Contents

5.2 Serialisability

A transaction manager should ensure that the effects of functions accessing an Immutable Data Structure are equivalent to those of a serialisable execution. Functions acting on an Immutable Data Structure can be mapped onto abstract read and write operations on variables and a concurrency control protocol can be enforced on these operations to ensure serialisability. The protocol permits functions to act on an Immutable Data Structure simultaneously, although not all of them succeed.

Functions acting concurrently on an Immutable Data Structure can be made linearizable but enforcement of this property restricts scalability because when two functions simultaneously act on the same data structure only one of them is successful. To improve scalability functions should be able to simultaneously act on the same data structure successfully. The problem is how to ensure the serialisability of functions that simultaneously act on a data structure?

The main contribution of this section is a technique for making functions simultaneously acting on an Immutable Data Structure serialisable. This section focuses on mapping these functions onto abstract read and write operations on the variables considered by a concurrency control protocol.

5.2.1 Simultaneous access

This section considers how two functions can be permitted to act simultaneously on an Immutable Data Structure.

When functions simultaneously access a semantically linearizable Immutable Data Structure only one of them succeeds.

Section 5.2.3 discusses the semantics of functions concurrently accessing an Immutable Data Structure.

The property of immutability allows the implementation of a mechanism that permits simultaneous access while ensuring that the actions of one function appear to precede those of the other.

Section 5.2.4 discusses the semantics of functions simultaneously accessing an Immutable Data Structure.

The problem of ensuring the serialisability of functions which simultaneously access shared data has been successfully solved in the context of database systems. A database system ensures the correct concurrent semantics of transactions simultaneously acting on a relational table by serialising the file operations on the records that implement it. The file operations on these records are mapped to abstract read and write operations and the transaction manager enforces a concurrency control protocol on these operations to ensure that their effect is equivalent to a serial execution. In a database system the variables on which the concurrency control protocol acts are records and the operations that it considers are file operations. The records are, typically, the leaves of a B+tree data structure which maintains the application data. There is a layer of abstraction between a database table and the B+tree which implements it, so there is a complex relationship between a transaction expressed in SQL and the abstract read and write operations considered by the concurrency control protocol [GR92].

5.2.2 Implementing Concurrency Control

The problem of ensuring the serialisability of functions simultaneously accessing an Immutable Data Structure is one of mapping the functions onto a concurrency control protocol and enforcing that protocol.

A concurrency control protocol is normally expressed in terms of a history of abstract read and write operations on a system of variables so the functions must first be mapped to operations on a set of variables.

Section 5.2.6 describes how an Immutable Data Structure is mapped onto variables for the purposes of concurrency control.

Section 5.2.7 describes how the functions acting on the Immutable Data Structure are mapped onto abstract read and write operations on variables.

The concurrency control protocol is enforced by a validate function that ensures that conflicting operations conform to the protocol. Functions which contain non-conforming operations are rejected.

Section 5.2.8 describes how conflicting operations can be detected by a validate function.

A concurrency control protocol can be expressed as a set of invariants on meta-data associated with abstract read and write operations. The Time Stamp Ordering concurrency control protocol requires that these abstract read and write operations are associated with time stamp meta-data which the functions collect and record.

Section 5.2.9 describes how information about the operations can be recorded as meta-data within an Immutable Data Structure and how the Time Stamp

Ordering concurrency control protocol can be enforced.

5.2.3 Concurrent semantics

Immutable Data Structures have the property of structural linearizability. Structural modifications take place in isolation and appear to be atomic so no matter which functions are concurrently applied to the data structure the resulting structure is always a valid structure. However, the property of structural linearizability does not endow the ADT presented by the data structure with any meaningful concurrent semantics. The concurrent behaviour of a structurally linearizable data structure is uncertain because it may not reflect the action of all of the functions that have acted upon it.

An Immutable Data Structure can be made semantically linearizable which ensures that concurrently executing functions appear to take place at a single moment in time. No matter which functions are concurrently applied to the data structure the resulting structure is equivalent to some serial execution of those functions. The concurrent semantics of a semantically linearizable Immutable Data Structure are intuitive because functions appear to occur in some serial order. However, only one of the functions simultaneously accessing the Immutable Data Structure is successful and this limits scalability.

