Contents

2.2 Parallelism

It is difficult to express an algorithm, within an existing imperative program, in such a way that a computer can execute it concurrently. The key to making this easier is to remove the concept of shared mutable state from both the expression of the program and its execution. This can be achieved by incorporating pure functions and immutable data into the imperative programming paradigm. This relieves the programmer of having to reason about dependencies and coordinate concurrent execution.

Software is becoming more and more complex. Much of this complexity is incidental, arising from the way problems are solved, rather than the problems themselves. Unfortunately, the mechanisms required to utilise the concurrency afforded by Chip Multi-Processors introduce even more incidental complexity. The functional programming and transactional programming paradigms offer ways to reduce the incidental complexity arising from the utilisation of concurrent execution.

The main contribution of this section is to identify concepts fundamental to the support of concurrent programming within the functional and transactional programming paradigms. This section focuses on combining the concepts of pure functions and immutable data within the context of an existing imperative programming language.

2.2.1 Temporal Uncertainty

The imperative programming paradigm does not offer a satisfactory solution to the problem of coordinating the concurrent actions of multiple processors as it relies on the, essentially serial, concepts of impure functions and mutable state.

For many organisations the investment in existing software is too large to contemplate entirely re-writing working programs just to gain a performance benefit from concurrency. Only a small region of a program benefits from concurrent execution. Finding a way to support the concurrent execution of performance-critical regions of existing imperative programs is of great commercial importance.

This thesis focuses on the problem of expressing concurrency, within an existing imperative program, in such a way that a Chip Multi-Processor can obtain speed-up from concurrent execution. Our approach is to combine aspects of functional programming and transactional programming within the imperative

programming paradigm.

The aim is to reduce the incidental complexity introduced into an algorithm when it is expressed in such a way that it can execute concurrently on a Chip Multi-Processor. This incidental complexity is not restricted to the additional code required to allow a routine to execute concurrently. Mechanisms to support concurrency also make it more complex to design, code, test, debug and maintain a concurrent algorithm than an equivalent serial algorithm.

The source of this complexity is the uncertainty about the passage of time perceived by the concurrently executing components and this temporal uncertainty originates from uncertainty about the dependencies between functions and the interleaving of memory operations.

Functional programming overcomes the incidental complexity of determining whether concurrently executing functions are dependent on each other as it emphasises the use of pure functions in which all dependencies are explicit. This raises the question of how to express pure functions in such a way that programmers can incorporate them into existing imperative programs easily?

Section 2.2.3 discusses this problem in detail.

Functional programming eliminates the incidental complexity inherent in the management of mutable shared state. Functional programming emphasises the use of immutable data which can be safely shared between processors. This raises the question of how to express immutable data in such a way that programmers can incorporate it into existing imperative programs easily?

Section 2.2.4 discusses this problem in detail.

Transactional programming reduces the incidental complexity of coordinating concurrent actions between processors. Transactions permit the simultaneous speculative execution of functions in the absence of complete information about their dependencies. This raises the question of how to express Memory Transactions in such a way that programmers can incorporate them into existing imperative programs easily?

Section 2.2.5 discusses this problem in detail.

2.2.2 Minimising Temporal Uncertainty

The problem of supporting concurrency in an imperative programming language can be broken down into the problems of determining dependencies between functions, managing shared state and coordinating concurrent actions. The functional and transactional programming paradigms offer solutions to these problems. Functional programming languages emphasise the use of pure functions and immutable data as a means of identifying dependencies and managing shared state respectively. Transactional programming emphasises the use of transactions as a means of coordinating concurrent actions. However, the concurrency problem is not solved by translating a concept from one programming paradigm into another. Solutions should be found to the problem of balancing concurrent work while ensuring that operations appear to occur in the correct semantic order.

The elimination of impure functions and mutable state permits functional programs to be decomposed into functions which can be evaluated in parallel.

Section 2.2.7 discusses the parallel evaluation of a functional program.

Certainty about the dependencies between functions permit a functional program to be decomposed into functions which can be evaluated speculatively.

Section 2.2.8 discusses the speculative evaluation of functional programs.

