⟨defer_color_step(l, this)⟩
     Value value = null;
     int index = find(key);
     if (index != -1) {
       value = get(index);
       remove(index);
     }                 // ⟨l: color_step(this)⟩
   return value;
}
```

Fig. 3. Composition of color steps.

The programmer then inserts the `atomic` directive, which ensures that `getAndRemove` proceeds on a single version of `this`. This solution to the composition problem is similar to operation-centric synchronization: The execution of color steps for the color of the `this` object in downstream methods is disabled as illustrated in Figure 3. It is the compiler that automatically synthesizes **defer_color_step** and **color_step**, specifying that color steps on the color of the `this` object should be disregarded until execution reaches label `l`. Note that the color of the `key` object is released at the end of `find`.

This approach, which forces computation between the two directives onto a single version of a domain, is consistent with the principles of the data coloring programming model: the programmer must be aware of any disruption of

isolation (non-atomicity) and have simple means to enforce domain consistency within the local scope of a method or code block.

2.4 Synchronization Defects

In the data coloring model, the nature of synchronization defects, their detection, and consequences differ from critical section synchronization. Synchronization defects fall into one of:

- *Incorrect coloring*: Three cases are possible: (i) Shared mutable locations are not colored. Such a situation can be detected by tracking accesses to uncolored locations. (ii) Variables with mutual consistency constraints are colored differently. In such case, inconsistencies among the variables can arise due to unordered concurrent access. In a debugger, color information can help identify race conditions as a possible reason for data inconsistency. The data inconsistency due to such race conditions can be corrected by giving the same color to the variables that are found with inconsistent values. (iii) Variables with different consistency constraints have the same color. In this case, when the programmer releases a color in a color step, she may be committing updates to variables that should be in different consistency domains, leading to unexpected program behavior. Static analysis allows the programmer to obtain a report, check and correct colors for variable declarations.
- *Omission of a color step*: This `cannot` lead to consistency (safety) violation. However, a thread that holds a color and does not step ahead in the version space may harm overall progress (liveness defect). This situation can be detected easily, identifying the thread and the color that caused the problem.
- *Violation of atomicity*: A potential violation of atomicity [6] can occur during the execution of a block or method when encountering more than one step of the same color which are not coalesced by a surrounding code-centric transaction specification. Since the programming model assumes that the occurrence of color steps (and hence atomicity properties) are specified along with methods, violations of atomicity can be reliably detected through program analysis either within the block or traversing the call graph.

In summary, the detection of synchronization defects in the data coloring model can be achieved in a mostly determinate manner, i.e., independent of the thread scheduling. Omitting color steps affects liveness properties but not safety.

3 Transaction Inference

Domain-level consistency can be implemented using transactional memory (TM). Based on the specification of colors, the color steps and the dynamic memory access stream, a system can automatically infer the points in the execution when transactions should be started and committed. Figure 4 illustrates this inference process.

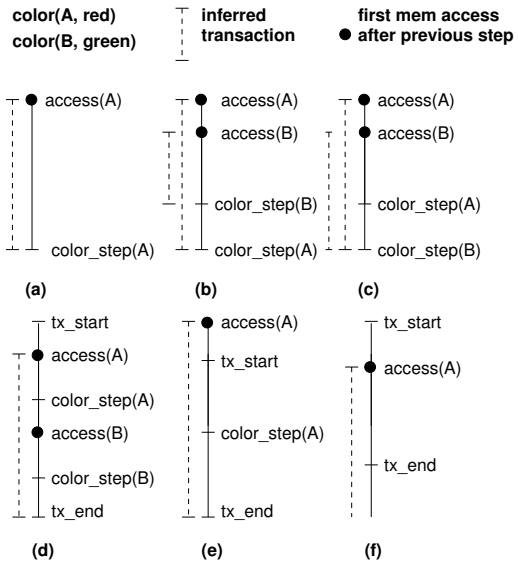

Fig. 4. Transaction inference from the memory access stream and the color steps. **tx_start** an **tx_end** denote start and commit of explicit transactions. Nested transactions in this implementation follow closed nesting semantics.

When a colored memory location is accessed for the first time after the previous step of the same color, the executing thread acquires the color and starts a transaction. When executing a color step for the color, the thread releases the color and commits the transaction. Inferred transactions nest following closed nesting semantics, as shown in Figures 4(b) and (c). Figure 4(b) illustrates the properly nested case, where the color step after the access to B is for B's color. Figure 4(c) shows the case where the first access to B is followed by a color step for A's color — making the colored section not properly nested.