An Immutable Data Structure that permits tardy read access to past versions while ensuring the serialisability of mutating functions has the property of partial persistence. This ensures that mutations appear to take place at a single moment in time but the result of non-mutating functions do not necessarily reflect the latest version of the data structure. The concurrent semantics of a partially persistent data structure are easy to understand and can be useful. Some applications require that mutations are serialised to ensure that a data structure eventually reflects their effects, while permitting tardy read accesses. Partial persistence can improve the scalability of concurrent applications because mutating and non-mutating functions can execute at the same time.

For example, communication routers usually map symbolic names to IP addresses using a map based data structure called a PATRICIA trie [Mor68]. The map is read each time a message is processed, which occurs frequently, but it is only written when new IP addresses are added, which happens rarely. If the penalty for an incorrectly routed message is small then a partially persistent map can be appropriate. A partially persistent map permits read-only and write accesses to take place simultaneously, while serialising writes. It separates the concerns about the structure of the data, which is ensured by serialising access to the root, from both concerns about the semantic order of modifications, which is ensured by serialising writes, and concerns about the routing of messages.

5.2.4 Simultaneous semantics

This section considers the simultaneous behaviour programmers might expect from an ADT.

Deque

The most desirable simultaneous behaviour for a deque would be for it to permit serialisable simultaneously accesses to both ends. This behaviour can be described in terms of the serialisable access to two variables, each representing a different end of the queue.

For example, a Producer Consumer Queue is an application of a deque used to communicate between concurrent processes. One process inserts elements on one end of the queue and another process removes them from the other end. It is desirable that processes can simultaneously insert and remove elements.

Map

The most desirable simultaneous behaviour for a map would be for it to permit simultaneous access to different groups of elements while ensuring that the accesses to a single group of elements are serialisable. This can be described in terms of the serialisable access to variables.

For example, fine-grained serialisability can be ensured by associating a small discrete group of elements with a variable and coarse-grained serialisability can be ensured by associating a larger discrete group with a variable.

Priority queue

The most desirable simultaneous behaviour for a priority queue would be for it to permit simultaneous insertion of elements into the queue while ensuring that the highest priority element is removed in serialisable order. The desirable behaviour can be described in terms of the serialisable access to two variables, one representing the highest priority element and the other representing the rest of the priority queue.

For example, an event scheduler is an application of a priority queue used to communicate between concurrent processes. A process requesting an event inserts an element onto the priority queue. Another process services the queue by removing the highest priority element from the priority queue. It is desirable that elements can be inserted by one process while the highest priority element is removed by another. The insertion of elements onto the priority queue should be serialisable and the removal of the highest priority element should also be serialisable but it is not necessary to impose a serial order on all operations.

Vector

The most desirable simultaneous behaviour for a vector is semantic linearizability. A mutating function has the potential to modify the relationship between the ordinal numbers and values of any element in the data structure so simultaneous access cannot be permitted. The desirable behaviour can be described in terms of the serialisable access to a single variable representing the entire vector.

5.2.5 Previous work

The problem of permitting simultaneous access to the data structures used in on-line gaming is commercially important and has received significant attention. Multi-player on-line game applications are usually constructed around a massive aggregate data structure called the game tree. The game tree contains information about all of the objects within the game such as players and weapons and their relationships. Actions in the game, such as a player dropping a weapon and another player picking it up, are represented by actions on the game tree. Access to the game tree is typically serialised by mutual exclusion. Sweeney identifies the serial nature of actions on the game tree as a significant obstacle restricting the performance of on-line games [Swe06]. Gajinov describes how Transactional Memory can be used to improve the performance of an on-line game by allowing actions on the game tree to execute speculatively [GZU$^+$09]. The challenge is to ensure correctness while permitting multiple functions simultaneous access the data structure.

