THE UNIVERSITY OF CHICAGO

PARALLEL FUNCTIONAL PROGRAMMING WITH MUTABLE STATE

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
LARS BERGSTROM

CHICAGO, ILLINOIS
JUNE 2013

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my parents, Patricia and Larry, and my brother, Bjorn, for their unwavering faith in my ultimate success, even when that was not at all clear to me. My wife, Yee Man, provided love and support — both emotional and financial — throughout this process.

My adviser, John Reppy, has been not only a mentor and guide in my journey, but also a collaborator across all levels of detail in our work together. It has truly been a joy to work with him, on both this dissertation and the wide variety of other projects we have done during my tenure.

My other committee members, Matthew Fluet and David MacQueen, provided not only feedback on this work, but on much other work throughout my graduate career. They are two of the most patient people I have ever met, readily making available as much time as I needed to teach, mentor, or review my work.

Without the implementation help, design brainstorming, and moral support of my fellow graduate students, this work could not have been completed. I would especially like to thank Adam Shaw and Mike Rainey, my fellow Manticore brothers-in-arms, for their many hours of collaborations.

My undergraduate adviser, Ian Horswill, provided me with the research experiences that would ultimately weigh heavily in my decision to go to graduate school, though I did disappoint him by pursuing programming languages instead of artificial intelligence.

Finally, I would like to thank my high school programming teacher, Edward Walter (1942–2012). He ignited my interest in programming and continued to fan the flame and support me long after I had exhausted all the resources the school could provide.

The research presented in this dissertation was supported by the NSF, under the NSF Grants CCF-0811389 and CCF-1010568.

ABSTRACT

Immutability greatly simplifies the implementation of parallel languages. In the absence of mutable state the language implementation is free to perform parallel operations with fewer locks and fewer restrictions on scheduling and data replication. In the Manticore project, we have achieved nearly perfect speedups across both Intel and AMD manycore machines on a variety of benchmarks using this approach.

There are parallel stateful algorithms, however, that exhibit significantly better performance than the corresponding parallel algorithm without mutable state. For example, in many search problems, the same problem configuration can be reached through multiple execution paths. Parallel stateful algorithms share the results of evaluating the same configuration across threads, but parallel mutation-free algorithms are required to either duplicate work or thread their state through a sequential store. Additionally, in algorithms where each parallel task mutates an independent portion of the data, non-conflicting mutations can be performed in parallel. The parallel state-free algorithm will have to merge each of those changes individually, which is a sequential operation at each step.

In this dissertation, we extend Manticore with two techniques that address these problems while preserving its current scalability. *Memoization*, also known as function caching, is a technique that stores previously returned values from functions, making them available to parallel threads of executions that call that same function with those same values. We have taken this deterministic technique and combined it with a high-performance implementation of a dynamically sized, parallel hash table to provide scalable performance. We have also added *mutable state* along with two execution models — one of which is deterministic — that allow the user to share arbitrary results across parallel threads under several execution models, all of which preserve the ability to reason locally about the behavior of code.

For both of these techniques, we present a detailed description of their implementations, examine a set of relevant benchmarks, and specify their semantics.

CHAPTER 1

INTRODUCTION

Hardware has shifted over the last few years from mostly sequential to mostly parallel. While the clock speed of top-end Intel processors has remained relatively constant from 2006–2013, the number of parallel processing elements has transitioned from one processor per chip to up to six. Further, GPUs and other acceleration boards are now delivering hundreds of parallel threads of execution on a single expansion card.

Unfortunately, despite the new languages, new language features, and libraries invented to help with parallelism, it is still hard to write general-purpose programs that take advantage of this parallelism. Figure 1.1 depicts the space of language and library features for parallel programming with shared state. In general, languages that are closer to the deterministic end of the spectrum are easier to program but less able to take advantage of the parallel hardware due to the costs of guaranteeing determinism. Languages that are non-deterministic make it hard to reason about the correctness of the programs written in them, but generally provide less restrictions on taking full advantage of the parallel hardware. This tradeoff forces programmers to choose: do I want to have a program that I fully understand, or one that takes advantage of the parallel hardware?

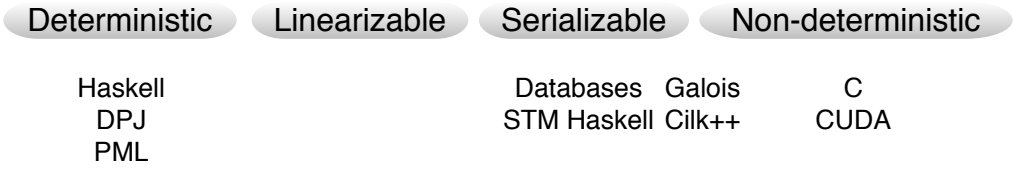


Figure 1.1: The spectrum of parallel languages by deterministic behavior.

There are three points in the determinism spectrum that are well-explored. Languages such as C and CUDA allow complete access to the parallel hardware, though they provide limited mechanisms for ensuring either the correctness or determinism of the programs, making it difficult to reason about them. Deterministic Parallel Java (DPJ) and most of the parallel libraries used in

Haskell, on the other hand, provide deterministic programming models that make it easy to reason about the correctness of the program but which make it difficult to take advantage of the parallel hardware for all problems. Databases, STM Haskell, and Galois provide some points closer to the middle of the space, where there are some guarantees on the semantics of programs, greatly reducing the space of possible executions, but still require the programmer to reason about the execution of the entire program and its library usage to ensure correctness.

1.1 Our approach

Ideally, all programs would be written in a deterministic parallel programming language and they would take full advantage of the hardware. Unfortunately, some problems cannot be efficiently solved in a pure parallel program. These problems have asymptotically faster algorithms that rely on state, forcing a programmer who uses a deterministic parallel language to choose between a slower, state-free algorithm that is parallel and a faster sequential algorithm that uses state.

In this work, we extend our language, Parallel ML (PML), with language features that allow restricted sharing between threads and high performance without the drawbacks of race conditions provided by the other high performance models. Our goal is to add these language features while preserving both the scalable performance and straightforward reasoning about program correctness of PML. We add two language features in this work: memoization, which is deterministic, and mutable state, which is linearizable.

1.1.1 Thesis

Memoization and mutable state extend the Parallel ML language with inter-thread communication in a correct, concise, local, and high-performance manner.

Correctness is provided through an operational semantics that characterizes the full behavior of these features. These extensions remain *concise* by using the traditional language features for

mutable values available in Standard ML and a single additional keyword for memoization. We also do not require or support any additional type annotations, region, or effect declarations.

Our model supports *local reasoning*, which is a term used in separation logic that refers to the ability to reason about the behavior of a function and its usage of the heap in isolation from any other functions — including concurrent functions — that mutate the heap [59]. Providing this property requires that the language and runtime ensure separation. That is, at any point in the execution of the program, there exists a partition of the state such that each concurrent task has private ownership of the portion of the heap that it accesses. The only impure portion of the language that accesses the shared heap and thus requires proof of this property is mutable state.

In both of the execution models describe above, separation holds. In the serial model, there is no concurrent access to the heap, ensuring that any task that accesses the heap has exclusive access to it. In the transactional model, each task performs its updates in isolation from the global heap and only modifies it during transactional commit, ensuring that each task is executed as if it had exclusive ownership of the portion of the heap that it accessed throughout its execution.

Performance is shown through a series of example benchmarks and comparison to other implementations.

1.1.2 Memoization

The first feature is *memoization*, also known as function caching. This technique takes advantage of pure function calls by noting that once a pure function has been invoked with a given set of arguments, all future calls sharing those arguments may simple reuse the first result. This approach has long been used by hand, particularly in dynamically-typed functional languages such as Scheme and LISP. Most efficient solutions to dynamic programming, state search, and SAT/SMT solvers use some variant of result caching to turn exponential problems into polynomial ones. Our design provides this feature as an annotation to function definitions. This feature is deterministic. While much work was previously done in the context of dynamic languages on making memoiza-

tion efficient, our extension of those concepts to a strongly typed, parallel language and runtime is novel [61, 62].

Parallel Fibonacci A simple example of the usefulness of memoization is in the calculation of the Fibonacci numbers, as this program is exponential in the naïve case but linear if each Fibonacci number is calculated only once. In the code below, the Fibonacci numbers are computed in parallel and with memoization.

```
mfun pfib i = (case i
  of 0 => 0
   | 1 => 1
   | n => let
     val (a,b) = (| pfib i-1, pfib i-2 |)
    in
      a+b
    end
   (* end case *))
```

Parallelism is introduced by the parallel tuple construct used in the two recursive calls. Memoization is introduced through the use of the keyword **`mfun`**, which is a hint to the runtime that it might be profitable to cache the results from evaluating the `pfib` function.

Without memoization, the user has two options for implementing the parallel calculation of Fibonacci numbers manually:

- Pass along and return a cache of computed values, merging them after each parallel subtask completes.
- Dramatically transform the program based on deeper understanding of the dependency between tasks so that fewer values need to be recomputed.

The first option requires significantly more work and is particularly unfortunate for this program since so little work (just a single addition) is done at each parallel step. The work done to merge the cache will dominate the addition, even with very efficient operations to merge the data. The second option requires reordering the computation so that parallel tasks are only spawned when the tasks will not repeat a previous computation. In this example, that second option will also remove all of the parallelism, as there are no subcomputations that can be performed in parallel without possibly repeating previous computations.

1.1.3 Mutable state

The second feature is *mutable state*. We have extended our language with the traditional Standard ML mechanisms for mutable storage — the `ref` cell and `array` type. In a sequential context, these features are deterministic. In a parallel context, the allowable interleavings of concurrent expressions that manipulate state determine the semantics. We provide two different execution models for mutable state that affect these interleavings, both of which are a compile-time decision for the programmer and require no additional annotations.

- *Serial* behavior ensures that any two expressions that modify state shared between them will be executed in sequential program order.

- *Transactional* execution also allows multiple expressions that modify state to execute concurrently, but the first expression to complete will have its changes reflected in memory.

The serial execution model is deterministic. The results of a program that uses these features are guaranteed to be the same under a parallel execution as they were in a sequential execution.

The transactional execution model, however, may return not only a different result than the sequential execution in parallel but also may return a different result on each parallel execution. That said, these results are not completely non-deterministic, but are *linearizable*. That is, any two functions called in parallel with one another will appear to the caller as if one of the two of them

ran to completion in isolation of the other, though it provides no guarantee on which one will do so. This model provides the programmer the ability to perform local reasoning, which allows them to reason about the behavior of their function in isolation from any changes that occur in concurrent pieces of code.

The goal of our state design is to remain faithful to stateful programming in Standard ML while at the same time providing both good parallel performance and code understandability. While there has been a wealth of research on stateful programming with locks, transactions, and linearizable data structures, ours is unique in that it does not require any explicit annotations of locking, transaction regions, or effects, unlike previous systems [41, 42].

Best move so far In games such as Pousse, the problem is to come up with the best possible move in a limited amount of time [7]. The inter-thread communication in this type of program is small (update the "best known move"), since the search space does not feature repeated positions that would benefit from memoization and the space is easily subdivided up-front into parallel threads of work. In the example below, the code `update_best` checks the best move so far to determine if it should be updated and, if so, replaces the value in the global mutable state. When called from multiple threads without any protection, there is a race condition if both of them decide to update but the final writer's score is not actually the best. In both of the state models provided by our system, the resulting code will be deterministic, though this is not the case for mutable state in all programs.

- In the *serial* model, concurrent inspections of the state space are not allowed to run in parallel, since there is code that mutates shared state in them. The program execution is sequential.
- In the *transactional* model, the first writer to shared state wins and any other writer will be aborted and retried.

In all cases, the locking and retry operations are generated automatically by the compiler and runtime without user annotation. The following code provides an example of a core update function

that can be executed concurrently from multiple parallel tasks:

```
val best = ref EMPTY
fun update_best(new_move, score) =
  case !best
   of EMPTY => (best := MOVE(new_move, score))
    | MOVE(_, old_score) => (
      if (score > old_score)
      then (best := MOVE(new_move, score))
      else ()
```

Without our implementation of mutable state, in a non-deterministic language the programmer would have to implement locking to update both the best new move and new high score. While not a difficult solution in this case, the programmer is then also responsible for ensuring that any other locks that might be used in conjunction with these are always used in the same order or risk deadlock. Further, for all future changes to the program, any access to the value `best` must occur ether within a properly-locked section or outside of the parallel work sections that update it. None of these issues need to be handled by the programmer in our system.

1.2 Challenge with current deterministic languages

Parallel functional programming languages are a good fit for algorithms that have independent, scalable pieces of work. Since there is no or limited communication between those pieces of work, the compiler and runtime are free to optimize the implementation while still maintaining deterministic behavior. In the presence of communication, though, the compiler and runtime will at least have to accommodate synchronization issues. On the examples mentioned before — dynamic programming, state search, and SAT/SMT solvers — the shared state required for a high-performance implementation requires significant synchronization and coordination in order to provide deterministic executions. For these problems, that level of determinism is *unnecessary*, as the correctness

of their results does not require identical executions and intermediate states. Additionally, these problems might even have multiple correct solutions, all of which are equivalently useful. Determinism is not only overkill for many problems, but results in degraded performance and a more complicated programming model.

1.3 Contributions

The unique contribution of this work is the use of declarative annotations that make it easy for the programmer to use shared state while enabling rich automatic compiler and runtime optimizations. Memoization is particularly well-suited to automatic optimizations when relaxing the need to always cache and return results. If we always try to cache and share results, then inter-processor communication on the shared memory location can cause scalability issues, as seen in the implementation of lazy evaluation in Haskell compilers even when synchronization is removed [36]. Further, under aggressive caching strategies the cache can grow quickly, potentially both using excessive memory for the cache and defeating the weak generational hypothesis that most young objects are not long-lived, resulting in poor garbage collector behavior. Our major contribution in memoization is the efficient implementation of caching with least recently used (LRU) replacement and a growth strategy tuned to avoid excessive growth.