Figures 4(d), (e) and (f) illustrate scenarios when colored sections nest with explicit, i.e., operation-centric, transactions (operation-centric synchronization). Figure 4(d) shows the simplest case, where an explicit transaction fully encapsulates the work of colored sections. Figure 4(e) shows a non-properly nested case that is legal in our implementation because of the support for software composability. Figure 4(f) shows a case that we consider legal but that may hint to an omitted color step: An explicit transaction is ended (**tx_end**) before the color step of a section that started after the explicit transaction start is encountered. Since an explicit transaction should be employed to carry out a set of operations atomically – if these operations involve access to colored data – they are considered complete when the explicit transaction ends.

One important aspect of the transaction-based implementation of our model is directly related to the nesting semantics: The step terminating a nested colored section that only read data performs an early release [7] of all data of the respec-

tive color. Note that color step declares the intention to observe a new version of the domain and hence this naturally maps to an early release implementation.

3.1 Architectural Support

Our DCS programming model requires a system to inspect all memory accesses in order to determine the color of the memory location accessed. A software implementation is impractical because of the need to instrument every memory access to determine its color, potentially resulting in a prohibitive runtime overhead.

Processors already validate every memory access for protection. We propose to extend the protection mechanism to determine the color of a memory location at access time and the color context of the accessing thread. Also, hardware support could determine what colors are live and check for cases of nesting violation with minimal overhead.

A fully detailed architecture proposal and its implementation is described in [2]. In this section, we summarize the set of hardware and software primitives that suffice to fully support our DCS programming model described in Section 2.2, as follows:

Memory Coloring . Support for coloring arbitrary regions of memory. The finer the granularity the better, a cache-line granularity suffices in practice. This can be implemented extending memory protection information with the color id [2]. The mechanism to attribute colors to regions of memory requires joint hardware and operating system support, since it involves manipulating information associated with memory protection.

Color Context Tracking . Instructions to acquire and release colors. The hardware keeps track of the set of colors held by each thread. A color step is mapped to a color release instruction that releases the corresponding color and commits the corresponding transaction. Note that in the examples in Section 4 we show **color_step** taking a variable or address as parameter; this means that the color information for that address is obtained implicitly.

Event Triggering . An event triggering mechanism to notify the software when synchronization actions are necessary. An event is raised when a thread accesses a colored location outside the corresponding color context. This event calls a software handler that acquires the corresponding color using the acquire instruction and starts a transaction.

Underlying TM Support . We assume transactional memory as the underlying synchronization support. This includes the typical instructions for TM, like **tx_start**, **tx_end**, etc. Note we also require support for closed nested transactions.

4 Example: Concurrent Bounded Buffer

We use the following example to show how critical sections, e.g., implemented by (nested) transactions, can limit concurrency and hence hamper scalability and performance. The data coloring model does not suffer from this problem.

```
class BoundedBuffer {
  Object[] buffer = new Object[SIZE];
  int putIndex, takeIndex,nofUsedSlots;
  ...
  void put (Object o) {
    while (true) {
      atomic {
        if (nofUsedSlots < buffer.length) {
          insert(o);
          nofUsedSlots++;
          return;
        }
      }
      /* backoff and wait */
    }
  }
  Object take() {
    while (true) {
      atomic {
        if (nofUsedSlots > 0) {
          Object o = extract();
          nofUsedSlots--;
          return o;
        }
      }
      /* backoff and wait */
    }
  }
  void insert (Object o) {
    buffer[putIndex] = o;
    putIndex = (putIndex + 1) % SIZE;
  }
  Object extract() {
    Object ret = buffer[takeIndex];
    takeIndex = (takeIndex + 1) % SIZE;
    return ret;
} }
```

Fig. 5. Bounded buffer based on critical sections.

Figure 5 illustrates a bounded buffer permitting concurrent access at both ends by producers (method `put`) and consumers (method take). The implementation is based on a sliding window (`putIndex`, `takeIndex`) over a fixed size array (`buffer`) with wrap-around. Implementation details like backoff and queuing are omitted for clarity.

An important goal of the design of a concurrent bounded buffer is to decouple the operation of putters and takers as much as possible. With critical sections, this goal can be achieved by minimizing the time period during which variables that are commonly updated/accessed by putters and getters must be kept consistent.

In the case of the `BoundedBuffer` class, the highly contended variable is `nofUsedSlots`. The design with flat transactions shown in Figure 5 does not achieve the aforementioned goal: variable `nofUsedSlots` is accessed in one transaction together with the `insert` and `extract` methods.