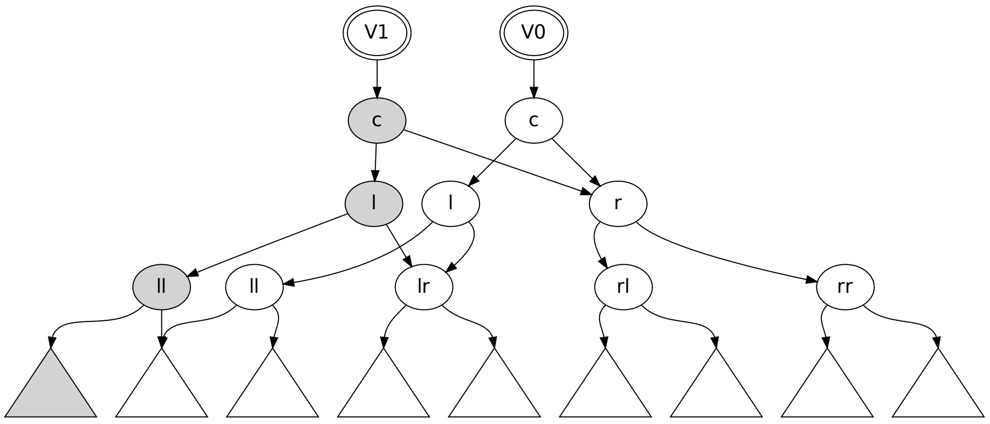


Figure 5.1: **Labelling of variables in the cap of an Immutable Data Structure.** The variables in the cap of an Immutable Data Structure are labelled **ll**, **l**, **lr**, **c**, **rl**, **r** and **rr**. The shaded path represents new instances of the variables. The triangles represent subtrees suspended by the cap.

5.2.6 Variables

For the purposes of concurrency control an Immutable Data Structure can be regarded as a system of variables. A concurrency control protocol ensures the correct concurrent semantics of abstract read and write operations acting on these variables. The functions implemented by the Canonical Binary Tree do not maintain any mutable state so variables must be maintained immutably within the Immutable Data Structure itself. The relative position of a vertex to the root can be regarded as a variable and the annotation of a vertex can be regarded as its value. A variable can have different values in each version of the data structure even though the vertices that implement it are immutable.

Figure 5.1 illustrates the labelling of variables within a tree. Each variable represents a position relative to the root. The value of a variable can only be altered by creating a new version of the tree.

For the purposes of concurrency control it is only necessary to consider the variables represented by a subset of the relative positions in the tree that we call the cap. A version of the Immutable Data Structure can be larger or smaller than the cap so a vertex may or may not correspond to a variable in cap. When the data structure is larger than the cap the variables represented by the leaves of the cap act as proxies for the subtrees which they suspend.

The desirable behaviour of a deque can be described in terms of the serialisable access to three variables **l**, **r** and **c**. Variables **l** and **r** represent the front and back of the deque respectively and the variable **c** represents the empty queue.

A map can be represented either at a fine level of granularity, or at a coarse level of granularity. The size of the cap determines the level of granularity.

A priority queue can be represented by two variables. The variables **c** and **l** represent the highest priority element and the rest of the priority queue respectively.

A sequence can be represented by a single variable **c**.

5.2.7 Functions and operations

When a variable is read or written information about the operation is recorded in the data structure. A variable in the cap is either read, written or unaffected by a function. A function is implemented by a path copy which creates new nodes. A node records the type of abstract read and write operations that created it, along with the time stamp meta-data required to enforce the concurrency control protocol.

For the purposes of concurrency control the annotation of a node corresponds to the value of a variable. An operation is regarded as writing a variable if the annotation associated with its relative position in the tree changes. A read operation records an access to a variable which did not change its value. When the annotation associated with a relative position that is not in the cap changes a write operation is recorded as acting on the variable that corresponds to the node's most junior ancestor in the cap.

A *query*() function causes every variable on the path to be read but it is only necessary to record reads in nodes corresponding to variables in the cap. The nodes on the path read by the query function that correspond to variables in the cap are copied so that read operations can be recorded.

The *insert*() and *delete*() functions also read every variable on the path but they also cause the annotation of some of the variables on the path to change. The variables in the cap act as proxies for the variables in the subtrees they suspend so a change in the annotation of a node at some point on the path is represented by a write operation on a variable within the cap.

