# Contents

## 2.1 Concurrent IO

The difficulty of presenting a consistent view of shared state to entities outside the control of a program is central to the problem of concurrent programming. A solution to this problem defines the structure of the concurrent program and the nature of the interface with shared state. Concurrent programs do not exist in isolation, they interact with the Operating System, the Network and human interfaces. The requirements for the interaction with entities outside the control of the system should be a primary design concern of any concurrent system.

Concurrent programs that conform to the database programming model present a consistent view of shared state to their external interfaces. A concurrent programming model for Chip Multi-Processors can be developed from the database programming model.

The main contribution of this section is the identification of the features of the database programming model that facilitate interaction with external entities. This section focuses on adapting these features to create a concurrent programming model for a Chip Multi-Processor.

### 2.1.1 The interaction with external entities

Transactional Memory systems do not treat the interaction with external entities as a primary design priority. In fact many Transactional Memory systems do not present any solution for interaction with external entities, other than to support the output of a result at the end of program execution.

A concurrent program must conduct a serial interaction to each outside entity. There must *actually* be a causal, "happens before", relationship between events and their responses and concurrent programs must be built around this causal relationship. It is not possible for a concurrent program to interact with an external entity while it is executing speculatively. Speculative execution may be aborted and restarted, its effects on shared state are speculative and can be undone. However, the interaction with an external entity cannot be undone so this interaction must be restricted to those parts of a concurrent program that are executed inevitably.

A concurrent program presents the appearance of a serialisable interaction with shared state. There must *appear* to be a causal, "happens before", relationship between events and their responses from the point of view of a particular

external entity.

The requirements of the interaction with external entities dictate both the structure of the concurrent program and the nature of its interface with shared state. The causal relationships required by external interaction cannot easily be engineered into a system designed with other priorities.

Transactional Memory systems execute some program code speculatively within an atomic section, which prevents the program from interacting freely with external entities.

Section 2.1.3 discusses atomic sections.

The problem of presenting a consistent view of shared state to entities outside the control of the concurrent program has been successfully solved by the database programming model which describes how a concurrent system should interact with external entities.

Both the Database programming model and the Transactional Memory programming model rely on a transactional approach to concurrent processing. They are targeted at different problem areas and assign different priorities to design criteria. The main difference between them is that the database model regards the concurrent interaction with the client as the primary design concern and regards concurrency as a performance enhancement, whereas Transactional Memory regards concurrent performance as the primary design goal.

Many Chip Multi-Processor systems, such as those used in embedded systems, are heterogeneous. These systems consist of isolated components that communicate with each other thereby internalising the problem of interaction.

Section 2.1.6 discusses the difficulty of writing concurrent applications for heterogeneous processors.

### 2.1.2 The Database Programming Model

This section identifies features of the database programming model that facilitate external interaction and adapts them to create a concurrent programming model for Chip Multi-Processors. It suggests that a program that executes inevitably can present a consistent view of shared state to external entities.

A database application program executes inevitably and restricts speculation to the interaction with the database, whilst allowing the application to interact freely with external entities. The proposal is that a concurrent program should execute inevitably and that speculative execution should be restricted to the

interaction with shared state.

A database system isolates shared state from an application by implementing the client server model, thereby allowing the interaction with shared state to be treated as a transaction. Most Transactional Memory systems isolate shared state weakly to improve performance and this prevents the program from presenting a consistent view of shared state to external entities. The proposal is that shared state should be stored in objects that are isolated from local state. The interfaces to these shared objects should support Memory Transactions. This allows the program to present a consistent view of shared state to external entities.

Section 2.1.5 discusses the Client Server Database programming model.

In our model Memory Transactions are specified in terms of an Application Programming Interface (API) to shared memory. Application programs can interact freely with external entities because they execute inevitably, whilst the shared memory interface executes speculatively. This speculative execution is encapsulated within Memory Transactions that present a consistent view of shared state to the application. This view is passed on to an external entity by the application.

### 2.1.3 Atomic Sections

An atomic section is a programming idiom that supports the development of concurrent programs. An atomic section is a section of program code that appears to be performed atomically and in isolation. An atomic section differs from a critical section because the instructions within the atomic section can be simultaneously executed by more than one processor, whereas a critical section guarantees that only a single processor executes program code within the section at any particular moment in time.

Speculative lock elision is an execution technique that permits the simultaneous speculative execution of program code within a critical section [RG01]. It permits a concurrent program written using mutual exclusion to be interpreted as a program containing atomic sections. Within a section all memory writes are considered speculative and when a conflict occurs the speculative state is rolled back and corrective action is taken. The rationale behind speculative lock elision is that conflicts are rare and that execution of the section is unnecessarily serialised by mutual exclusion.