Our model of state does not require any annotations, either of the operations performed by the program or of the desired locking behavior. The removal of these annotations allows us to provide the programmer with the freedom to choose which implementation strategy delivers a good tradeoff of performance and predictability. In addition, our transactional implementation differs from other language implementations that are linearizable by avoiding requiring either explicit use of software transactional memory (STM) libraries or special, safe data structures provided by the implementation. With STM, the programmer needs to both reason about which transactional context or contexts a function can be called from in order to write both correct and high-performance code, since unnecessarily creating nested transactions increases overheads. With implementation-

provided data structures, if the programmer either needs to use multiple of them together or needs a richer data structure, they must either reason globally about the concurrent operations that could be performed or must create their own linearizable data structure. Our approach avoids all of these issues.

Linearizability has mainly been investigated in terms of individual object-oriented classes or functional datatypes and their related operations. Our transactional state model extends this concept to the entire programming language, providing linearizability without user annotations or global reasoning.

1.4 Dissertation overview

The rest of this dissertation elaborates the design, implementation, and evaluation of this thesis.

- Chapter 2 provides the background on existing models for communication between parallel threads, the Manticore system, and Parallel ML.
- Chapter 3 describes the implementation of the memoization feature.
- Chapter 4 similarly details the implementation of state.
- Chapter 5 provides an operational semantics for a reduced version of Standard ML, extended with memoization, `ref` cells, and a single parallel language feature from Parallel ML, parallel tuples.
- Chapter 6 surveys related work.
- Chapter 7 concludes.

CHAPTER 2

BACKGROUND

The initial design and implementation of the Manticore platform excluded shared mutable state at the language level because the programming model is significantly simpler in the absence of that state. For the user, a language that is parallel but where the parallel subtasks cannot communicate eliminates both many correctness errors and race conditions. For the compiler implementer, removing this communication reduces both the analysis burden of safely performing optimizations and the implementation burden associated with efficient synchronization, locking, and memory updates across parallel computations. Our prior work has shown that we can write benchmarks in a pure functional language, Parallel ML (PML), that both provides good sequential performance and that scales well on parallel hardware up to an AMD 48 core machine and an Intel 32 core machine [10].

Unfortunately, there are some algorithms whose efficient parallel implementation requires communication between subtasks. For example, in state search problems communication can reduce a tree of computations to a directed acyclic graph (DAG) through sharing of results. Any parallel algorithm that operates on a large shared data structure, such as triangle mesh tesselation, requires some form of communication in order to update that shared data structure. In these two problems, the use of shared state is often not just constant factors faster than the intuitive parallel implementation, but is asymptotically faster.

The approach we take in this work is to develop mechanisms that extend our programming model with the ability to communicate between subtasks. Because our model lacks explicit threads and do not want to complicate our model with them, these mechanisms are necessarily limited.

In this chapter, first we provide background information on several different implementation models for shared-state parallel programming. Then, we describe our programming model and the extensions we have made, along with how those extensions relate to those implementation models. Finally, we describe the target hardware of our system, on which benchmarks are evaluated in this

work.

2.1 Non-deterministic languages

Languages such as C provide semantic guarantees of correctness or race freedom in the presence of shared memory access between threads only as far as those provided by the hardware memory model. In order to produce programs that are deterministic, there is access to low-level atomic memory update instructions and libraries that provide a variety of lock and barrier instructions [16]. These instructions do not guarantee even race or deadlock freedom, much less determinism. It is the burden of the programmer to write, test, and debug their programs.

Language extensions such as Cilk++ and libraries such as OpenMP do not reduce this burden [12, 19]. While both of these approaches provide a dialect of C that is more well-behaved, it is still the burden of the programmer to both adapt their program to the model required by those approaches and ensure that no other portion of their program or *any library that they use* violates that programming model.

2.2 Deterministic languages

Pure functional languages use stepwise parallelism with merging of intermediate results. The previous implementation of Parallel ML (PML) provides this model, as does the Glasgow Parallel Haskell (GPH) project through its `par` and `pseq` constructs. In these models, pure expressions are performed in a parallel without sharing information, merging the threads of computation at regular intervals to combine intermediate results and then again perform a parallel step of work. This merge can be performed either by joining the parallel threads to collect their intermediate results and then fork threads again or through a communication mechanism such as channels that allows all threads to wait until a defined point in the execution of the program to combine those results. This model works well at low numbers of threads, but suffers from poor scalability due to

the sequential merge operations.

2.3 Atomicity-based language models

Transactional memory provides a model in-between completely non-deterministic programming models and deterministic ones and was inspired by the work in database systems on atomicity [66]. In this programming model, the programmer marks shared data that they want to control the access to as transactional. This data is then read and written with special primitives and wrapped in `begin` and `end` operations. The hardware or software will ensure that either all of the values read and written between those two operations are the same as if they were read and written outside the presence of any other threads. Otherwise, the wrapped region of code is transparently executed again. These systems have been tuned for performance over the last decade, but many programmer burdens remain:

- Overall system performance depends on transaction size, which also will need to vary for different machines.
- In systems that allow data to be accessed outside of transactions, ensuring correctness requires global reasoning about all variables and all possible transaction interactions.
- Individual transactional library settings have different performance based on the ratio of conflicting to non-conflicting transactions, which again may be machine-specific.

The Galois [46] system provides an alternative programming model to explicit transactions that also provides atomic semantics. In this model, experienced developers write custom implementations of objects such as queues and graphs that can be shared across threads but which are safe for concurrent access. This model allows users of the language to write code with atomic semantics without specifying the extent of transactions explicitly. The main challenge, though, is when there is need for an atomic library object that does not exist. For example, if the program requires an

object to be atomically removed from a queue and inserted into a stack, unless the library provides a merged stack/queue object, there will be a window where concurrent threads may experience a state where the object is present in neither the queue nor the stack.

2.4 Manticore

Before describing the memoization and mutable state features, we first describe the design and implementation of the host compiler and runtime, Manticore. The Manticore project began in 2007, with the goal of designing a compiler and runtime for general-purpose applications on multicore processors. In this work, we are focused on the interaction of our language features with the implicitly-threaded parallel features, particularly parallel tuples, parallel arrays, and parallel bindings. Interaction of the features implemented in this work with parallel case and the explicitly-threaded features of PML is left for future work. A more complete discussion of their implementation and the explicitly-threaded features of PML is available in our journal paper [24].

2.4.1 Implicitly-threaded parallelism

PML provides implicitly-threaded parallel versions of a number of sequential forms. These constructs can be viewed as hints to the compiler and runtime system about which computations are good candidates for parallel execution. Most of these constructs have deterministic semantics, which are specified by a translation to equivalent sequential forms [67]. Having a deterministic semantics is important for several reasons:

- it gives the programmer a predictable programming model,
- algorithms can be designed and debugged as sequential code before porting to a parallel implementation, and
- it formalizes the expected behavior of the compiler.

```
datatype tree
  = Lf of int
  | Nd of tree * tree

fun trProd (Lf i) = i
  | trProd (Nd (tL, tR)) =
      (op * ) (|trProd1 tL, trProd1 tR|)
```

Figure 2.1: Tree product with parallel tuples.

The requirement to preserve a sequential semantics does place a burden on the implementation. For example, we must verify that subcomputations in an implicit-parallel construct do not send or receive messages. If they do so, the construct must be executed sequentially. Similarly, if a subcomputation raises an exception, the implementation must delay the delivery of the exception until all sequentially prior computations have terminated.

Parallel tuples

Parallel-tuple expressions are the simplest implicitly-threaded construct in PML. The expression

$$(|e_1, \ldots, e_n|)$$

serves as a hint to the compiler and runtime that the subexpressions $e_1, \ldots, e_n$ may be usefully evaluated in parallel. This construct describes a fork-join parallel decomposition, where up to n threads may be forked to compute the expression. There is an implicit barrier synchronization on the completion of all of the subcomputations. The result is a normal tuple value. Figure 2.1 illustrates the use of parallel tuples to compute the product of the leaves of a binary tree of integers.

The sequential semantics of parallel tuples is trivial: they are evaluated simply as sequential tuples. The sequential semantics immediately determines the behavior of an exception-raising subexpression: if an exception is raised when computing its ith element, then we must wait until all preceding elements have been computed before propagating the exception.

Parallel arrays

Support for parallel computations on arrays is common in parallel languages. In PML, we support such computations using a nested parallel array mechanism that was inspired by NESL [11], Nepal [17], and DPH [18]. A parallel array expression has the form

$$\texttt{[|}e_1\texttt{, ..., }e_n\texttt{|]}$$

which constructs an array of n elements. The delimiters `[| |]` alert the compiler that the e_i may be evaluated in parallel.

Parallel array values may also be constructed using *parallel comprehensions*, which allow concise expressions of parallel loops. A comprehension has the general form

$$\texttt{[|}\ e\ \texttt{|}\ p_1\ \texttt{\textbf{in}}\ e_1\texttt{, ...,}\ p_n\ \texttt{\textbf{in}}\ e_n\ \texttt{\textbf{where}}\ e_f\ \texttt{|]}$$

where e is an expression (with free variables bound in the p_i) computing the elements of the array, the p_i are patterns binding the elements of the e_i, which are array-valued expressions, and e_f is an optional boolean-valued expression that is used to filter the input. If the input arrays have different lengths, all are truncated to the length of the shortest input, and they are processed, in parallel, in lock-step.[1] For convenience, we also provide a parallel range form

$$\texttt{[|}\ e_l\ \texttt{\textbf{to}}\ e_h\ \texttt{\textbf{by}}\ e_s\ \texttt{|]}$$

which is useful in combination with comprehensions. (The step expression "**by** e_s" is optional, and defaults to "**by** `1`.")

Parallel bindings

Parallel tuples and arrays provide fork-join patterns of computation, but in some cases more flexible scheduling is desirable. In particular, we may wish to execute some computations speculatively.

1. This behavior is known as *zip semantics*, since the comprehension loops over the zip of the inputs. Both NESL and Nepal use zip semantics, but Data Parallel Haskell [18] supports both zip semantics and *Cartesian-product semantics* where the iteration is over the product of the inputs.

PML provides the parallel binding form

```
let pval p = e1
in
  e2
end
```

that hints to the system that running e_1 in parallel with e_2 would be profitable. The sequential semantics of a parallel binding are similar to lazy evaluation: the binding of the value of e_1 to the pattern p is delayed until one of the variables in p is used. Thus, if an exception were to be raised in e_1 or the matching to the pattern p were to fail, it is raised at the point where a variable from p is first used. In the parallel implementation, we use eager evaluation for parallel bindings, but computations are canceled when the main thread of control reaches a point where their result is guaranteed never to be demanded.

Parallel case

The parallel case expression form is a parallel non-deterministic counterpart to SML's sequential case form. Parallel case expressions have the following structure:

```
pcase e1 & ... & em
 of π1,1 & ... & πm,1 => f1
  | ...
  | π1,n & ... & πm,n => fn
```

Here both e and f range over expressions. The expressions e_i, which we refer to as the *subcomputations* of the parallel case, evaluate in parallel with one another. Note that **pcase** uses ampersands (`&`) to separate both the subcomputations and the corresponding patterns from one another. This syntax simultaneously avoids potential confusion with tuples and tuple patterns, and recalls the related join-pattern syntax of JoCaml [51].

The $\pi_{i,j}$ in a parallel case are *parallel patterns*, which are either normal patterns or the special *non-deterministic wildcard* `?`. A normal wildcard matches a finished computation and effectively discards it by not binding its result to a name. A non-deterministic wildcard, by contrast, matches a computation (and does not name it) even if it has not yet finished.

2.4.2 Runtime model

Our runtime system consists of a small core written in C, which implements a processor abstract layer and garbage collection. The rest of our runtime system, such as thread scheduling and message-passing, is written in ML extended with first-class continuations and mutable data structures.

Process abstraction Our system has three distinct notions of process abstraction:

1. *Fibers* are unadorned threads of control. A suspended fiber is represented as a unit continuation.

2. *Threads* correspond to the explicit threads of our language. Because threads may execute fine-grain parallel computations, a single thread can consist of multiple fibers running in parallel.

3. *Virtual processors* ("*VProcs*") are an abstraction of a hardware processor. Each VProc is hosted by its own pthread and we use use the Linux processor affinity extension to bind pthreads to distinct cores.

   Each VProc has local state, including a local heap and scheduling queue. A VProc runs at most one fiber at a time, and, furthermore, is the only means of running fibers. The VProc that is currently running a fiber is called the *host VProc* of the fiber.

We have designed our system to minimize the sharing of mutable state between VProcs. We distinguish between three types of VProc state: fiber-local state, which is local to each individual computation; VProc-local state, which is only accessed by code running on the VProc; and global state, which is accessed by other VProcs. The thread-atomic state, such as that of machine registers, is protected by limiting context switches to "safe points" (*i.e.*, heap-limit checks).

Garbage collection Functional languages tend to have high allocation rates and require efficient garbage collectors. We have designed our heap architecture and garbage collector to maximize locality and to minimize synchronization between processors. Our design is based on a combination of Appel's semi-generational collector [2] and the approach of Doligez, Leroy, and Gonthier [21, 20].

The heap is organized into a fixed-size local heap for each VProc and a shared global heap, which is a collection of memory chunks. Following Appel [2], each local heap is divided into a nursery where new objects are allocated and an old-object region. In addition to its local heap, each VProc "owns" a chunk of the global heap. We maintain the invariant that there are no pointers into the local heap from either the global heap or another VProc's local heap. Our runtime system uses four different kinds of garbage collection:

1. *Minor GC* is used to collect the nursery by copying live data into the old region of the local heap. After a minor GC, the remaining free space in the local heap is divided into half and the upper half is used as the new nursery.

2. *Major GC* is used to collect the old data in the local heap. The major collector is invoked at the end of a minor collection when the amount of local free space is below some threshold. The major collector copies the live old data into the global heap (except for the data that was just copied by the minor collector; it is left in the local heap).

3. *Promotion* is used to copy objects from the local heap to the global heap when they might become visible to other VProcs. For example, if a thread is going to send a message, then the message must be promoted first, since the receiver may be running on a remote VProc.