Figure 5.2 illustrates the abstract read and write operations on variables

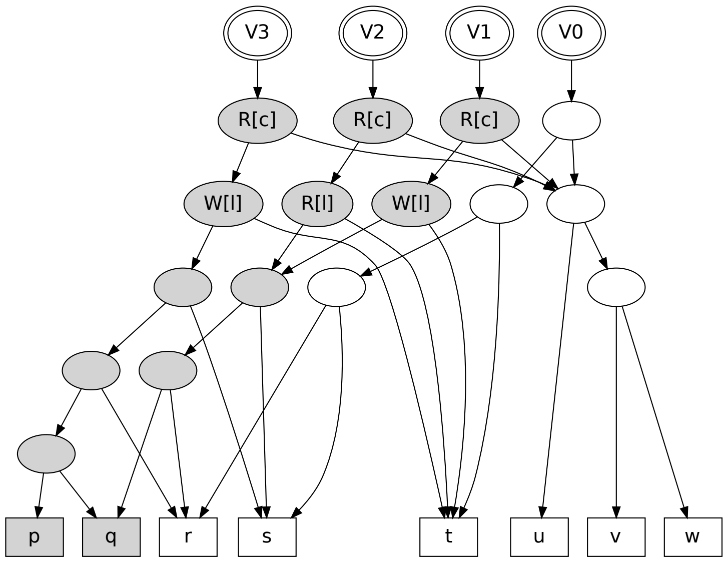


Figure 5.2: **Operations on variables in the cap of a deque.** The cap contains three variables **l**, **r** and **c**. The function $Push\_front(q)$ acts on version V0 containing $\{r, s, t, u, v, w\}$ to create version V1 containing $\{q, r, s, t, u, v, w\}$. The operations $\{\mathbf{W}[\mathbf{l}],\ \mathbf{R}[\mathbf{c}]\}$ are recorded in the vertices corresponding to the cap. The function $Front()$ acts on version V1 to create version V2. The vertices corresponding to the cap are copied to record the operations $\{\mathbf{R}[\mathbf{l}],\ \mathbf{R}[\mathbf{c}]\}$ performed by this non-mutating access. The function $Push\_front(p)$ acts on version V2 to create version V3 containing $\{p, q, r, s, t, u, v, w\}$. The operations $\{\mathbf{W}[\mathbf{l}],\ \mathbf{R}[\mathbf{c}]\}$ are recorded in the vertices corresponding to the cap.

Cap	**ADT**	**Semantics**	**Access**
$\emptyset$	All	Structural linearizability	Uncontrolled
{**c**}	All	Semantic linearizability	Serialised
{**c**}	All	Partial persistence*	Tardy reads
{**l**,**c**,**r**}	Deque	Serialisable	Simultaneous
{**l**,**c**}	Priority queue	Serialisable	Simultaneous
{**l**,**c**,**r**}	Map	Fine grain serialisable	Simultaneous
{**ll**,**lr**,**l**,**c**, **rl**,**r**,**rr**}	Map	Coarse grain serialisable	Simultaneous

Table 5.1: **Cap topology and granularity of concurrency.** The topology of the cap controls the granularity at which concurrency control is enforced. The variables represented by the cap are listed in the first column. The third column describes the semantics of the ADT. The permitted access is listed in the forth column.
(*) Partial persistence is ensured by serialising mutating functions only.

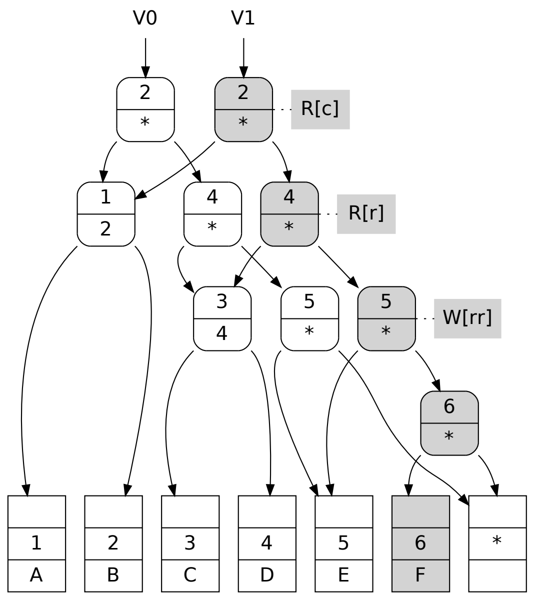
recorded in the cap of a deque. Nodes in the Immutable Data Structure record information about the operation that created them. Read and write operations on the right node, **r**, can be labelled **R**[**r**] and **W**[**r**] respectively.