To detect conflicts it is necessary to distinguish variables that are shared from

variables that are local to a section. In programming languages that allow the use of pointers, such as C, the locality of a variable is not explicitly defined by the program. This limits the utility of atomic sections in general and speculative lock elision in particular. In programming languages that do not allow the use of pointers, such as Java, a system can attempt to determine the locality of variables within a section using techniques such as escape analysis [SR01].

Implementations of atomic sections require the programmer to indicate the locality of variables to the run-time system in some way. However, the object oriented programming model encourages programmers to place logically connected variables with different access characteristics together in the same object. The object oriented model is orthogonal to a model in which the locality of each variable is considered individually.

The apparent simplicity of the use of the *atomic* keyword to identify an atomic section belies the subtle complexities of the use of atomic sections. Atomic sections do not have intuitive concurrent semantics [CGE08]. They are prone to isolation pathologies and are not composable [MBL06].

Database systems support transactions without explicitly supporting atomic sections [WA02]. However, it is informative to consider applying the programming model adopted by Transactional Memory to the programming of a Relational Database system. SQL is a complete functional programming language so complex routines can be written as single SQL statements rather like atomic sections.

Not surprisingly, a database program written in this way has many of the negative characteristics of a program written for Transactional Memory. SQL does not have an IO mechanism so interaction with external systems is restricted. SQL requires that each shared variable must be specified in the database schema so such a program would be tedious to write. For these reasons Database programmers rarely write programs in this style and do not generally express transactions as atomic sections.

The original proponents of Hardware Transactional Memory envisaged a hardware system that would be able to execute programs written for mutual exclusion by concurrently executing critical sections as atomic sections. They imagined that this hardware system could implement transactions transparently and that critical sections could be converted into atomic sections so that applications would not have to be changed. Today, few believe that this is achievable. Transactional

Memory systems require that an application program is significantly modified to support Memory Transactions. Atomic sections seem at odds with modern networked and object oriented applications. Despite this, the basic approach of expressing Memory Transactions as atomic sections has remained the same since Transactional Memory was first proposed.

### 2.1.4 Previous work

It is common for Transactional Memory systems to treat IO and Operating System interaction as engineering problems to be addressed at a late stage in the implementation. However, it is difficult to engineer a serial interaction with external entities into a system primarily designed around the requirements of concurrent execution. This section describes attempts to engineer support for external interaction into Transactional Memory systems.

The main reference book on Transactional Memory describes how Transactional Memory systems perform IO and interact with the Operating System [HLR10]. However, the limited coverage of the topic suggests that the interaction with external systems is not the primary design concern when developing a Transactional Memory system nor is it the main focus of Transactional Memory research.

Transactional Memory systems take three general approaches to interaction. Firstly, they delay interaction by buffering the output produced within an atomic section. The buffer can be discarded if the atomic section is restarted. Secondly, they undo interaction with the Operating System. A memory allocation within an atomic section can be undone should the atomic section be restarted. Thirdly, they stop concurrent execution before interacting with an external entity.

xCall is a Transactional Memory aware API that has been proposed for handling system calls [VTG+09]. xCall addresses the problem of performing IO while executing speculatively. It also addresses the problem that the atomicity and isolation guarantees made by the transactional system do not apply to the Operating System kernel.

xCall provides output facilities to Memory Transactions by buffering IO operations until a transaction has committed. This buffered output can be discarded if speculation fails. The technique makes writing monolithic programs easier as output can be built up as the program runs.

xCall improves the concurrent semantics of some system calls by undoing their

effect when the transaction is aborted. This technique works well for memory allocation but not all Operating System calls are reversible.

Applications in the STAMP benchmark suite stop all concurrent execution before initiating output [CMCKO08]. Software Transactional Memory systems generally approach Operating System interaction in the same way as output. They stop all concurrent execution before making a call to the Operating System.

Operating System interaction complicates the implementation of Hardware Transactional Memory systems and a great deal of engineering effort is required to support it. Many Operating System calls involve a context switch. The state of the transaction prior to the context switch must be preserved and this state must be restored after the Operating System call is complete [KHLW10]. For Hardware Transactional Memory systems that buffer speculative state in cache the context switch associated with Operating System calls is particularly problematic. The Hardware Transactional Memory system must ensure that speculative state held in cache is not flushed during the Operating System call.