4. *Global GC* is used to collect the global heap. The global collector is invoked when a major collection detects that the amount of data allocated in the global heap has exceeded a threshold. The global collector is a stop-the-world parallel collector. We currently do not attempt to load-balance the global collector; each VProc traces the from-space data that is reachable

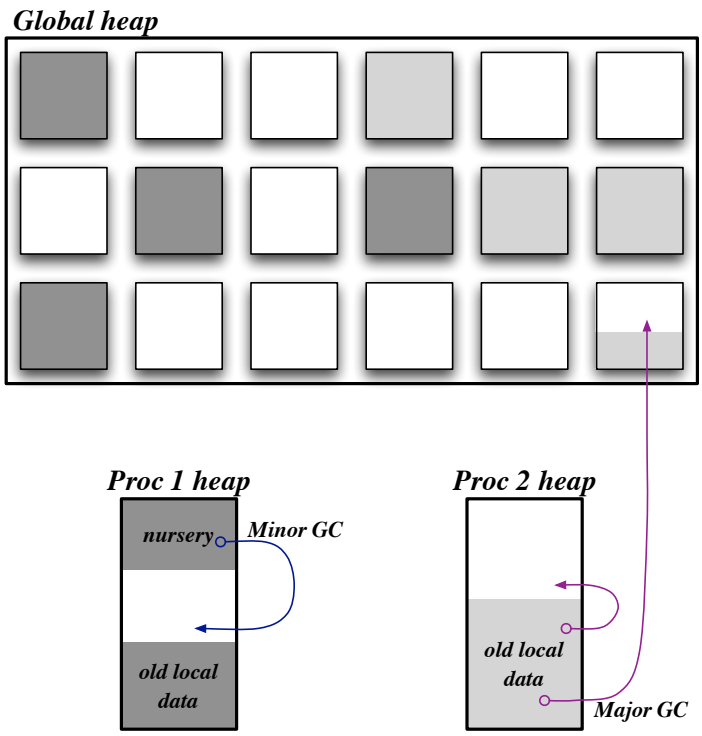


Figure 2.2: Heap architecture for two processors. VProc 1 is running a minor collection, while VProc 2 is running a major collection.

from its roots, stopping when it hits a forward pointer.

Figure 2.2 shows the heap for a two-processor system.

There are two important consequences of our heap design. On the positive side, most garbage-collection activity is asynchronous. The minor collections require no synchronization with other VProcs, and major collections and promotions only require synchronization when the VProc's global chunk becomes full and a new chunk must be acquired. The major drawback of this design is that any data that might be shared across VProcs must be promoted into the global heap. A more detailed description and analysis is available in our workshop paper on the Manticore garbage collector [5].

Inter-VProc communication Because of our heap architecture, code running on one VProc cannot directly access data that lives on another VProc, but we do allow a VProc to *push* a fiber onto another VProc. Each VProc has a *landing pad*, which is a stack that any VProc may push a fiber onto. Each VProc periodically checks its landing pad and puts any fibers there into its scheduling queue. If the originating VProc wants the fiber to be scheduled immediately, it can send a POSIX signal to the other VProc, which will cause it to check its landing pad.

Work stealing Our runtime system uses *work stealing* [15, 35] to load-balance fine-grain parallel computations. The basic strategy is to create a group of workers, one per VProc, that collaborate on the computation. Each worker maintains a deque (double-ended queue) of tasks, represented as continuations, in the global heap.[2] When a worker reaches a fork in the computation, it pushes a task for one branch on the bottom of the deque and continues executing the other branch. Upon return, it pops the a task off the bottom of the deque and executes it. If the deque is empty, then the worker steals a task from the top of some other worker's deque. Because the deques are in the global heap, the tasks must be promoted before they can be pushed onto the deque. These promotions account for almost 50% of the overhead in the work-stealing scheduler.

2.4.3 Compilation model

The compiler operates on the whole program at once, reading in the files in the source code alongside the sources from the runtime library. As covered in more detail in an earlier paper [23], there are six distinct intermediate representations (IRs) in the Manticore compiler:

1. Parse tree — the product of the parser.

2. AST — an explicitly-typed abstract-syntax tree representation.

3. BOM — a direct-style normalized λ-calculus.

2. Our notion of a task is similar to GHC's notion of a *spark* [53].

4. CPS — a continuation passing style λ-calculus.

5. CFG — a first-order control-flow-graph representation.

6. MLTree — the expression tree representation used by the MLRISC code generation framework [27].

All of the features described in this work are implemented as a combination of new libraries, runtime features, and extensions of the frontend phases of the compiler — the Parse Tree, AST, and BOM representations. No modifications were required to the backend.

2.5 Hardware

In addition to the ongoing increase in the core count of processors, modern servers also have between 2 and 4 processors. Modern multi-processor systems use a non-uniform memory architecture (NUMA), where each processor is attached to a private bank of memory and requests for data in those private banks from another processor are serviced via the interconnect between the processors [9]. This design presents an additional challenge, as poor allocation behavior can place all of the frequently-used data on a single NUMA node's memory bank and then the requests from the other processors may saturate the interconnect and result in poor program performance where an alternative allocation strategy may have not resulted in this bottleneck. Further complicating efficient use of these systems, the default allocation policies for physical memory pages are also dynamic. The two most popular policies are *interleaved* and *first-touch*. In the interleaved policy, memory pages are allocated in a round-robin strategy around the processors. In the first-touch policy, memory pages are allocated on the closest physical memory bank to the executing code that has unallocated pages. In our system, we rely on the first-touch policy (which is also the default behavior) and have engineering our system to attempt to preserve locality. This work is the first attempt to implement a parallel hash table that addresses these NUMA issues.

CHAPTER 3

MEMOIZATION

Memoization, also known as function caching, is a technique that caches previously computed results to avoid recomputing them unnecessarily. This optimization exploits the fact that a pure function will always return the same result when called with the same arguments. The tradeoff, though, is that in order to save time by avoiding recomputation, the system must now manage the cache of values, which costs both time and space. The space cost is associated with the storage of the cache of values in memory. The time costs are both from looking up entries in the cache to see if they have already been computed and in managing the cache itself. For example, the runtime may need to to dynamically grow or shrink the size of the cache.

While memoization has long been used in sequential programming languages to avoid sequential redundant computations, we are introducing it to a *parallel* programming language in order to also avoid redundant computations across separate parallel threads. Without the explicit addition of this feature to the language and runtime, parallel threads could each memoize the results of their computations, but would have to join the parallel threadsand then perform a sequential merge of their results before again forking parallel work to share results across those parallel threads. This feature also preserves deterministic behavior, so long as the memoized function is pure. Since the result of the program with memoization is the same as the result of the program without memoization, reasoning is easy for the programmer.

This feature is designed for use in portions of the program where both recomputation costs are high and restructuring the program to explicitly share results would either be unwieldy or result in a loss of parallelism. For example, in Section 1.1.2, we presented the parallel Fibonacci algorithm. In this case, recomputation of previously computed results is expensive and restructuring the computation to share prior results produces a program that is sequential. In Section 3.7.2, we evaluate this benchmark and our implementation of memoization in more detail.

A more compelling example is game search. In this example, multiple parallel threads search-

ing for the best move may evaluate the same game configuration repeatedly if the result of evaluating those configurations is not shared. In Section 3.7.4, we provide a search problem as a benchmark and show how memoization allows the programmer to share evaluation results across parallel threads without either synchronizing the threads or resorting to non-deterministic programming.

We have added memoization to our programming model through an extension of the syntax of the **`fun`** keyword in Standard ML. As in the work on selective memoization by Harper *et al.* [1], we use the keyword **`mfun`** to mark functions that the programmer would like memoized. Unlike in that prior work, our system does not guarantee memoization but, similarly to the parallel language features in Parallel ML, this annotation is a hint to the system that memoization may be profitable for a given function. Since the results of memoization on the evaluation of pure functions are equivalent to the function evaluation without memoization, our runtime system is provided additional freedom to perform optimizations that reduce overheads that could otherwise limit parallelism.

In this chapter, we first describe and explore the translation of the surface language implementation of memoization to library calls. Then, we explore implementation alternatives for those library calls, and finally evaluate the performance of these alternatives with some benchmarks.

3.1 Core-PML with memoization

Figure 3.1 contains the base language that we will use to explain the semantics of this work. This language is a core subset of the Parallel ML (PML) language, extended with memoization and mutable state. Variables, constants, functions, application, case statements (with exactly two branches), and binding behave as in Standard ML. The parallel tuple (pair) form evaluates a two expressions in parallel. In PML the evaluation steps for these two expressions have an unspecified interleaving. Similarly, the parallel binding form evaluates an expression in parallel with the body of the binding form, waiting for that parallel expression to complete when its bound variable is accessed in the body. Memoization is provided through a new keyword, **`mfun`**, which otherwise has the same syntax as a normal function declaration.

e	::=	x	variables
	|	b	constants
	|	`()`	unit value
	|	**SOME** e	option type with value
	|	**NONE**	option type with no value
	|	**mfun** $f\ x = e_1$ **in** e_2	memoized functions
	|	**fun** $f\ x = e_1$ **in** e_2	functions
	|	$e_1\ e_2$	application
	|	**case** e_1 **of** $p_1 \Rightarrow e_2$ **|** $p_2 \Rightarrow e_3$	case analysis
	|	**(|** e_1, e_2 **|)**	parallel tuples
	|	**let** $p = e_1$ **in** e_2	bindings
	|	**plet** $p = e_1$ **in** e_2	parallel bindings
p	::=	x	
	|	b	
	|	`()`	
	|	(p_1, p_2)	
	|	**SOME** p	
	|	**NONE**	

Figure 3.1: Source language with memoization

3.2 Translation to implement memoization

We translate the language shown in Figure 3.1, which includes the **mfun** keyword, with the translation shown in Figure 3.2. This translation replaces the **mfun** definition with a **fun** definition and calls to library functions that implement the caching behavior. These function calls consist of:

- The allocation and creation of the memoization table via the `mktable` function.
- A function, `hash`, that maps values of the parameter type down to an integer suitable for lookup in a generic hashtable. In the current implementation, we restrict the function domain to those with a single integer parameter.
- A search function, `find`, that looks up a value in the memoization table.
- An update function, `insert`, that makes a new cached value available for future calls in the memoization table.

$$
\begin{array}{rcl}
\mathcal{T}[\![x]\!] & = & x \\
\mathcal{T}[\![b]\!] & = & b \\
\mathcal{T}[\![\texttt{mfun}\ f\ x = e_1\ \texttt{in}\ e_2]\!] & = & \texttt{let}\ table\ =\ mkTable()\ \texttt{in} \\
 & & \texttt{fun}\ f\ x = \\
 & & \quad \texttt{let}\ hashed\ =\ hash\ (x)\ \texttt{in} \\
 & & \quad \texttt{let}\ entry\ =\ find\ (table, hashed)\ \texttt{in} \\
 & & \quad \texttt{case}\ entry \\
 & & \quad\quad \texttt{of SOME}\ v\ \Rightarrow\ v \\
 & & \quad\quad \texttt{| NONE}\ \Rightarrow \\
 & & \quad\quad\quad \texttt{let}\ result\ =\ \mathcal{T}[\![e_1]\!]\ \texttt{in} \\
 & & \quad\quad\quad \texttt{let}\ ()\ =\ insert\ (table, hashed, result)\ \texttt{in} \\
 & & \quad\quad\quad\quad result \\
 & & \texttt{in}\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2]\!] & = & \texttt{fun}\ f\ x = \mathcal{T}[\![e_1]\!]\ \texttt{in}\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![e_1\ e_2]\!] & = & \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\ \texttt{(|}e_1, e_2\texttt{|)}\ ]\!] & = & \texttt{(|}\mathcal{T}[\![e_1]\!], \mathcal{T}[\![e_2]\!]\texttt{|)} \\
\mathcal{T}[\![\texttt{let}\ p\ =\ e_1\ \texttt{in}\ e_2]\!] & = & \texttt{let}\ p\ =\ \mathcal{T}[\![e_1]\!]\ \texttt{in}\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\texttt{plet}\ p\ =\ e_1\ \texttt{in}\ e_2]\!] & = & \texttt{plet}\ p\ =\ \mathcal{T}[\![e_1]\!]\ \texttt{in}\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\texttt{ref}\ e]\!] & = & \texttt{ref}\ (\mathcal{T}[\![e]\!]) \\
\mathcal{T}[\![\texttt{!}e]\!] & = & \texttt{!}(\mathcal{T}[\![e]\!]) \\
\mathcal{T}[\![e_1\ \texttt{:=}\ e_2]\!] & = & (\mathcal{T}[\![e_1]\!])\ \texttt{:=}\ (\mathcal{T}[\![e_2]\!])
\end{array}
$$

Figure 3.2: Translation to remove memoization.

3.3 Memoization table interface and behavior

As the previous section implies, the performance of memoization is dominated by the implementation of the memoization table. Over the next several sections, we investigate several alternative implementations of the memoization table that explore different points in the design space.

Each of the implementations of the memoization table implement the same signature:

```
signature MEMO_TABLE =
 sig
```

```
    type 'a table

    val insert : 'a table * int * 'a -> unit
    val find :  'a table * int -> 'a option
end
```

As a simplification for the implementation, this signature currently restricts the domain of hash keys to integers. In Section 7.1.1, we discuss extending this mechanism to handle more arbitrary types and the case where hash keys collide.

Since memoization is an optimization, the implementation requirements are looser than those of a traditional hash table, which must be safe for concurrent access [65]. In particular, the implementations described in this chapter may take advantage of the following properties:

- Insertions need not be visible to subsequent find operations, even within the same function that performed the insertion.
- There is no guarantee which insertion will win when two are performed simultaneously.

Both of these properties are safe because memoization is an optimization that relies on the purity of the function whose values are cached. In the case where a function is evaluated even though a cached result should have been available, the resulting value from the computation is the same as if the cached result had been found and returned. These properties remove all need for atomic operations or locking and greatly simplify the implementation, at the cost of unpredictable execution behavior.