Figure 5.3 illustrates the abstract read and write operations on variables recorded in the cap of a map.

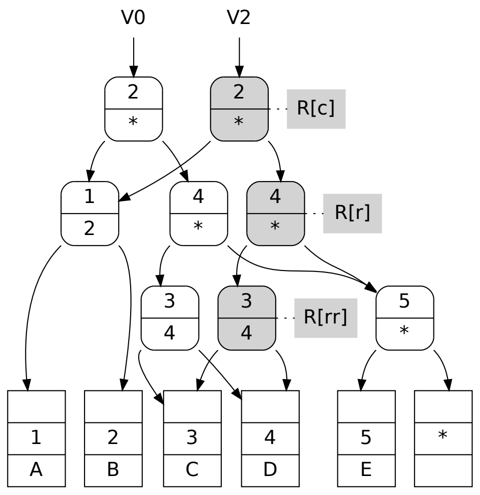
The cap can enforce serialisability at any level of granularity including making all functions accessing the data structure linearizable. A vector can be made linearizable by serialisable access to a single variable **c**.

Table 5.1 describes how the cap determines the granularity at which concurrency control is enforced.

The Canonical Binary Tree hides structural information from the application making the topology of the tree independent of the functions acting on it. The topology of a Canonical Binary Tree is not invariant because the tree may be balanced at any time. During balancing the topology of the tree is modified causing new nodes to be created. These new nodes must maintain information about the abstract read and write operations on the variables they represent. After a balancing rotation, information about abstract read and write operations is in the same positions relative to the root. The skew and split balancing rotations cause the annotation of a node to change resulting in an abstract write operation to the corresponding variable.



(a)



(b)

Figure 5.3: **Operations on variables in the cap of a map.** The cap contains the variables **ll**,**lr**,**l**,**c**,**rl**,**r** and **rr**.
(a) The path created by the function $Insert(6 \mapsto F)$ which creates version V1 is shaded. The operations $\{$**W**[**rr**], **R**[**r**], **R**[**c**]$\}$ are recorded in the vertices corresponding to the cap. The annotation of the variable **rr** does not change, but it is recorded as a write because there is a change in the subtree that it suspends.
(b) The $Query(4)$ operation creates a new version V2 of the data structure to record the operations $\{$**R**[**rl**], **R**[**r**], **R**[**c**]$\}$ in the vertices corresponding to the cap.

5.2.8 Validation

The concurrency control protocol is enforced by the validate function that takes as its arguments two versions of the Immutable Data Structure. It considers the operations on variables in the caps of both versions. Operations conflict if they act on the same variables and one of them is a write. Conflicting operations may or may not conform to the concurrency control protocol. The validate function determines whether the versions contain conflicting operations that violate the protocol.

Figure 5.4 illustrates conflicting and non-conflicting operations on a deque.

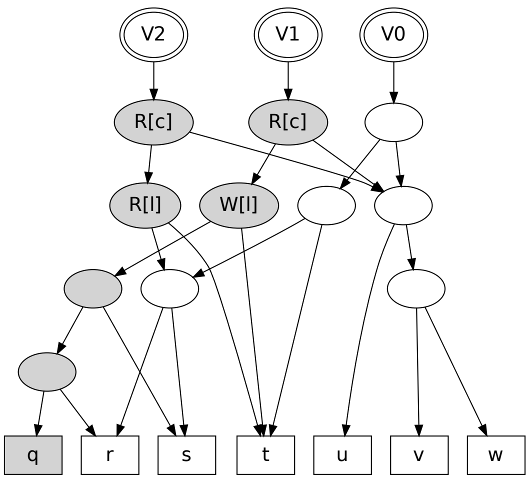
A value representing the topology of the cap is passed as a parameter to the validate function. The validate function traverses the nodes in the cap of both versions and compares the operations acting on nodes corresponding to the same variable. When conflicting operations are detected the time stamp meta-data is considered. The function returns a binary value which indicates whether or not the two versions contain conflicting operations that are not permitted by the protocol.