Nested transactions [3–5] have been proposed to mitigate the negative impact of contention on the performance of memory transactions. Figure 6 illustrates such a design for the `put` method: Access to `nofUsedSlots` is encapsulated in nested atomic blocks. The code is correct. However, the goal of removing the

```
class BoundedBuffer {
  ...
  void put (Object o) {
    while (true) {
      atomic {          // (1)
        bool do_insert;
        atomic {        // (2)
          do_insert = nofUsedSlots < buffer.length;
        }
        if (do_insert) {
          insert(o);
          atomic {      // (3)
             nofUsedSlots++;
          }
          return;
        }
      }
      /* backoff and wait */
} } }
```

Fig. 6. Optimized put method with nested transactions.

```
class BoundedBuffer {
  Object[] buffer color(this) = new Object[SIZE];
  int putIndex,takeIndex;
  int nofUsedSlots color();
  ...
  void put (Object o) {
    while (true) {
      if (nofUsedSlots < buffer.length) {
        color_step(&nofUsedSlots);           // (a)
        insert(o);
        nofUsedSlots++;
        color_step(&nofUsedSlots, buffer);   // (b)
        break;
      }
      /* backoff and wait */
} } }
```

Fig. 7. Implementation of `put` method with color steps.

(potentially lengthy) execution of `insert` from the critical path during which isolation of variable `nofUsedSlots` has to be preserved is not achieved: If transactions (2) and (3) follow the semantics of *closed nesting* [4], then transaction (3) must observe the same value of variable `nofUsedSlots` as transaction (2). This is not necessary for this code to be correct. Alternatively, if transaction (2) followed the semantics of *open nesting* then the code would not be correct because the value of `buffer.length` and the state of the buffer encountered during `insert` might not be consistent.

Figure 7 shows how data coloring can achieve this complex synchronization without over-specifying consistency. There are two consistency domains: one for variable `nofUsedSlots`, and another domain for the remaining variables of a `BoundedBuffer` instance. This coloring is achieved through the `color` declarations. Fields without explicit color declaration obtain the color specified when the object is allocated. Notice that there is no explicit critical section in the code because synchronization in the bounded buffer is purely data centric. The color step instruction (a) explicitly expresses that – for the correct operation of the buffer – it is *not* necessary that the variable is held in isolation beyond that point. The `insert` operation and the following increment of `nofUsedSlots` occur

atomically. Finally, (b) allows both color domains to simultaneously transit to the next version. It would be incorrect to specify the color steps in sequence, such that the update of `nofUsedSlots` counter would be visible before the update to the `buffer`. Also, the fine grain specification of color steps would be incorrect if variables `nofUsedSlots` and `buffer` would share the same color (see discussion on incorrect coloring in Section 2.4), since the consistency window on the `buffer` object would be unduly disrupted (due to color aliasing) by color step (a). The declaration of variable `nofUseSlots` explicitly requests a color different from the other fields of the `BoundedBuffer` instance (**`color()`**).

Notice that it is easy to infer from the code or an execution, that the body of the while loop may not execute atomically because there are two steps of the same color in that block. It is exactly this selective relaxation of atomicity that is not achieved by transaction nesting.

We feel that the addition of color steps is a natural way to gradually weaken consistency to achieve higher concurrency. True transaction nesting targets the same goal but seems in this case unintuitive and is not capable to achieve the desired semantics.

5 Quantitative Justification

We support our claim that data-centric concurrency control is the common case by analyzing synchronization patterns in three large software packages: MySQL, WebSphere and Sun's Java Runtime Environment (JRE). Table 1 summarizes the collected data.

MySQL 5.0.22		**WebSphere**		**SUN JRE 1.50**	
LOC	1.5m	# Classes	11343	# Classes	13081
# Files	2336	# Sync Methods	2029	# Sync Methods	5337
# CS	1275	# Static Sync Methods	546	# Static Sync Methods	915
Data-centric CS	84%	# Sync Blocks	2119	# Sync Blocks	5337
		Sampled DC Sync Blocks	72%		
		Data-centric CS	75%		

Table 1. Estimation of the proportion of data-centric critical sections in MySQL, WebSphere and SUN JRE.

MySQL is a large parallel application with thousands of files and over a million lines of code, written in C. Its critical sections operate under the protection of thousands of different lock instances. We inspected all critical sections in the code and classified them as data-centric or operation-centric. Most critical sections (84%) in MySQL are data-centric. Among the many locks used, three global locks (`kernel_mutex, LOCK_thread_count, and LOCK_global_system_variables`) protect a large number of distinct critical sections (70, 62 and 31 respectively) and are spread over many files (14, 14, and 7 respectively). In this code-centric, non-local reasoning using critical sections

for large applications, it is easy to overlook the need for a critical section and introduce data races. Using a data-centric model, the programmer simply colors the data-structures, and marks the places in the code where data is consistent with color steps. All accesses to shared data would be guaranteed to be inside a system synthesized transaction.