The `insert` function adds an entry to the table in the slot specified by the integer. The `find` function looks up an entry in the table. Allocation of the memoization table implementations varies for each of the different structures, depending on their configuration space (e.g., starting size, maximum size, etc.).

3.4 Full size memoization table

When designing a memoization table that maps from integer hash keys to arbitrary values, one simplifying design is to assume that there is sufficient space to store all of the hash keys in memory and that the associated space cost is not prohibitive. While clearly not a design that will scale to all problems, this approach allows us to investigate some performance tradeoffs in data layout in isolation. This design requires problems whose keys are in a range that fits into memory and where there are no collisions.

If we do nothing special but simply allocate a large array to hold all of the values, then the default behavior of the memory system will be to physically assign an underlying memory page to each block of that array on the associated memory subsystem of the node that first touches it [9]. As mentioned in Section 2.5, when there are multiple NUMA nodes in the system, this allocation behavior can result in program slowdowns due to bandwidth saturation if most of the values used are located on only one NUMA node.

There are two obvious alternative designs for a hash table that is memory system aware. First, we could *distribute* the data at each NUMA node, caching only values discovered at that node. Alternatively, we could *partition* the data, so that both the hash values are split across more partitions and we can support larger table sizes than the simple single array approach.

3.4.1 Simple hash table

The default memoization table is a datatype consisting of an integer that counts the maximum size of the table and an array that holds pointers to the values. To distinguish unset entries, we store an `option` type, using an initial `NONE` value in all slots to represent an unset entry. The datatype is defined below:

```
type 'a table = int * 'a option Array.array
```

A picture of the representation in-memory is shown in Figure 3.3. The maximum size — in this example, six — is stored in a tuple alongside a pointer to the underlying array. The array then can either contain the in-place value `NONE`, indicating that no element has yet been stored there, or a pointer to a value wrapped in the constructor `SOME`. Unfortunately, due to representation limitations, our runtime cannot currently store simple values (such as an integer wrapped in an `option`) inline in the array.

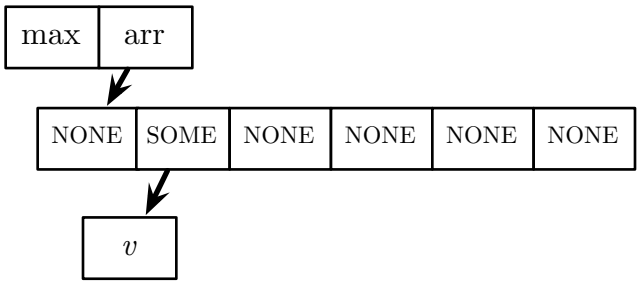


Figure 3.3: The in-memory representation of the simple memo table approach.

Insert and lookup are implemented as direct references into this array, as shown in Figure 3.4. This implementation is designed to support only hash key values that are within the allocated size of the memoization table, so it raises an exception if a key greater than that allocated size is provided.

```
fun insert ((max, arr), key, item) = (
  if (key >= max)
  then raise Fail "Index out of range"
  else ();
  Array.update (arr, key, SOME item))

fun find ((max, arr), key) = (
  if (key >= max)
  then raise Fail "Index out of range"
  else ();
  Array.sub (arr, key))
```

Figure 3.4: Insertion and lookup in a simple full-sized memoization table implementation.

3.4.2 Distributed

In the distributed memoization implementation, each NUMA node in the system has a separate copy of the simple memoization table described previously. This implementation introduces one extra level of indirection (shared among all of the nodes) to retrieve the target table and removes some sharing, but also reduces the amount of memory traffic between NUMA nodes because a given node will never look up values from another NUMA node. The datatype extends the previous implementation by creating an array of the memoization tables. The datatype is defined below:

```
type 'a table = int * 'a option Array.array option Array.array
```

Allocation of the individual memoization tables is done lazily. This approach has two benefits. First, it avoids creating any memoization tables that will never be accessed. More importantly, by waiting for the first access to perform the initialization, we ensure that memoization table will be located on a physical memory page close to the thread that accesses it, as described in Section 2.5, rather than allocating it in one local to the node that happened to allocate the memo table. In Figure 3.5, there is an example memory layout of the data structure used in a four-node machine (similar to the benchmark machine described in Section 3.7.1). Two of the nodes have accessed the memo table, causing the underlying arrays, again of size six, to be created. The second node has computed and stored the second value, but that value will not be made available to processors on other nodes.

The `insert` function must first check to see if the per-node array has already been created. If it has not, the code creates it and initializes the value to `NONE`. This initialization step, since it touches all of the memory pages of the array, forces them to be allocated on the physical memory bank located adjacent to the currently executing processor. The `find` operation simply checks the appropriate key within per-node array, if that array exists. The corresponding PML code is in Figure 3.6.

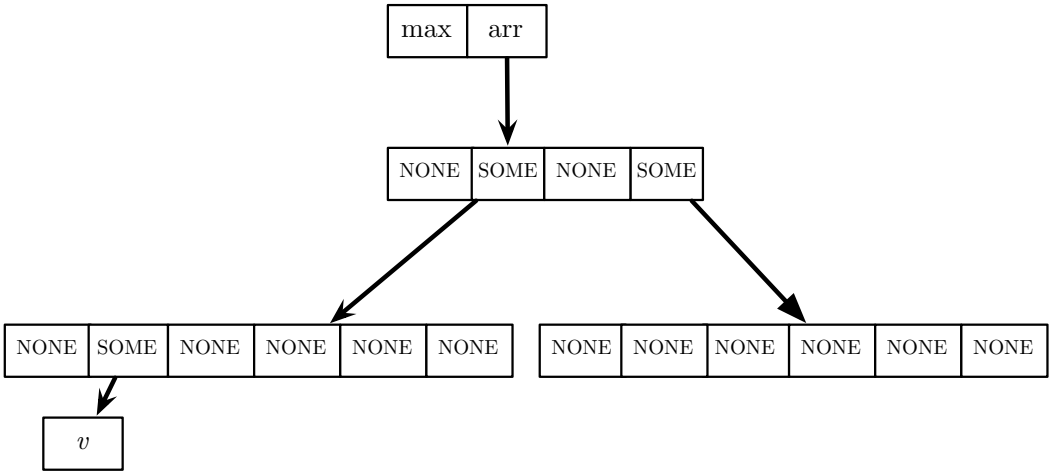


Figure 3.5: The in-memory representation of the distributed memo table approach.

3.4.3 Partitioned

The final full-sized table implementation partitions the values into chunks, to test the performance when we explicitly balance those chunks across the NUMA nodes in the system. This implementation is nearly identical in representation to the distributed implementation except that it also has an integer to track how many elements are in the per-partition arrays. The datatype is defined below:

```
type 'a table = int * int * 'a option Array.array option Array.array
```

The partitioned implementation cycles each consecutive hash value across the separate arrays in a round-robin fashion. So, on a system with four partitions and with the six entries from the previous examples, the layout would be as shown in Figure 3.7. The value stored, whose hash value corresponds to the second index, is now in the first entry of the second partition. Unlike the distributed design, that value is available to processors on all nodes.

Insertion and lookup both first compute the partition by dividing the hash value by the size of the partition (`leafSize` in the code below). Then, the individual slot used for the hash value is just the remainder when divided by the size of the partition. The implementation is shown in Figure 3.8.

```
fun insert ((max, arr), key, item) = (
  if (key >= max)
  then raise Fail "Index out of range"
  else ();
  case Array.sub (arr, VProcUtils.node())
   of NONE => (
        let
          val newarr = Array.array (max, NONE)
          val _ = Array.update (newarr, key, SOME item)
        in
          Array.update (arr, VProcUtils.node (), SOME newarr)
        end)
    | SOME internal => (
        Array.update (internal, key, SOME item)))

fun find ((max, arr), key) = (
  if (key >= max)
  then raise Fail "Index out of range"
  else ();
  (case Array.sub (arr, VProcUtils.node())
    of NONE => NONE
     | SOME internal => Array.sub (internal, key)))
```

Figure 3.6: Insertion and lookup in a full-size distributed memoization table implementation.

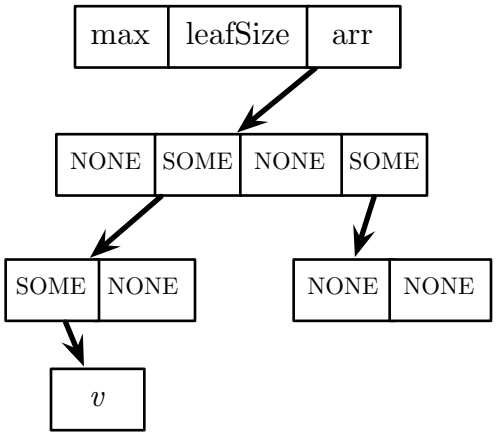


Figure 3.7: The in-memory representation of the partitioned memo table approach.

```
fun insert ((max, leafSize, arr), key, item) =
  if (key >= max)
  then raise Fail "Index out of range"
  else ();
  case Array.sub (arr, key div leafSize)
   of NONE =>
       let
           val newarr = Array.array (leafSize, NONE)
           val _ = Array.update (newarr, key mod leafSize, SOME item)
       in
           Array.update (arr, key div leafSize, SOME newarr)
       end
    | SOME internal =>
        Array.update (internal, key mod leafSize, SOME item)

fun find ((max, leafSize, arr), key) =
  if (key >= max)
  then raise Fail "Index out of range"
  else ();
  case Array.sub (arr, key div leafSize)
   of NONE => NONE
    | SOME internal => (Array.sub (internal, key mod leafSize))
```

Figure 3.8: Insertion and lookup in a full-size partitioned memoization table implementation.

3.5 Limited-size memoization table

It is often neither possible nor desirable to allocate sufficient space to memoize every potential computed value. Some problems, such as the 0-1 knapsack problem described in Section 3.7.3, use space that is quadratic in the size of the inputs. For those problems, it is not possible to retain all computed values except for very small input sizes.

Large tables also increase the memory pressure on the garbage collector, increasing the amount of memory that must be copied and scanned during each global collection. Further, if values are cached past their useful lifetime, then all of that space is unnecessary overhead.

3.5.1 Fixed size

The fixed-size table extends the partitioned approach described in Section 3.4.3, but contains additional support for collisions in the memoization table, where two different hash values map to the same storage location due to a hash value space that is larger than the number of unique locations in the memoization table. We have chosen this implementation strategy because the partitioned approach provides better performance than the distributed approach at larger dataset sizes, as described in Section 3.7.2.

As in the partitioned implementation, indexes are cycled between the memoization table entries for each partition. But, rather than guaranteeing that there is sufficient space for all entries, we instead use a fixed number of elements combined with a finite *bucket* size for each element, to handle collisions. The entries in this table thus need to maintain not only the corresponding value, but also the original key (since many map down to the same element) and an additional piece of data to help decide which entry to discard if the finite bucket is full. This additional piece of data is currently a timestamp that corresponds to either the insertion time of the entry or the last time that it was retrieved, whichever is more recent. These entries use the following type, where the first element is a long integer used for the timestamp, the second is the key, and the final element

is the value:

```
datatype 'a entry = ENTRY of long * int * 'a
```

The datatype for the fixed-size table itself is defined below:

```
(* (Number of nodes, elements per node,
    buckets per element, array of arrays) *)
type 'a table = int * int * int *
  'a entry option Array.array option Array.array
```

Again on a system with four partitions and with the six entries from the previous examples, the layout would be as shown in Figure 3.9. The value stored, which corresponded to the second global element, is now in the first entry of the second partition and is available to processors on other nodes. The *time* value in the entry corresponds to the long value from the `ENTRY` datatype above and records the age of the entry. This value is used in the collision-handling policy.

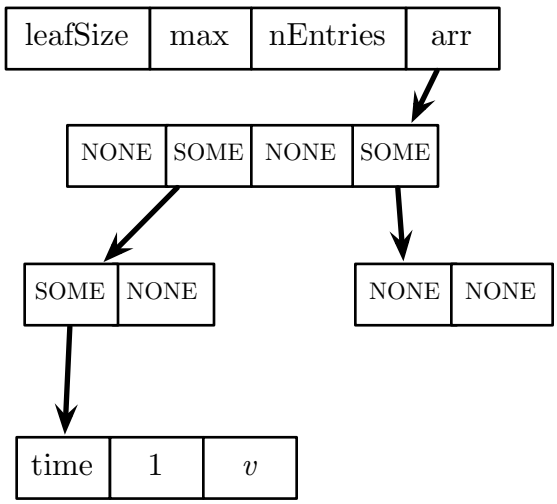


Figure 3.9: The in-memory representation of the fixed-sized partitioned memo table approach.

3.5.2 Replacement policy

Since there are fixed numbers of buckets, a replacement policy is needed to determine which value stored in a full bucket of a memo table should be overwritten when a new entry needs to be stored. In this design, we remove the oldest element in the bucket, where the age of an element is the newer of:

- The time at which it was inserted into the memo table.
- The time at which it was last found and returned.

3.5.3 Finding an entry

The strategy listed above provides a fairly straightforward implementation of insertion, as shown in Figure 3.10. First, we find the corresponding partition for the key by computing `key mod nodes`. Then, starting at the entry within that node's table, we search through them to find the requested key. In the event that the element is found, its entry is updated with the new current time.

3.5.4 Insertion and replacement policy

Insertion is more complicated than lookup, owing to the need to replace the oldest element currently in the memo table. As shown in Figure 3.11, we first create the entry, then ensure that partition-specific array has been created. After looping through all of the non-matching elements corresponding to that key, we then update the oldest element of the array to point at the entry.