5.2.9 Meta-data

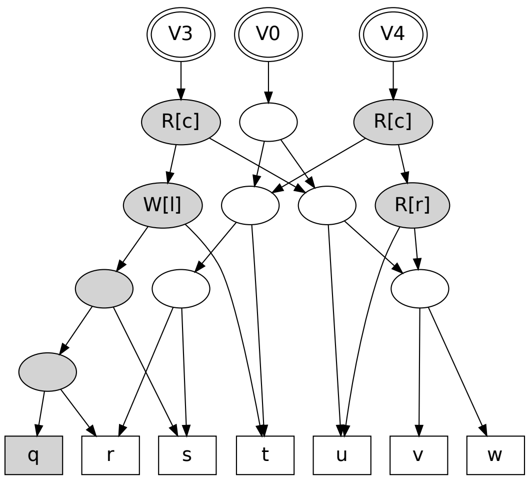
The two versions of the Immutable Data Structure considered by the validate function do not necessarily represent the application of single functions to a common ancestor version. If they did then conflict resolution would only be a matter of detecting conflicting abstract read and write operations on the same variable. Indeed, the paths considered by the validate function are of arbitrary complexity as they may represent the action of multiple functions applied to a common ancestor version. To resolve conflicts the Time Stamp Ordering concurrency control protocol is applied to the time stamp meta-data recorded in the nodes.

The Time Stamp Ordering protocol works by comparing the read and write time stamps of conflicting operations to determine whether the operations appear to occur in the order given by the time stamp of the functions. Details of the protocol and a proof that the operations that it permits are always equivalent to a serial execution can be found in Bernstein [BHG87].

The Time Stamp Ordering concurrency control protocol requires that a serialisable system maintains a unique monotonically increasing time stamp source and that a time stamp should be associated with all operations on variables. For



(a)



(b)

Figure 5.4: **Conflicting and non-conflicting operations on a deque.** The cap contains three variables **l**, **r** and **c**.
(a) Conflicting operations. The path created by the function *Push_front*() records the operations {**W**[**l**], **R**[**c**]} in the nodes corresponding to the cap. The function creates a new version V1 by path copying from version V0. The path created by the function *Front*() records the operations {**R**[**l**], **R**[**c**]}. The function creates a new version V2 by path copying from version V0. The functions conflict because they both act on variable **l** and one of them is a write.
(b) Non-conflicting operations. The path created by the function *Push_front*() records the operations {**W**[**l**], **R**[**c**]} in the nodes corresponding to the cap. The function creates a new version V3 by path copying from version V0. The path created by the function *Back*() records the operations {**R**[**r**], **R**[**c**]}. The function creates a new version V4 by path copying from version V0.

the purposes of concurrency control the serialisable system can be considered as a single Immutable Data Structure so a time stamp source is maintained independently by each data structure. A time stamp source can be implemented by an ordinal number using an atomic increment instruction.

A unique time stamp is associated with each function call. Each node retains the time stamp associated with the function that wrote the variable to which it corresponds. Each node also retains the highest time stamp of any function that reads the variable to which it corresponds.

The time stamps must be maintained in the correct positions relative to the root and this requirement dictates the implementation of the functions of the Canonical Binary Tree. Without this requirement the implementation of path copy is somewhat arbitrary. For example, an element can be inserted into a tree by creating a new root node whose children are the past version of the tree and a leaf containing the element. The *insert*() operation cannot be implemented in this way because it will alter the relative position of existing nodes. Instead, the path to an existing leaf must be copied when an element is inserted into the Canonical Binary Tree. The time stamps associated with each node on the path are copied to the new node corresponding to the variable it represents. For example, the second element in a Canonical Binary Tree must be inserted by a leaf to root path copy operation which creates two new nodes to maintain the relative position of time stamps.

Maintaining time stamps in the correct relative positions during balancing is straightforward. The relative position of a node in the subtree suspended by a pivotal node is altered by a balancing rotation. Both the skew and split balancing rotations can be regarded as writing to the variable corresponding to the pivotal node. It is not necessary to consider the time stamps of a node in the subtree suspended by the pivot because this write operation will conflict with any operation affecting a variable in this subtree.

Bibliography

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[GZU$^+$09] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Quaketm: parallelizing a complex sequential application using transactional memory. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 126–135, New York, NY, USA, 2009. ACM.

[Mor68] Donald R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[Swe06] Tim Sweeney. The next mainstream programming language: a game developer's perspective. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 269–269, New York, NY, USA, 2006. ACM.