There are synergies to be gained by incorporating these concepts into the imperative programming paradigm in an integrated manner. Pure functions, immutable data and Memory Transactions are difficult to incorporate into the imperative programming paradigm, independently. However, each component impose restrictions on the others so the combination is much less complicated to implement and use than the sum of the parts. The imperative programming paradigm can be used in program code that will not be executed concurrently so modifications to existing imperative programs can be restricted to performance-critical regions. The challenge is to combine these concepts in a way that makes concurrent programming easier within the imperative programming paradigm.

The proposal developed in this thesis is to decompose programs into functions acting on immutable data and to execute those functions speculatively as Memory Transactions. In this way the transactional and functional programming paradigms can be combined to support concurrent execution.

2.2.3 Functional Dependencies

A function that uses a value produced by another function is said to be dependent on that function. The dependency implies that the functions should be executed in a particular order called the precedence order of the functions. Functions that are not dependent on each other may be executed concurrently. Precedence is usually a weak ordering offering many opportunities for concurrent execution. When there is uncertainty about the dependencies between functions it is not safe to execute them concurrently but when the uncertainty about dependencies is reduced the opportunities for exploiting concurrency increase.

An impure function is one that does not necessarily produce the same return value each time it is executed with a given set of parameters. Impure functions may have side effects. They read the state of memory in addition to their parameter list and they can modify the state of memory in addition to returning a value. These side effects introduce dependencies between functions which are not expressed in the function's parameter list or return value. The dependencies between functions should be known if they are to be executed concurrently.

Imperative programming languages are not usually expressive enough to allow the identification of all functional dependencies. Consequently, it is often not possible to identify sets of routines that do not contain dependencies and that can safely be executed concurrently.

Functional programming is a style of programming that emphasises the use of pure functions. Pure functions do not have side effects. The dependencies of pure functions are easily determined because they are restricted to their parameters. The effects of pure functions are restricted to the return values of the function.

An expression formed by the composition of pure functions is said to be referentially transparent. A referentially transparent expression corresponds to an expression in pure mathematics; it is a timeless statement of truth.

Pure functions have many advantages over the impure functions typical of imperative programming languages. The use of the functional program style in imperative programs is explored in [HM97]. However, the functional program style is in many ways orthogonal to the style in which imperative programs are written. The difficulty of using pure functions in an imperative context arises because imperative programming languages lack the expressiveness necessary to enforce purity through the use of the type system. Imperative programming languages permit the expression of simple pure functions but do not provide a

suitably powerful mechanism to compose functions while retaining purity.

The advantages of pure functions are not compelling enough to overcome the awkwardness of programming in a functional style within an imperative programming language. However, in the context of concurrent execution, certainty about functional dependencies makes concurrent programming a lot easier.

2.2.4 Mutable Shared State

When a function modifies data that is shared it must ensure that no function executing on another processor is accessing that data at the same moment in time. A function can only be certain about mutable data that is never shared. The conventional approach to ensuring that a function has exclusive access to shared data is to serialise access to it using mutual exclusion. An alternative approach is to eliminate mutable shared data altogether and share only immutable data. Both approaches increase the opportunities for exploiting concurrency by reducing uncertainty about the order of access to shared data.

Imperative programming languages permit mutable shared data in the form of variables, objects and data structures. Shared data cannot be simultaneously modified by multiple processors safely. To prevent simultaneous modification imperative programming languages implement mutual exclusion which serialises the execution of a code section accessing shared data. The association between a serialised code section and the shared data which it protects is a convention. It is not expressed in, and is not enforced by, the programming language.

Functional programming emphasises the use of immutable data. Immutable values cannot be modified once they have been written and can be safely shared between processors without requiring mutual exclusion.

Immutable data can be organised into Immutable Data Structures. Immutable versions of many common data structures are described in the literature [Oka98]. These Immutable Data Structures can have access times and space requirements similar to their mutable counterparts.

Immutable Data Structures have received little attention outside the field of functional programming languages and there are no publicly available libraries of Immutable Data Structures implemented in imperative programming languages. Immutable Data Structure are traditionally regarded as more difficult to implement than their ephemeral counterparts.

The use of immutable data is in many ways orthogonal to the imperative

program paradigm. In general, the use of Immutable Data Structures does not make imperative programming easier and very few imperative programs make use of them. However, in the context of concurrent execution immutable data is much easier to reason about than mutable shared data so the use of Immutable Data Structures makes concurrent programming a lot easier.