In the case of WebSphere – a few million lines of Java code – we classify locks according to the following criteria: synchronized methods are data-centric, while static synchronized methods and synchronized blocks are operation-centric (since their lock is not associated with any specific object instance). We randomly sampled and manually inspected the code to determine that this classification holds over a large number of cases. For the SUN JRE, we use the same criteria as WebSphere but did not inspect the source code to determine the nature of the synchronized blocks.

When inspecting the code, we classified critical sections as data-centric when their purpose was obviously to keep shared data structures consistent. Critical sections were classified as operation-centric when they performed a collection of unrelated tasks, typically calls to apparently unrelated functions or methods. When in doubt, we classified critical-sections as operation-centric. Using this conservative approach, we find about 75% of the critical sections to be data-centric.

When inspecting synchronized blocks in WebSphere, we frequently observed cases where synchronized blocks were used to avoid having long critical sections by making the whole method synchronized. In those situations, the blocks were synchronized on *this*. Other common patterns were composite objects whose methods had synchronized blocks on the encapsulated object instances.

6 Related Work

McKenney [8] discusses several synchronization patterns encountered in parallel programs. Particularly related to this paper are the Code Locking, Data Locking and Data Ownership patterns. In our classification, both Code Locking and Data Locking are code centric; they differ in the granularity at which locks are placed in the program.

Vaziri et al. [1] present extensions to Java to allow the association of synchronization constraints to objects. In their proposal, the programmer, when declaring the fields of a class, can group them into "atomic sets". Based on the atomic set annotations and using static analysis, locks are inferred to guarantee method-level atomicity. McCloskey et al. [9] developed Autolocker, which allows the programmer to associate data structures with locks together with atomic section annotations. The system then automatically infers the synchronization operations necessary to provide the specified atomicity.

Monitor-based concurrency control [10] is data-centric, as data is explicitly declared as shared and is associated with a predefined set of procedures. If the data accesses are performed through the monitor procedures, mutual exclusion is automatically guaranteed.

Transactional memory (TM), e.g., [11, 12, 5] is a mechanism for optimistic concurrency control. The common programming model for TM is code-centric: Transaction boundaries are specified in the control flow of the program. There is no explicit specification of shared data. Recent work by Ni et al. [13] explores how transaction nesting can be specified in a data-centric manner.

Finally, this work extends the data-centric synchronization model in Colorama [2]. Most notably, Colorama does not have the concept of a color step. It relies on an exit policy – e.g., automatically end a critical section at the end of the method that started it.

7 Conclusions

In this paper, we introduce concurrency control with *Data Coloring*, a programming model based on the principles of our previous work on architecture support for data-centric synchronization [2]. We justify the choice of data-centric synchronization with an empirical study of critical sections in several large applications. The study revealed that more than 75% of the critical sections are used in a data-centric manner, even though the concurrency control is specified in a code-centric manner using critical sections with locks.

The data coloring programming model emphasizes data-centric synchronization and matches intent with implementation. Consistency domains are specified using colors, while color steps allow the programmer to express when the transitions between consistent states are safe. The additional information helps the system infer the right granularity of transactions, without unnecessarily burdening the programmer. Data coloring enables programmers to reason locally about consistency properties. Additional benefits of data coloring are its enhanced safety properties: synchronization defects can be detected efficiently and in a determinate manner.

References

1. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL'06, New York, NY, USA 334–345
2. Ceze, L., Montesinos, P., von Praun, C., Torrellas, J.: Colorama: Architectural support for data-centric synchronization. In: HPCA'07
3. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime support for effficient software transactional memory. In: PLDI'06
4. Moss, J.E.B., Hosking, T.: Nested transactional memory: Model and preliminary architecture sketches. In: SCOOL'05
5. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: A high performance software transactional memory system for a multi-core runtime. In: PPoPP'06
6. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI'03
7. Herlihy, M., Luchangco, V., Moir, M., William N. Scherer, I.: Software transactional memory for dynamic-sized data structures. In: PODC'03

8. McKenney, P.: Selecting locking designs for parallel programs. In: Pattern Languages of Program Design. (1996)
9. McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: Synchronization inference for atomic sections. In: POPL'06, New York, NY, USA 346–358
10. Hoare, C.: Monitors: An operating system structuring concept. CACM **17(10)** (1974) 549–557
11. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional Memory Coherence and Consistency. In: ISCA'04
12. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA'93, New York, NY, USA 289–300
13. Ni, Y., Menon, V., Adl-Tabatabai, A.R., Hosking, A.L.: Open nesting in software transactional memory. In: PPoPP'07