3.5.5 Improved hashing

One potential problem is that for certain table sizes and hash value domains, there is an unevenly distributed chance of hash collisions. For example, if the table is of size 6 but the data is of size 9, then items 0-3 and 7-9 will collide, but items 4-5 will not. Further, if the access patterns are

```
fun find ((leafSize, max, nEntries, arr), key) =
  case Array.sub (arr, (key mod max) div leafSize)
   of NONE => NONE
    | SOME internal => (
      let
        val startIndex = (key mod leafSize) * nEntries
        fun findEntry (i) = (
          if (i = nEntries)
          then NONE
          else (
            let
              val e = Array.sub (internal, startIndex + i)
            in
              case e
               of NONE => findEntry (i+1)
                | SOME (ENTRY(_, key', value)) =>
                  if key' = key
                  then (Array.update (internal, startIndex+i,
                          SOME (ENTRY(Time.now(), key', value)));
                        SOME value)
                  else (findEntry (i+1))
          end))
      in
          findEntry 0
      end)
```

Figure 3.10: Lookup in a fixed-size partitioned memoization table implementation.

regular and close to the size of the table, then it is possible that very little of the memoization table will be touched at a given time, but there will still be a large number of collisions.

A common method of addressing this problem is to implement a hashing function of the following form:

$$h(K) = c * K \bmod P$$

where c is a large constant and P is a large prime number. This method (known as the *division* method) has been shown effective in practice [49], though some choices of constant and prime will work better than others on particular data sets. Other implementations of hashing also use this approach [32, 48], and we evaluate its effectiveness in our implementation in the experiments in

Section 3.7.

```
fun insert ((leafSize, max, nEntries, arr), key, item) = let
  val age = Time.now()
  val new = ENTRY (age, key, item)
  val subarray = (
    case Array.sub (arr, (key mod max) div leafSize)
     of NONE =>
       let
         val newarr = Array.array (leafSize * nEntries, NONE)
         val _ = Array.update (arr, (key mod max) div leafSize,
                               SOME newarr)
       in
           newarr
       end
      | SOME arr => arr)
  val startIndex = (key mod leafSize) * nEntries
  fun insertEntry (i, oldestTime, oldestOffset) =
    if i = nEntries
    then (Array.update (subarray, startIndex + oldestOffset,
                        SOME new))
    else (
      case Array.sub (subarray, startIndex + i)
       of NONE => (Array.update (subarray, startIndex + i,
                                 SOME new))
        | SOME (ENTRY (t, _, _)) =>
          if t < oldestTime
          then insertEntry (i+1, t, i)
          else insertEntry (i+1, oldestTime, oldestOffset))
in
  insertEntry (0, Int.toLong (Option.valOf Int.maxInt), 0)
end
```

Figure 3.11: Insertion into a fixed-size partitioned memoization table implementation.

3.6 Dynamically sized memoization table

Even if the performance of the fixed-sized tables can be made acceptable, the problem still remains of needing to tune the size of the table to both individual benchmarks and input data sets. Therefore, the final design we investigate is based on work originally done on dynamic hash tables by Larson [48]. Similar to the partitioned implementation in Section 3.4.3, we start with an array of pointers to individual partitions. In this case, however, only one partition is allocated initially. Keys are hashed using the hashing function described in Section 3.5.5 and then found in the table by taking the hash value modulo the total current capacity. When the table exceeds a fullness threshold, we allocate another segment and increase the total available capacity, moving elements in an on-demand basis from the bucket they previously were mapped to into the new one. In order to distinguish empty elements from uninitialized ones, we have changed the `entry` datatype to use multiple constructors, rather than simply wrapping it in the `option` type. There are now two empty states, `UNINIT` and `INIT`. The `ENTRY` constructor again has a first element that is a long integer used for the timestamp, the second is the key, and the final element is the value:

```
datatype 'a entry = UNINIT | INIT | ENTRY of long * int * 'a
```

The datatype for the dynamically sized memoization table itself is defined below. Here we need to track the current number of segments, a fullness threshold, and a pointer to the actual underlying array of segments.

```
(* (number of segments, current status, array of segments) *)
type 'a table  = int Array.array * int Array.array *
  'a entry Array.array option Array.array
```

Using our example from before, we have the layout shown in Figure 3.12. The segment array will always be filled in from left to right as the individual segments fill. All elements of a segment that have not yet been touched will be set to the value `UNINIT`.

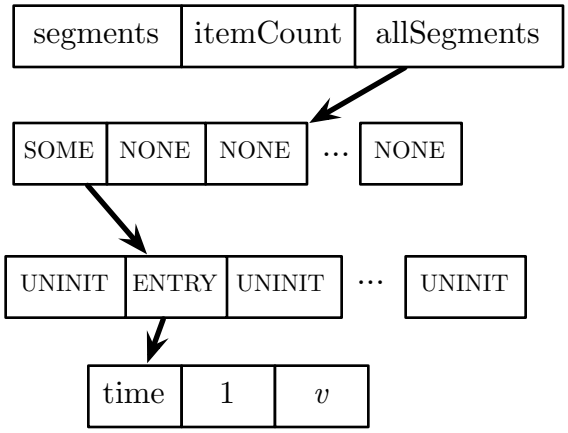


Figure 3.12: The in-memory representation of the dynamically-sized memo table approach.

3.6.1 Increasing capacity

In this design, segments are of a fixed size (we use 200,000 elements) and the table is grown by one segment each time that the fullness threshold is reached. The best-performing design, shown in Figure 3.13, uses a fullness threshold that is based on the total number of collisions on a per-NUMA node basis, allocating a new segment and clearing all of the collision counts. A collision is an instance where an insertion was performed but there was insufficient space in the memoization table and so a previously memoized result was discarded. We also investigated several other strategies, whose performance is detailed in the context of a benchmark in Section 3.7.3. These strategies include:

- Tracking the global number of entries in the table
- Tracking the number of entries on a per-NUMA node basis
- Tracking the number of entries on a per-NUMA node basis with padding
- Tracking the global number of collisions
- Tracking the number of collisions on a per-NUMA node basis

- Tracking the number of collisions on a per-NUMA node basis with padding

The design tension in this implementation is the tradeoff between tracking a quickly-updating value across many processors versus tracking a representative number that we can reference independently. The padding option involves ensuring that the location where the number of entries or collisions are tracked fits within its own cache block, to avoid false sharing between processors.

```
fun growIfNeeded (segments, itemCount, allSegments) = let
  val segmentCount = Array.sub (segments, 0)
in
  if ((maxCollisions < (Array.sub (itemCount,
                                   VProcUtils.node() * padding)))
      andalso segmentCount < maxSegments)
  then let
         val segmentCount = segmentCount +1
         val newSize = (capacity segmentCount -
                        capacity (segmentCount-1))
                       * buckets
         val new = Array.array (newSize, UNINIT)
         val _ = Array.update (allSegments, segmentCount-1,
                               SOME new)
         fun clearCollisions i =
             if i < 0
             then ()
             else (Array.update (itemCount, i*padding, 0);
                   clearCollisions (i-1))
         val _ = clearCollisions (VProcUtils.numNodes() - 1)
       in
         Array.update (segments, 0, segmentCount)
       end
  else ()
end
```

Figure 3.13: Increasing the capacity of a dynamically sized memoization table implementation.

3.6.2 Initializing a segment and its buckets

Each segment has many buckets, each of which holds entries. Due to the dynamically growing nature of this implementation, before we can perform a `find` or `insert`, we first need to ensure

that a segment is initialized. In the first segment, all buckets are initialized by default. In any additional segment, however, buckets begin in the uninitialized state. This engineering decision was made because when the memoization table grows in size, some of the previously stored entries may now belong to a new bucket. For example, given an initial capacity of 4, element 5 would be hashed to bucket 1. But, once the capacity is increased to 8, that same element now belongs in bucket 5. The decision is therefore whether to move items aggressively or lazily. Following the example of actual implementations, we lazily initialize buckets [32, 65]. The code in Figure 3.14 lays out the procedure that we follow. If we are in the first segment, we assume that the item is initialized. If we are in a subsequent segment, then we first perform an initialization of the bucket that we need to split items from at the prior size. Once it is initialized, we scan the entries for pre-existing items that hash to the new location and copy them as appropriate. In the event that there are no items, we mark the new bucket initialized.

This strategy is more like the original strategy proposed by Larson [48] than the more modern work in Icon [32]. Icon was designed to handle very large numbers of tables and limited available memory, which led to a design with smaller starting segment sizes that grew through doubling. The cost of this approach, unfortunately, is significant additional copying at small sizes. Since memory is less of a concern in our context than theirs — both because memories are larger and these tables are limited to one per static function definition — we have elected to use larger, fixed-sized segments.

3.6.3 Finding an entry

With the exception of ensuring initialization of the bucket, the implementation of `find` is nearly identical to that of the fixed table. As shown in Figure 3.15, we first distribute the key and then find the corresponding partition for the key by computing `key mod capacity`. Then, starting at the entry within that node's table, we search through them to find the requested key. In the event that the element is found, its entry is updated with the new current time.

3.6.4 Insertion and replacement policy

Insertion also uses a similar strategy as in the fixed-sized implementation. The code shown in Figure 3.16 performs nearly identical operations, with the small addition of tracking whether or not we needed to perform an overwrite in order to log that on a per-NUMA node basis for use by the table growth mechanism. We first create the entry and then loop through all elements of the bucket that corresponds to the key until we find either an empty spot or reach the end and have to replace the oldest element of the array to point at the new entry.

```
fun initBucket (segmentIndex, subIndex, hash, index,
                segmentCount, allSegments) = let
  val startIndex = subIndex * buckets
  val SOME(segment) = Array.sub (allSegments, segmentIndex)
in
  case Array.sub (segment, startIndex)
   of UNINIT => (
     if segmentCount = 1
     then (Array.update (segment, startIndex, INIT))
     else (let
       val segmentCount' = segmentCount-1
       val index' = hash mod (capacity segmentCount')
       val (segmentIndex', subIndex') = findSegment index'
       val _ = initBucket (segmentIndex', subIndex',
                           hash, index', segmentCount', allSegments)
       val startIndex' = subIndex' * buckets
       val M' = capacity segmentCount'
       fun maybeMoveItems (i, next) = (
         if (i = buckets)
         then (Array.update (segment, startIndex, INIT))
         else (let
             val SOME(segment') = Array.sub (allSegments,
                                             segmentIndex')
             val e = Array.sub (segment', startIndex' + i)
           in
             case e
              of INIT => Array.update (segment, startIndex+next,
                                       INIT)
               | ENTRY(t, key', value) => (
                   if (key' mod M' = index)
                   then (Array.update (segment, startIndex+next,
                                       ENTRY(t, key', value));
                         maybeMoveItems (i+1, next+1))
                   else (maybeMoveItems (i+1, next)))
               | UNINIT => Array.update (segment,
                                         startIndex+next,
                                         INIT)
           end))
       in
         maybeMoveItems (0, 0)
       end))
    | _ => ()
end
```

Figure 3.14: Initialization of a bucket in a dynamically sized memoization table implementation.

```
fun find ((segments, itemCount, allSegments), key) = let
  val key' = Int.toLong key
  val hash = (c * key') mod M
  val hash = Long.toInt hash
  val segmentCount = Array.sub (segments, 0)
  val index = hash mod (capacity segmentCount)
  val (segmentIndex, subIndex) = findSegment index
  val _ = initBucket (segmentIndex, subIndex, hash, index,
                      segmentCount, allSegments)

  val SOME(segment) = Array.sub (allSegments, segmentIndex)
  val startIndex = subIndex * buckets
  fun findEntry (i) = (
    if (i = buckets)
    then NONE
    else (
      let
        val e = Array.sub (segment, startIndex + i)
      in
        case e
         of INIT => NONE
          | ENTRY(_, key', value) =>
            if key' = key
            then (Array.update (segment, startIndex+i,
                                ENTRY(Time.now(), key', value));
                  SOME value)
            else findEntry (i+1)
          | UNINIT => (NONE)
      end))
in
  findEntry 0
end
```

Figure 3.15: Lookup in a dynamically sized memoization table implementation.

```
fun insert ((segments, itemCount, allSegments), key, item) = let
  val age = Time.now()
  val new = ENTRY (age, key, item)
  val key' = Int.toLong key
  val hash = (c * key') mod M
  val hash = Long.toInt hash
  val segmentCount = Array.sub (segments, 0)
  val index = hash mod (capacity segmentCount)
  val (segmentIndex, subIndex) = findSegment index
  val _ = initBucket (segmentIndex, subIndex, hash, index,
                      segmentCount, allSegments)
  val SOME(segment) = Array.sub (allSegments, segmentIndex)
  val startIndex = subIndex * buckets
  val _ = growIfNeeded (segments, itemCount, allSegments)
  fun insertEntry (i, oldestTime, oldestOffset, overwrite) = (
    if i = buckets
    then (if overwrite
          then (let
                  val node = VProcUtils.node() * padding
                in
                  Array.update (itemCount, node,
                                Array.sub (itemCount, node) + 1)
                end)
          else ();
          Array.update (segment, startIndex + oldestOffset, new))
    else (case Array.sub (segment, startIndex + i)
           of INIT => (Array.update (segment, startIndex + i, new))
            | ENTRY (t, _, _) =>
              if t < oldestTime
              then insertEntry (i+1, t, i, overwrite)
              else insertEntry (i+1, oldestTime, oldestOffset,
                                true)
            | UNINIT => (Array.update (segment, startIndex + i,
                                       new))))
in
  insertEntry (0, Int.toLong (Option.valOf Int.maxInt), 0, false)
end
```

Figure 3.16: Insertion into a dynamically sized memoization table implementation.

3.7 Evaluation

In this section, we compare the implementation strategies described earlier for the memoization table. These strategies are explored through three benchmarks.

3.7.1 Experimental setup

Our benchmark machine is a Dell PowerEdge R815 server, outfitted with 48 cores and 128 GB physical memory. This machine runs x86_64 Ubuntu Linux 10.04.2 LTS, kernel version 2.6.32-42. The 48 cores are provided by four 12 core AMD Opteron 6172 "Magny Cours" processors. Each of the four processors is connected directly to a 32 GB bank of physical memory. Each core operates at 2.1 GHz and has 64 KB each of instruction and data L1 cache and 512 KB of L2 cache. There are two 6 MB L3 caches per processor, each of which is shared by six cores, for a total of 48 MB of L3 cache.