2.2.5 Coordinating Concurrent Actions

An algorithm may be decomposed into tasks that can be executed concurrently on multiple processors. The actions of these tasks should be coordinated. However, imperative programming languages do not offer a general solution to the problem of coordinating actions on multiple processors.

Transactional programming is a style of programming that emphasises the use of speculative execution. Transactions permit speculation by allowing their affects to be undone should speculation prove incorrect. Transactions permit the separation of the *actual* order of execution from the order in which operations *appear* to have executed. It is the separation of the actual and apparent order of execution which permits speculation.

Concurrent actions are easier to coordinate if their affects are restricted to transactions. Transactions permit reactive and optimistic coordination, so conflicts can be detected after they happen, and can be corrected. Without transactions coordination must be preemptive and pessimistic, so conflicting events occurring on different processors must be anticipated and avoided.

The support for Memory Transactions within imperative programming languages is discussed extensively in this thesis. A central problem is how to express a Memory Transaction within the imperative programming paradigm without making extensive changes to existing applications?

2.2.6 Previous work

Harris develops a Software Transactional Memory system based on the speculative evaluation of functions in the functional programming language Haskell [HMPJH05]. Haskell prevents a programmer from using impure functions or mutable state so the choice of Haskell as the functional programming language eliminates uncertainty about both functional dependencies and the interleaving of memory operations. Harris describes the desirable properties of the functional

and transactional programming paradigms and attempts to combine them.

Harris describes how uncertainty about the state of concurrent actions can be reduced by speculative evaluation and proposes that Memory Transactions can be supported by a functional programming language. Harris's system permits a programmer to use the *atomic* keyword to describe functions that will be evaluated speculatively as Memory Transactions. Harris was able to demonstrate speed-up from concurrent execution.

Harris's system combines pure functions, immutable data and Memory Transactions to support concurrent execution. Our Transactional Data Structures also combine these elements. However, Harris chooses to regard Memory Transactions as atomic sections, whereas we choose to regard them as speculative access to shared data, because atomic sections make the interaction between a program and an external entity problematic. Brown ascribes much of the difficulty of implementing a concurrent system in a functional programming language to the problem of supporting IO [Bro08].

Harris's system is based on modifying an existing functional programming language and its run-time environment, whereas ours requires no modification to the programming language, compiler or development tool chain.

2.2.7 Parallel Execution of Functional Programs

The evaluation of a functional program is a form of graph reduction in which a function whose parameters are values can be replaced by its returned value. This value becomes the parameter of its parent function in the tree. Eventually, the function at the root of the abstract syntax tree can be replaced by the result of the program. Peyton-Jones describes the process of converting a functional program into an abstract syntax tree and performing the reduction to evaluate it [PJ87].

Figure 2.1 illustrates how a functional program can be expressed as an abstract syntax tree representing the order of precedence of the functions.

Certainty about the dependencies between functions enables programs written in functional programming languages to be evaluated concurrently. The problem of identifying functions, within the abstract syntax tree, that can be evaluated concurrently is the simple one of finding discrete sub-trees.

Figure 2.1 illustrates how discrete the sub-trees of an abstract syntax tree can be evaluated in parallel.

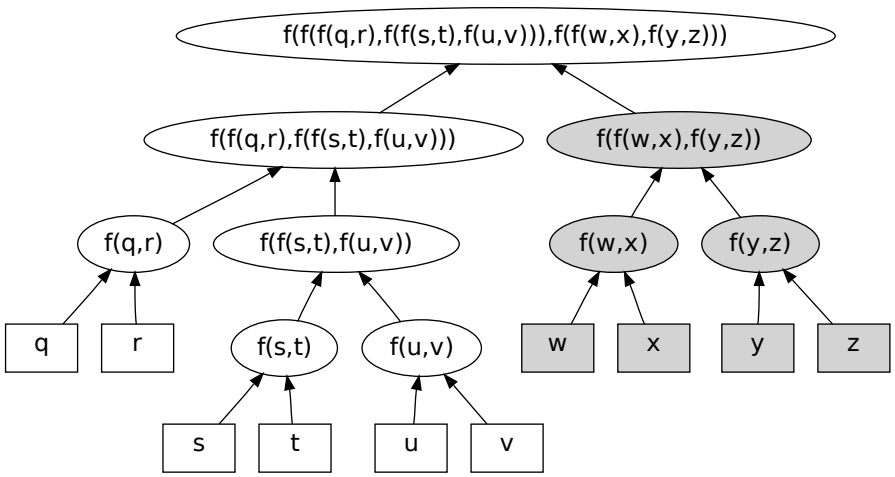

Figure 2.1: **Parallel evaluation of a functional program.** The shaded sub-tree of the abstract syntax tree can be evaluated independent of the white sub-tree.