In-memory databases implement the database programming model [Gra02]. In-memory database systems execute programs inevitably and present a consistent view of shared state to external entities. However, many of the features of in-memory databases, such as abstract query language and relational tables are not suitable as a model of shared state for concurrent programming. A concurrent programming solution should adopt only those features of the database model that are relevant to supporting the interaction with external entities.

### 2.1.5 The Client Server Database Model

The Client Server Database model addresses a similar problem to Transactional Memory and it shares the goals of supporting scalable concurrent execution and ease of programming. The reason why the programming styles and supporting systems appear so different is that database programs treat the interaction with external entities as the primary design concern and this affects every aspect of the program and supporting system.

The Client Server Database model is a software engineering concept in which the application processing and the management of shared data are regarded as distinct processing tiers. These tiers do not share access to each other's data and the interaction between the tiers is restricted to passing messages between them.

The Client Server Database model provides the appearance of serial execution

to entities outside the control of the system. This is achieved by isolating and serialising the interaction with any particular external entity through a client server relationship. The processing of the interaction with each client is treated as an independent task. These tasks can be executed concurrently while each client experiences a serial interaction with the program.

In the Client Server Database model applications execute inevitably with speculative execution restricted to the accesses to shared data. The execution of a program can be regarded as serial because it is isolated from concurrently executing programs and because the access to shared data is serialised. In the Client Server Database programming model, output is only contingent on committed state so all speculative execution related to the output values must be committed before output can start.

The Client Server Database model describes how shared state should be restricted so that external entities experience a consistent view of shared state. This is achieved by giving the appearance of a serialised interaction with shared state to any particular external entity by using a database as the exclusive repository of shared state.

A Client Server Database system treats state local to an application and state shared between applications completely differently. State shared between processes is restricted exclusively to values in the database, which can only be accessed through the interface provided by the database, whereas state local to an application can be accessed by the usual memory operations.

In the Client Server Database model all state shared between users is restricted exclusively to the database. Data related to one client is isolated from data related to any other client. Output must be based on committed shared state. Typically, a database server will implement some kind of memory protection or address space restriction to prevent instances of concurrently executing programs affecting each other's execution.

### 2.1.6 Heterogeneous Processors

Message passing is the predominant model for programming heterogeneous Chip Multi-Processors. The message passing model restricts shared state to the internals of the message passing interface. The program must pass all shared values in messages. When message passing is orchestrated, as it is in a parallel processor, it can be a very efficient way of sharing data, but when messages must be

marshalled, as they are in an embedded Chip Multi-Processor, the overheads of routing messages can be very high.

A communications protocol is used to pass messages between processors. A programmer must be careful to abide by the rules of this protocol and handle all conditions relating to the transmission of the message. It is possible to implement layers of abstraction over message passing protocols but the fundamental interaction with the program cannot be abstracted away [Zim81]. The usual approach to programming heterogeneous systems is to avoid sharing any state at all by using a programming language such as Erlang [Arm07]. There is almost universal agreement that concurrent programs for heterogeneous Chip Multi-Processors are difficult to write [DL09].

The reason why Transactional Memory has not been proposed as a technique for making the programming of heterogeneous Chip Multi-Processors easier is that heterogeneous processors do not have mechanisms for ensuring the consistency of shared memory. Heterogeneous Chip Multi-Processors do not implement mechanisms, such as cache coherency, which would allow them to share state.

The solution to the problems of allowing heterogeneous systems concurrent access to shared data are solved by Client Server Databases which are naturally heterogeneous. The mechanisms used to maintain shared state in a database environment could serve as a model for heterogeneous Chip Multi-Processors.

# Bibliography


[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[CGE08] Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: Locking the right path for atomicity. In *CC '08: Proc. International Conference on Compiler Construction*, pages 276–290, March 2008.

[CMCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[DL09] Jack Dongarra and Alexey L. Lastovetsky. *High Performance Heterogeneous Computing*. Wiley-Interscience, New York, NY, USA, 2009.

[Gra02] Steve Graves. In-memory database systems. *Linux J.*, 2002:10–, September 2002.

[HLR10] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[KHLW10] Behram Khan, Matthew Horsnell, Mikel Lujan, and Ian Watson. Scalable object-aware hardware transactional memory. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 268–279, Berlin, Heidelberg, 2010. Springer-Verlag.

[MBL06] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5:17–, July 2006.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[SR01] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. *SIGPLAN Not.*, 36:12–23, June 2001.

[VTG+09] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xCalls: safe I/O in memory transactions. In *EuroSys*, pages 247–260, 2009.

[WA02] Michael Widenius and Davis Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[Zim81] Hubert Zimmermann. The ISO reference model for open systems interconnection. In *Kommunikation in Verteilten Systemen*, pages 39–57, 1981.