We ran each experiment configuration 30 times, and we report the average performance results in our graphs and tables. Times are reported in seconds.

3.7.2 Parallel Fibonacci

Using the parallel tuple syntax of PML, we compute each of the subcomputations in parallel and then add their result. Without any sharing, this strategy requires exponential time in the number of values. The PML code is shown in Figure 3.17.

```
fun add (m, n) = m + n

fun pfib i =
  case i
   of 0 => 0
    | 1 => 1
    | n => add (| pfib(i-1), pfib(i-2) |)
```

Figure 3.17: Parallel Fibonacci.

This micro-benchmark is ideal for testing the overhead associated with each of the proposed approaches, as extending the calls to `pfib` with memoization support performs minimal work other than exercising the memoization infrastructure. Extended to use memoization, the code now first checks the implementation of memoization for a value before recomputing it. Each of the three approaches described below use identical code, substituting only the implementation of the memoization infrastructure, as shown in Figure 3.18.

```
fun add (m, n) = m + n;

fun pfib (memo, i) = (
  case i
   of 0 => (0)
    | 1 => (1)
    | n =>
      (case MemoTable.find (memo, i)
        of NONE => (
            let
                val res = add (| pfib(memo, i-1), pfib(memo, i-2) |)
            in
                MemoTable.insert (memo, i, res);
                res
            end)
         | SOME res => res));
```

Figure 3.18: Parallel Fibonacci with memoization.

Performance data is shown graphically in Figure 3.19 and in numeric form in Table 3.1 when computing the Fibonacci numbers 10,000 and 60,000.[1] Data is provided only for the four memoization-based strategies, as the sharing-free parallel implementation strategy requires more than 11 seconds for a problem size of only 47. The time required for the Fibonacci of 10,000 does not differ significantly from 60,000 except at large numbers of processors on the dynamic implementation strategy. As expected, at low numbers of processors, the simple hash table is the fastest. That approach uses only a single pointer indirection, rather than the two required by both the distributed and

1. Owing to an internal design limitation of the Manticore runtime's scheduler, processes that generate more than 64k of simultaneous work items are not supported, limiting the total problem size we can handle here.

	Number of Processors							
Strategy	1	4	8	16	24	32	40	48
Simple	0.0110	0.0132	0.0144	0.0190	0.0255	0.0355	0.0497	0.0635
Distributed	0.0130	0.0146	0.0163	0.0209	0.0279	0.0405	0.0499	0.0696
Partitioned	0.0140	0.0156	0.0167	0.0210	0.0279	0.0360	0.0489	0.0709
Fixed	0.0219	0.0250	0.0267	0.0315	0.0380	0.0468	0.0552	0.0774
Dynamic	0.0451	0.0463	0.0488	0.0610	0.0837	0.0984	0.122	0.148

(a) Fibonacci number 10,000

	Number of Processors							
Strategy	1	4	8	16	24	32	40	48
Simple	0.125	0.130	0.133	0.147	0.164	0.185	0.219	0.316
Distributed	0.121	0.126	0.131	0.163	0.207	0.235	0.278	0.440
Partitioned	0.129	0.134	0.138	0.196	0.224	0.246	0.277	0.365
Fixed	0.256	0.291	0.292	0.314	0.331	0.361	0.449	0.569
Dynamic	0.406	0.408	0.412	0.450	0.675	0.779	0.942	1.21

(b) Fibonacci number 60,000

Table 3.1: Comparison of three memoization implementation strategies on parallel Fibonacci on 10,000 and 60,000. Execution times in seconds.

partitioned strategies.

Once 40 or more processors are in use, however, the partitioned strategy becomes competitive with the simple hash table strategy and is better than the distributed strategy. The distributed strategy performs additional, duplicated, work between the different processors, resulting in slightly more load, particularly on the Fibonacci of 60,000. In the partitioned strategy, the memory traffic is balanced among the various packages. The dynamic strategy performs significantly more per-element work (in particular, tracking collisions) in order to provide resizing capabilities, resulting in a factor of 2–3 slower performance on this artificial benchmark.

That said, parallel Fibonacci with caching support is not a showcase of performance scalability for Manticore — the 1 processor version provides the best performance in all cases. Scheduler overheads are very large in this benchmark relative to the amount of computation, which is why performance steadily degrades with increased numbers of processors.

3.7.3 0-1 knapsack

The 0-1 knapsack problem is to determine the maximum-value set of items that satisfy a cost constraint. By analogy to packing a knapsack, the question is: given a maximum weight budget and a set of items that each have a weight and value, which items should be selected subject to that weight budget to maximize the value?

Without remembering previously calculated values, this problem requires exponential time to compute. It is a standard example of the type of problem for which *dynamic programming* is appropriate. In dynamic programming, a table stores previously computed values by the algorithm. In this case, we would use that table to store information about a smaller problem size — with either fewer items or a lower weight budget. A straightforward example implementation is shown in Figure 3.20. While this implementation is clean and correct, its exponential runtime makes it inefficient for even moderate problem sizes.

This implementation can be made quadratic in the input parameters by adding the use of the memo table to store previously-seen values.

With a small problem size whose memoization table fits entirely in memory, this benchmark provides a test of the simple strategy against the fixed-size memoization table. In order to evaluate the cost of the fixed-size implementation's additional overhead, we first ran the knapsack problem with a smaller problem size. In this case, we use 155 distinct elements and a weight budget of 3100. At that size, no more than 490,000 elements need to be remembered, which easily fits into memory.

The results are shown graphically in Figure 3.22 and numerically in Table 3.2. The simple memoization table implementation — just an array — outperforms both the standard fixed implementation and the fixed implementation augmented with a hash distribution function. By comparison, a sequential memoizing Python implementation of this problem size requires 0.480 seconds to complete — worse than all parallel benchmarks and even the fixed hashing scheme on one processor. The difference between implementation strategies in our system largely comes down to the

	Number of Processors							
Strategy	1	4	8	16	24	32	40	48
Simple	0.130	0.129	0.131	0.0638	0.0598	0.0620	0.0745	0.0846
Fixed	0.460	0.271	0.201	0.157	0.142	0.138	0.141	0.155
Hashed	0.531	0.302	0.228	0.161	0.146	0.137	0.138	0.155
Dynamic	0.810	0.810	0.810	0.814	0.274	0.275	0.273	0.291

Table 3.2: Comparison of memoization implementation strategies on parallel 0-1 knapsack with a small number of items. Execution times in seconds.

cost of the additional pointer indirections due to an additional layer of pointers in all but the simple strategy. These additional pointer traversals cost nearly a factor of 4 at low processor counts, but less than a factor of 2 at large numbers of processors.

Additionally, we can see from these numbers that the additional work done in the hash-distributed implementation does not cost very much, except at low numbers of processors. Given that the goal of hash distribution is to gain performance by avoiding collisions but that we have no collisions in this problem and table size, this evaluation provides a good measure of the overhead associated in the calculation of the distribution function.

For the next experiment, we run at a larger benchmark size and evaluate the tradeoff between the number of elements in the fixed array and the number of collision buckets available, assuming the same overall memory budget. Additionally, we compare the direct-hash scheme with the hash-distributed implementation. The problem size used below is 200 distinct items and a weight budget of 4000. At that size, only roughly 800,000 elements need to be remembered, which can still fit into memory, but we will assume in this benchmark a budget of only 600,000 entries. A larger problem size cannot be used because this benchmark requires quadratic time, and at any size large enough to tax the memory system, the simple execution of the PML code (and the Python baseline) no longer completes in seconds, particularly at lower processor counts. The 600,000 entries were chosen because that enables us to test up to buckets of size 4. With any fewer entries (e.g., 400,000), using buckets of size 4 results in a program that does not terminate within tens of minutes on this problem size.

Table 3.3 shows performance results using a fixed-size memoization table and straight addressing of the hash values. In contrast, Table 3.4 provides results using the hashing function described in Section 3.5.5. There are several conclusions to take away from this data:

- Application of a hashing function both uniformly improves performance and narrows the range of observed values, particularly when there are a smaller number of elements and larger buckets.

- At large numbers of processors, performance is generally better the closer the size of the table is to the underlying data sets (or, the smaller the number of buckets).

- Best performance is not only a function of the problem size, but also the number of processors. At each table size, in Table 3.4(a), the best performance is at 24 processors whereas on (b) and (c) the best results are at 40 processors, with (d) rounding out the group with best performance at 32 processors.

Although they perform well, fixed-sized tables require too much manual tuning for use in an automatic feature, such as memoization. Therefore, we have taken the good ideas from the fixed-sized tables (generally small buckets and the application of a hashing function to better distribute the keys) and applied them to a dynamically resizing implementation, as described in Section 3.6.

Results of comparing the dynamic resizing strategy on both the 155 item and 200 item problem sizes are shown graphically in Figure 3.23 and numerically in Table 3.5. The six strategies all use relatively small buckets (two elements per bucket) and the same hash functions described previously. They differ in how they choose to count when an expansion of the underlying table is required, as described in Section 3.6.1. In both the small and large problem sizes, the best solution is to track conflict counts individually on NUMA nodes and to make sure that the storage assigned to each conflict count is padded out to a full cache block size. This implementation does not provide better performance than the best manually-tuned parameters to the fixed-sized table. Its performance is within a factor of 2 on this benchmark, which performs very little work apart

Number of Processors

Statistic	1	4	8	16	24	32	40	48
Mean	1.45	2.59	4.78	10.2	18.4	77.9	33.8	42.2
Max	1.46	4.88	11.8	92.8	263	1430	458	386

(a) 150,000 elements × 4 buckets per element

Number of Processors

Statistic	1	4	8	16	24	32	40	48
Mean	1.94	1.068	0.925	0.625	0.520	0.620	0.747	1.07
Max	1.96	1.31	1.98	1.04	1.14	1.01	1.94	3.00

(b) 200,000 elements × 3 buckets per element

Number of Processors

Statistic	1	4	8	16	24	32	40	48
Mean	3.92	1.06	0.735	0.485	0.399	0.356	0.340	0.372
Max	3.96	1.26	1.09	0.672	0.573	0.508	0.463	0.514

(c) 300,000 elements × 2 buckets per element

Number of Processors

Statistic	1	4	8	16	24	32	40	48
Mean	20.7	5.83	3.90	3.31	2.98	2.57	2.65	2.82
Max	20.8	7.16	5.12	4.83	3.97	3.58	4.13	4.26

(d) 600,000 elements × 1 buckets per element

Table 3.3: Comparison of the effects of bucket size on on parallel 0-1 knapsack with a fixed-sized memo table. Execution times in seconds.

from lookups in the memoization table.

Python implementation

One baseline program that we used in the above evaluation of the 0-1 knapsack problem is against a sequential Python implementation with an explicitly memoizing implementation of the algorithm. As shown in Table 3.6, this program has reasonable performance for a single-processor implementation. At a small problem size, the Python code is four times slower than the baseline fixed memoization table strategy and roughly the same speed as the fixed-sized table on a single processor. It outperforms Manticore at the large problem size with 1 and 2 processors, but once 4 processors are in use Manticore is faster. This baseline shows that our implementation has both

	Number of Processors							
Statistic	1	4	8	16	24	32	40	48
Mean	1.33	1.78	1.44	1.25	1.19	1.37	1.63	1.56
Max	1.34	2.36	2.63	2.37	2.11	2.43	4.55	2.28

(a) 150,000 elements × 4 buckets per element

	Number of Processors							
Statistic	1	4	8	16	24	32	40	48
Mean	1.14	0.687	0.448	0.335	0.273	0.245	0.241	0.249
Max	1.15	0.761	0.663	0.412	0.356	0.303	0.342	0.325

(b) 200,000 elements × 3 buckets per element

	Number of Processors							
Statistic	1	4	8	16	24	32	40	48
Mean	1.01	0.592	0.414	0.302	0.253	0.231	0.219	0.238
Max	1.02	0.677	0.590	0.396	0.337	0.303	0.287	0.371

(c) 300,000 elements × 2 buckets per element

	Number of Processors							
Statistic	1	4	8	16	24	32	40	48
Mean	0.891	0.509	0.367	0.268	0.235	0.196	0.207	0.214
Max	0.897	0.612	0.536	0.345	0.329	0.239	0.276	0.269

(d) 600,000 elements × 1 buckets per element

Table 3.4: Comparison of memoization implementation strategies on parallel 0-1 knapsack with a larger number of items and weight budget. The memoization table uses improved hashing. Execution times in seconds.

good baseline sequential performance and that it scales well.

Haskell implementation

While, in general, dynamic programming problems are easy to solve with an array in a sequential Haskell program, writing a deterministic *parallel* Haskell program that shares data across threads is more challenging. The best recent work on writing parallel Haskell programs that enable dynamic programming problems with good asymptotic behavior uses extensive program analysis and transformation to mimic the caching behavior of memoization approaches with a producer/consumer strategy [22]. This work provided a tuned parallel version of 0-1 knapsack, whose performance we

Strategy	Number of Processors 1	4	8	16	24	32	40	48
Elements	0.821	0.813	0.819	0.775	0.572	0.625	0.699	0.786
Per-Node Elements	1.20	1.08	1.07	1.00	0.423	0.443	0.484	0.537
Padded Per-Node Elements	0.973	0.971	0.974	0.776	0.313	0.315	0.273	0.316
Conflicts	0.805	0.797	0.801	0.722	0.518	0.492	0.565	0.677
Per-Node Conflicts	0.935	0.935	0.939	0.859	0.350	0.374	0.380	0.398
Padded Per-Node Conflicts	0.810	0.810	0.810	0.814	0.274	0.275	0.273	0.291

(1) 155 item problem size

Strategy	Number of Processors 1	4	8	16	24	32	40	48
Elements	1.85	1.37	1.35	1.40	1.16	1.17	1.40	1.51
Per-Node Elements	2.15	1.41	1.33	1.19	0.673	0.759	0.822	0.877
Padded Per-Node Elements	1.72	1.27	1.21	1.05	0.604	0.541	0.501	0.599
Conflicts	2.22	1.53	1.48	1.47	0.921	1.047	1.28	1.50
Per-Node Conflicts	2.01	1.35	1.29	1.16	0.547	0.550	0.619	0.617
Padded Per-Node Conflicts	2.24	1.43	1.39	1.28	0.522	0.481	0.519	0.574

(b) 200 item problem size

Table 3.5: Comparison of Manticore-based dynamically growing memoization implementation strategies on parallel 0-1 knapsack with two problem sizes. Times are reported in seconds and are mean execution times.

can see across a number of processors in Table 3.7. This Haskell implementation is more than two orders of magnitude slower than our memoization-based approach.