It is straight-forward to identify discrete sub-trees in the abstract syntax tree. However, the time taken to evaluate a sub-tree is not necessarily related to its size and the problem of statically identifying a set of sub-trees that balance the workload between processors is intractable in the general case. A great deal of research effort has been applied to the balancing problem in specific cases and for some types of parallel work there are sophisticated programming solutions to the balancing problem such as NESL [Ble96]. The balancing problem can also be solved by dynamically identifying and dispatching work to balance the execution between multiple processors.

2.2.8 Speculative Execution of Functional Programs

Certainty about the dependencies between functions enables programs written in functional programming languages to be evaluated speculatively.

A referentially transparent expression can be evaluated speculatively by substituting a value for a sub-expression that has not yet been evaluated. The speculation is that the sub-expression will evaluate to the speculative value. Concurrency is obtained by executing both the sub-expression yielding the value and the expression dependent on the value in parallel. If the speculation is incorrect

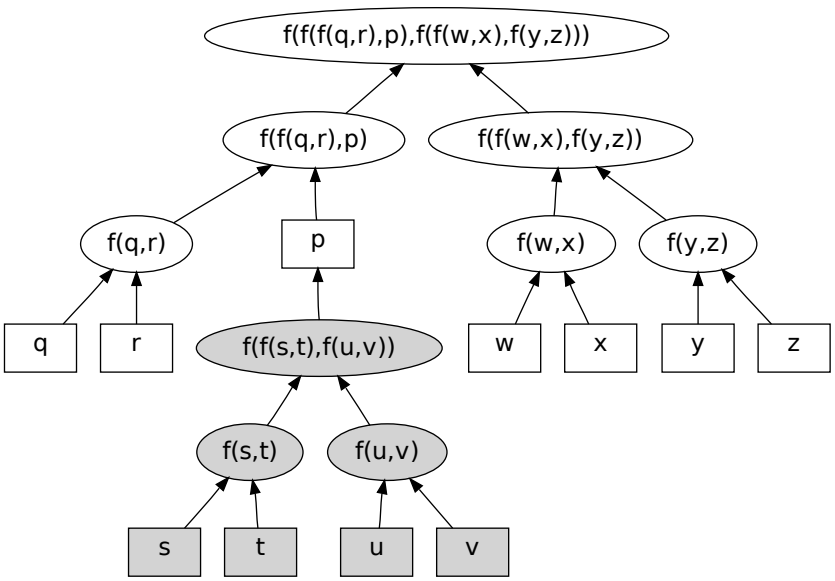

Figure 2.2: **Speculative evaluation of a functional program.** The speculation is that the shaded sub-tree of the abstract syntax tree evaluates to p. The white sub-tree can be evaluated independent of the shaded sub-tree. If the shaded sub-tree evaluates to p then the evaluation of the white tree will be correct.

then the dependent expression can be evaluated again [PJGF96].

Figure 2.2 illustrates the use of value speculation to allow the concurrent execution of a referentially transparent expression. The speculation is that the expression $f(f(s,t),f(u,v)))$ evaluates to p. The value p is used in place of the shaded sub-tree. Both the shaded and unshaded trees can be evaluated in parallel. If the shaded tree evaluates to p then the unshaded tree evaluates to the result of the expression. Otherwise, the unshaded part of the tree must be evaluated again using the actual value of the shaded tree.

Value speculation permits non-discrete sub-trees to be evaluated concurrently. However, it does not contribute a solution to the problem of load balancing.

Bibliography

[Ble96] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39:85–97, March 1996.

[Bro08] Neil Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In P.H. Welch et al., editor, *Communicating Process Architectures 2008*, pages 67–83, 2008.

[HM97] Pieter Hartel and Henk Muller. *Functional C.* Addison Wesley Longman, April 1997.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[Oka98] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, New York, NY, USA, 1998.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[PJGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 1996. ACM.