3.7.4 Minimax

Computer programs for games, such as chess, typically use α-β search with iterative deepening to determine the best move for a given board position [64]. A naïve implementation will treat the search space as a tree of possible board positions, where the edges correspond to moves. Since a given board position may be reached by different paths (*e.g.*, because of transposition of moves), many nodes in the tree will be redundant. A transposition table is a hash table that maps nodes in the search space to their evaluation [64]. Memoization is similar to that approach, as memoizing the evaluation function associated with each board position performs the same function as that

Size	Time (s)
155/3100	0.480
200/4000	0.790

Table 3.6: Timing for a sequential, explicitly memoizing Python implementation of 0-1 knapsack at small and large problem sizes.

	Number of Processors							
Size	1	4	8	16	24	32	40	48
155/3100	83.7	76.4	72.4	80.7	87.9	99.8	112	132
200/4000	195	174	175	192	215	255	287	329

Table 3.7: Haskell implementation of parallel 0-1 knapsack using the Generate-Test-Aggregate approach. Times are reported in seconds and are mean execution times.

transposition table.

The minimax algorithm is a simple version of α-β search, with the goal to find a move that maximizes the expected outcome from the game [64]. In this case, we are investigating the performance of several language implementations in searching for the best next move on a 4x4 tic-tac-toe board, limiting the depth of our search to four moves deep, since the full search space is impractical the benchmarked language implementations on modern hardware. The core inner loop is shown in Figure 3.24. At a given board state, for each possible move in parallel it investigates all of the possible selections of other positions by the other player and then reports back the best choice from those possible moves.

The performance of Manticore is shown in Table 3.8. Performance of the Haskell benchmark, obtained from the nofib parallel benchmark suite that is part of the compiler sources [28], is shown in Table 3.10. These implementations are compared graphically in Figure 3.25. As shown, though Manticore is slower at lower number of processors, once more processors are used, its implementation is faster even on the default parallel baseline. Further, memoization using the dynamic table-sizing strategy reduces the overall execution time.

In Table 3.9, we also show the performance of Manticore on the 5x5 tic-tac-toe board with a

depth of five moves. Though the parallel strategy takes nearly ten times as long at 48 cores as a depth of four moves, the memoized strategy is only twice as long, due to the sharing provided by the memoization strategy. This example shows how memoization can reduce the asymptotic complexity of an algorithm, at the expense of additional memory usage.

	Number of Processors							
Strategy	1	4	8	16	24	32	40	48
Parallel	6.36	1.59	0.811	0.445	0.344	0.315	0.307	0.317
Memoized	1.75	0.458	0.241	0.141	0.114	0.112	0.110	0.128

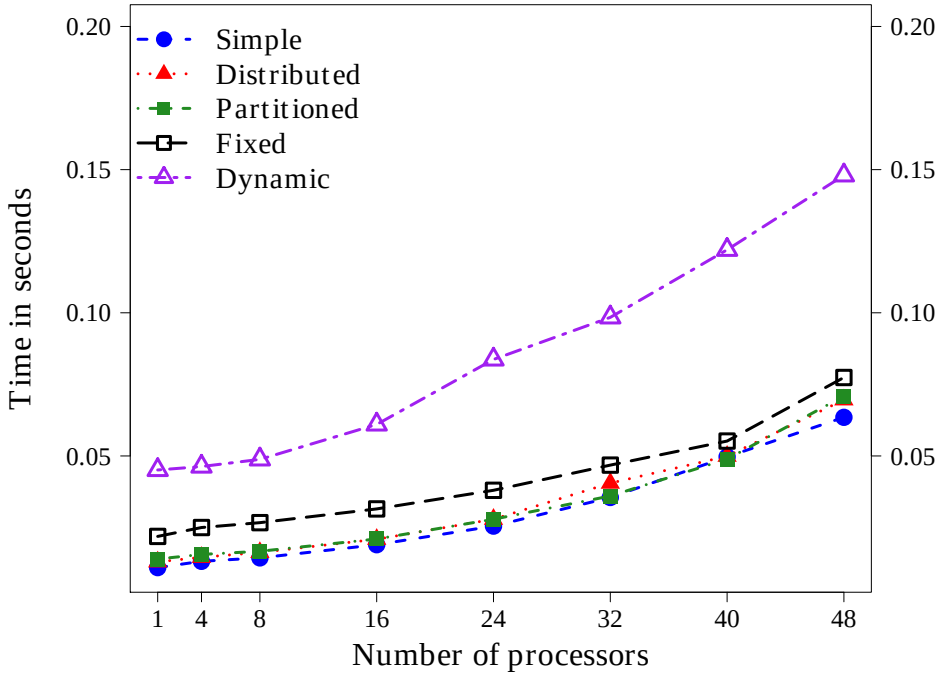
Table 3.8: Parallel implementation of minimax search on a 4x4 grid, using a cutoff depth of 4. Times are reported in seconds and are mean execution times.

	Number of Processors							
Strategy	1	4	8	16	24	32	40	48
Parallel	76.8	22.2	13.2	7.80	3.99	3.46	3.03	2.93
Memoized	7.09	1.89	0.960	0.506	0.369	0.312	0.284	0.285

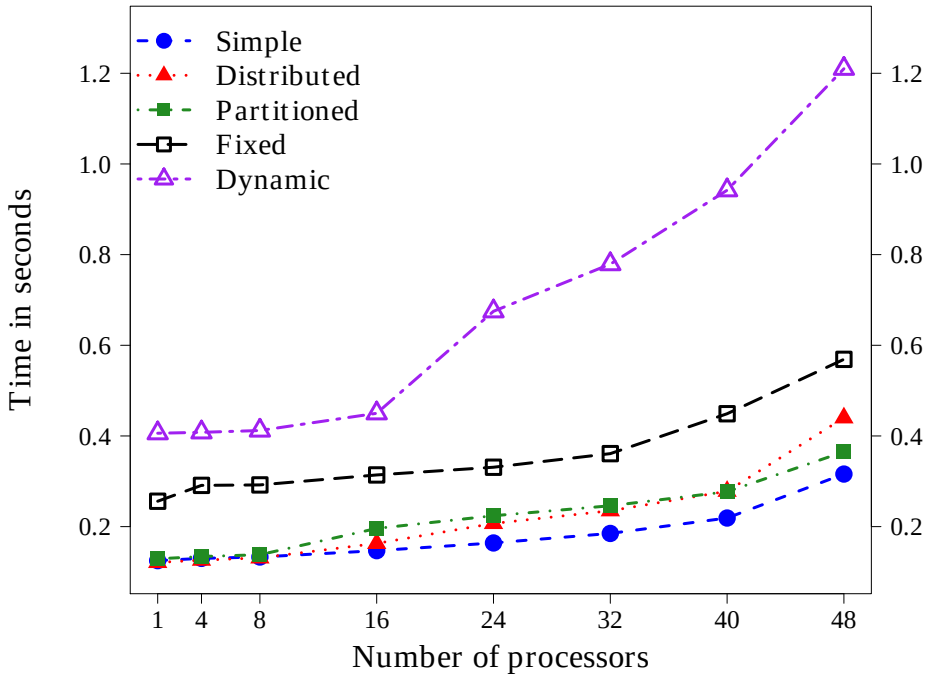
Table 3.9: Parallel implementation of minimax search on a 4x4 grid, using a cutoff depth of 5. Times are reported in seconds and are mean execution times.

	Number of Processors							
Strategy	1	4	8	16	24	32	40	48
Parallel	2.67	0.792	0.506	0.395	0.445	0.505	0.598	0.721

Table 3.10: Haskell implementation of minimax search on a 4x4 grid, using a cutoff depth of 4. Times are reported in seconds and are mean execution times.



(a) Fibonacci number 10,000



(b) Fibonacci number 60,000

Figure 3.19: Comparison of five memoization implementation strategies on parallel Fibonacci on 10,000 and 60,000. Execution times in seconds.

```
val weights = ...
val values = ...

fun knap (i, avail, weights, values) =
  if (avail = 0) orelse (i < 0)
  then 0
  else (if Vector.sub(weights, i) < avail
    then
      let
        val (a, b) = (| knap (i-1, avail, weights, values),
                        knap (i-1, avail - Vector.sub(weights, i),
                              weights, values) +
                        Vector.sub(values, i) |)
      in
        if a > b
        then a
        else b
      end
        else knap (i-1, avail, weights, values))
```

Figure 3.20: Simple parallel knapsack

```
val weights = ...
val values = ...

fun knap (i, avail, memo) =
  if (avail = 0) orelse (i < 0)
  then 0
  else (
    if Vector.sub(weights, i) < avail
    then let
      fun lookup (a, b) = let
          val index = a + Vector.length values * b
      in
          case MemoTable.find (memo, index)
           of SOME a => a
            | NONE => let
                val res = knap (a, b, memo)
            in
                MemoTable.insert (memo, index, res);
                res
            end
      end
      val (a, b) = (| lookup (i-1, avail),
                      lookup (i-1, avail - Vector.sub(weights, i))
                      + Vector.sub(values, i) |)
      in
        if a > b
        then a
        else b
      end
    else knap (i-1, avail, memo))
```

Figure 3.21: Memoized parallel knapsack

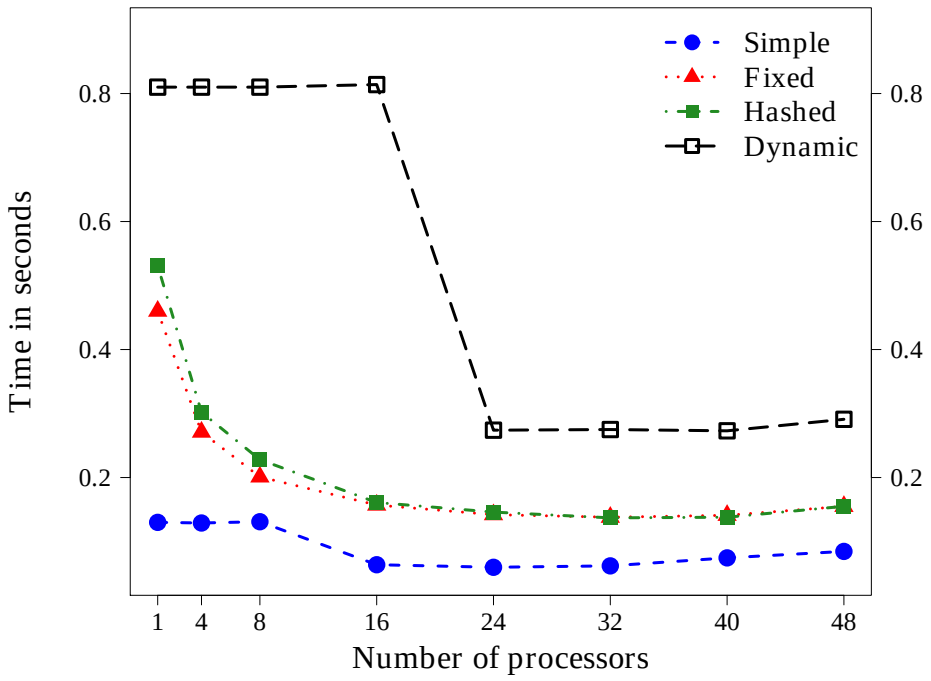
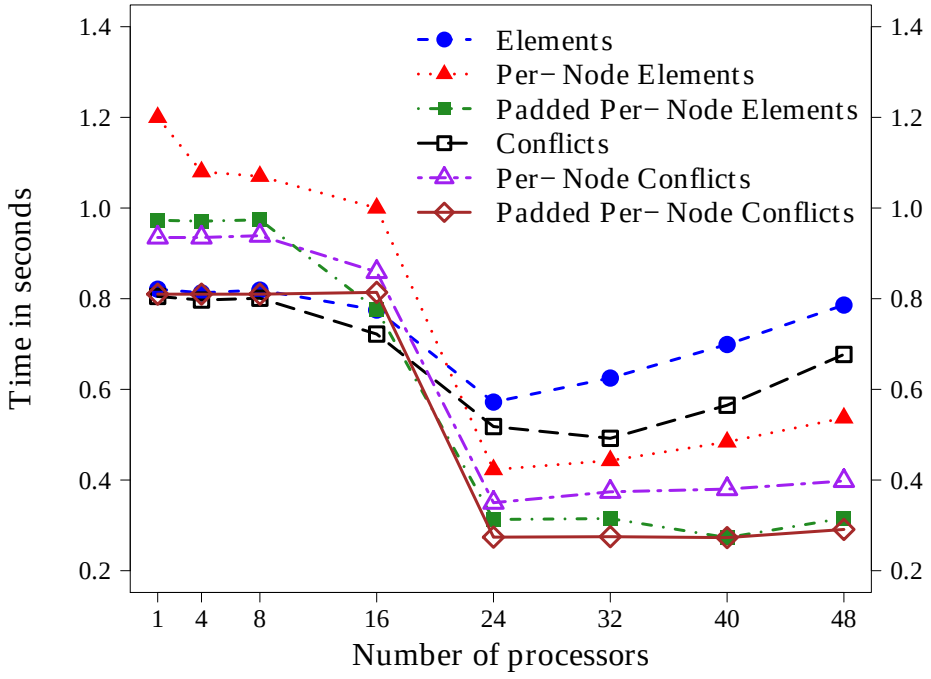
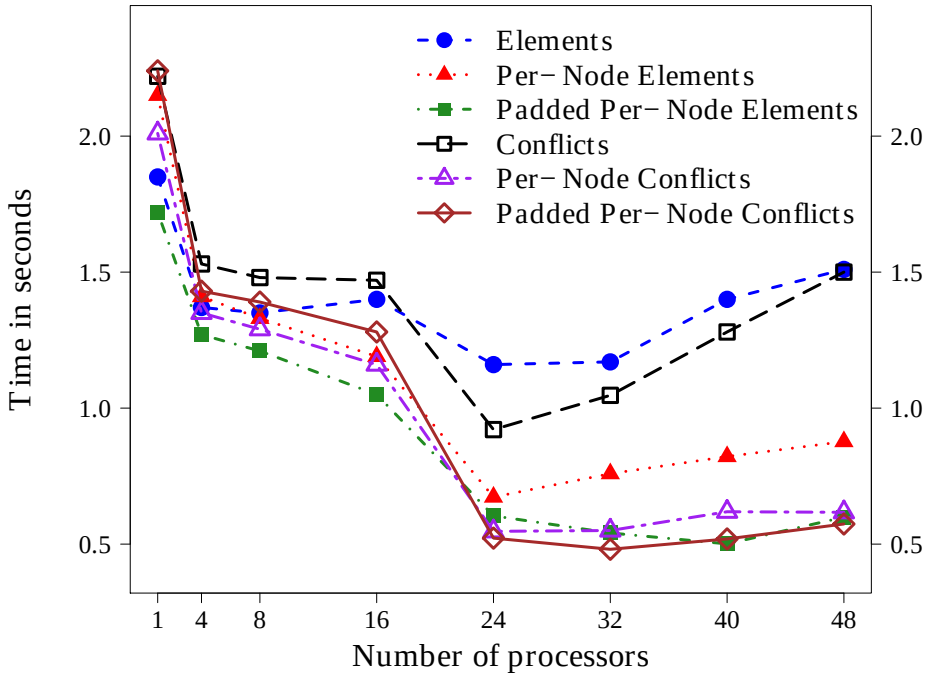


Figure 3.22: Comparison of memoization implementation strategies on parallel 0-1 knapsack with a small number of items.



(a) 155 item problem size



(b) 200 item problem size

Figure 3.23: Comparison of Manticore-based dynamically growing memoization implementation strategies on parallel 0-1 knapsack with two problem sizes.

```
datatype player = X | O

type board = player option list

datatype 'a rose_tree (* general tree *)
  = Rose of 'a * ('a rose_tree list)

(* minimax : player -> board -> (board * int) rose_tree *)
(* Build the tree of moves and score it at the same time. *)
(* p is the player to move *)
(* X is max, O is min *)
fun minimax (p : player, depth) (b : board) : game_tree =
  if (depth=0 orelse gameOver(b)) then
    mkLeaf (b, score b)
  else let
    val trees = parMap (minimax (other p, depth-1),
                        (successors (b, p)))
    val scores = map (compose (snd, top)) trees
  in
    case p
     of X => Rose ((b, listmax scores), trees)
      | Y => Rose ((b, listmin scores), trees)
  end
```

Figure 3.24: The minimax algorithm in Parallel ML.

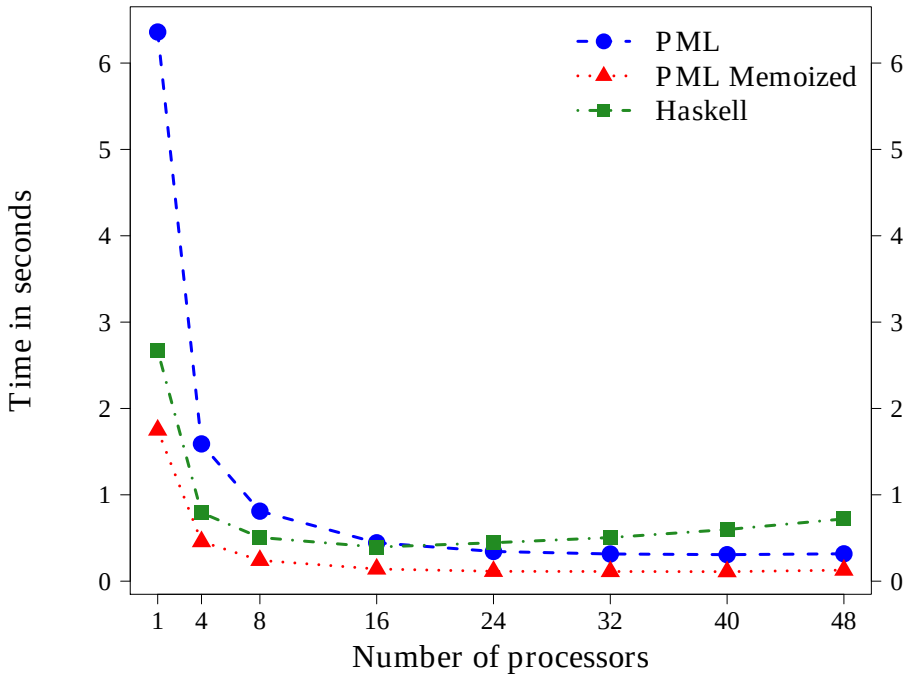


Figure 3.25: Comparison of minimax search on a 4x4 grid, using a cutoff depth of 4.

3.8 Conclusion

Memoization is a syntactically small extension to the surface language of Parallel ML (PML) that takes advantage of the purity of PML code to enable sharing across tasks without synchronization. Because it relies on purity and does not itself change the behavior of the code, the memoization feature itself is pure. While we could have built this feature on top of the implementation of mutable state, discussed in Chapter 4, we would lose the purity of the implementation, would have less sharing, and would be forced to implement synchronization, as described in that chapter.

From an implementation perspective, we have shown in this chapter that memoization is best implemented with a dynamically resized hash table. We have also shown that the use of the division method to distribute those hash values has a positive effect on performance. Finally, we have shown several example benchmarks that demonstrate that this feature both scales down to microbenchmarks with little computation and up to larger programs that perform more work.

CHAPTER 4

MUTABLE STATE

In Parallel ML (PML), it is not possible for multiple implicitly-threaded parallel computations to share results asynchronously. Memoization, as described in Chapter 3, adds the ability to share results between parallel threads for pure functions. But, that feature does not address the need to share the results of non-deterministic expressions, monotonically increasing values, or other forms of shared state where the result is not a pure function of its input. For example, consider the following code:

```
let val x = ref 1
in
  (| x := !x+1, x := !x+1 |);
  !x
end
```

There are two parallel threads that each should increment the value associated with the mutable reference cell bound to the variable `x` by 1. In this chapter, we provide an extension to PML called *mutable state* that implements two execution models, one of which is deterministic and one which is non-deterministic. In both, the value associated with `!x` at the end of this code will be 3.

Our mutable state implementation extends the current Manticore programming model with two stateful language features commonly used in Standard ML [25]. The reference cell, or `ref` cell, is a single location in memory that is mutable. Arrays are mutable vectors of data, providing a compact, integer-addressed sequence of mutable locations. These features permit *mutation* of values that can be shared between threads asynchronously. This sharing is unique to mutable state, as all other forms of inter-thread communication by Parallel ML programs require synchronization.

The key challenges with adding these two features are:

- Preserving local reasoning. Programmers should be able to understand the behavior of a given function by analyzing it in isolation from the rest of the program.

- Efficient implementation. The cost for using these features in parallel must not outweigh the alternatives of either writing the algorithm sequentially or using a less-efficient mutation-free parallel algorithm.

In this chapter, we first explain the design of our model and the translation of the high-level language with parallelism and direct access to mutable state into one with explicit parallel constructs and locks. In that model, we can then remove locks from expressions guaranteed not to access mutable state, preserving parallelism for pure computations. These locks and their optimizations are discussed in detail. We provide an evaluation of this approach on several benchmarks, which we follow with a more detailed discussion of local reasoning before we conclude.

4.1 Core-PML with memoization and mutable state

In Figure 4.1, we extend the language of Figure 3.1 with mutable state, in the form of reference cells. A reference cell is a location in memory that stores a value that is free to change over time. The **`ref`** operation creates a reference cell holding a value that is the result of evaluating an expression. The **`!`** and **`:=`** operators respectively read and update the value of the reference cell.

e	::=	x	variables
	|	c	constants
	|	**()**	unit value
	|	**SOME** e	option type with value
	|	**NONE**	option type with no value
	|	**fun** $f\ x = e_1$ **in** e_2	functions
	|	$e_1\ e_2$	application
	|	**case** e_1 **of** $p_1 \Rightarrow e_2$ **|** $p_2 \Rightarrow e_3$	case analysis
	|	**(**e_1, e_2**)**	tuples (pairs)
	|	**(|**e_1, e_2**|)**	parallel tuples
	|	**let** $p = e_1$ **in** e_2	bindings
	|	**ref** e	create mutable state
	|	**!**e	read mutable state
	|	e_1 **:=** e_2	update mutable state
p	::=	x	
	|	c	
	|	**()**	
	|	**(**p_1, p_2**)**	
	|	**SOME** p	
	|	**NONE**	

Figure 4.1: Source language with memoization and mutable state

4.2 Translation to atomic expressions

In this section, we translate a subset of the PML source language with parallel expressions and state to one that is also augmented with `atomic` expressions, which contain the extra static information required by the implementations of the two execution models. This target language is in Figure 4.2

In this translation, expressions that execute in parallel are wrapped within an `atomic` expression. This wrapper indicates that its expression should be evaluated in isolation from all other similarly wrapped expressions. In particular, this wrapper requires that the result of the evaluation should be as if no update operation performed by a parallel evaluation became visible during the evaluation of the isolated expression. The `lock` expression allocates a new lock.

The translation from the source to target language is shown in Figure 4.3. The parallel tuple construct is expanded and its subexpressions wrapped in `atomic` wrappers. These wrappers take three arguments: the first is a lock variable, the second is an integer used for sequencing that

e	::=	x	variables
	|	c	constants
	|	`()`	unit value
	|	**SOME** e	option type with value
	|	**NONE**	option type with no value
	|	**fun** $f\ x = e_1$ **in** e_2	functions
	|	$x\ e$	application
	|	**case** e_1 **of** $p_1 \Rightarrow e_2$ **|** $p_2 \Rightarrow e_3$	case analysis
	|	`(`e_1`,` e_2`)`	tuples (pairs)
	|	`(|`e_1`,` e_2`|)`	parallel tuples (pairs)
	|	**let** $p = e_1$ **in** e_2	binding
	|	**atomic** (e, c, e)	isolated execution
	|	**lock ()**	allocate a new lock
	|	**ref** e	create mutable state
	|	**!**e	read mutable state
	|	e_1 **:=** e_2	update mutable state
p	::=	x	
	|	c	
	|	`()`	
	|	`(`p_1`,` p_2`)`	
	|	**SOME** p	
	|	**NONE**	

Figure 4.2: Target language

corresponds to the either the first or second position within a parallel tuple, and the third is an arbitrary expression. In preparation for later effect analysis, the application form now requires a named variable for the target of the application instead of an expression.

Translation rules are written $\mathcal{T}'[\![e]\!]$, where e is a source language expression. In the parallel tuple rule, $\mathcal{T}'[\![$ `(|`e_1`,` e_2`|)` $]\!]$, first we allocate a new lock to share among the subexpressions. Then, we wrap the subexpressions in an `atomic` wrapper, annotating it with the static order of the expressions within the tuple. The allocated lock is only captured and accessed by these two wrapper expressions.

$$
\begin{array}{rcl}
\mathcal{T}'[\![x]\!] & = & x \\
\mathcal{T}'[\![c]\!] & = & c \\
\mathcal{T}'[\![\,\texttt{()}\,]\!] & = & \texttt{()} \\
\mathcal{T}'[\![\texttt{SOME}\ e]\!] & = & \texttt{SOME}\ \mathcal{T}'[\![e]\!] \\
\mathcal{T}'[\![\texttt{NONE}]\!] & = & \texttt{NONE} \\
\mathcal{T}'[\![\texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2]\!] & = & \texttt{fun}\ f\ x = \mathcal{T}'[\![e_1]\!]\ \texttt{in}\ \mathcal{T}'[\![e_2]\!] \\
\mathcal{T}'[\![e_1\ e_2]\!] & = & \texttt{let}\ x\ =\ \mathcal{T}'[\![e_1]\!]\ \texttt{in} \\
 & & \quad x\ (\mathcal{T}'[\![e_2]\!]) \\
 & & \quad \text{where } x \text{ is fresh} \\
\mathcal{T}'[\![\texttt{case}\ e_1\ \texttt{of}\ p_1\ \Rightarrow\ e_2\ \texttt{|}\ p_2\ \Rightarrow\ e_3]\!] & = & \texttt{case}\ \mathcal{T}'[\![e_1]\!]\ \texttt{of}\ p_1\ \Rightarrow\ \mathcal{T}'[\![e_2]\!]\ \texttt{|}\ p_2\ \Rightarrow\ \mathcal{T}'[\![e_3]\!] \\
\mathcal{T}'[\![\,\texttt{(}e_1,\ e_2\texttt{)}\,]\!] & = & \texttt{(}\mathcal{T}'[\![e_1]\!],\ \mathcal{T}'[\![e_2]\!]\texttt{)} \\
\mathcal{T}'[\![\,\texttt{(|}e_1,\ e_2\texttt{|)}\,]\!] & = & \texttt{let}\ l\ =\ \texttt{lock ()}\ \texttt{in} \\
 & & \quad \texttt{(|atomic}\ (l, 1, \mathcal{T}'[\![e_1]\!]), \texttt{atomic}\ (l, 2, \mathcal{T}'[\![e_2]\!])\texttt{|)} \\
 & & \quad \text{where } l \text{ is fresh} \\
\mathcal{T}'[\![\texttt{let}\ p\ =\ e_1\ \texttt{in}\ e_2]\!] & = & \texttt{let}\ p\ =\ \mathcal{T}'[\![e_1]\!]\ \texttt{in}\ \mathcal{T}'[\![e_2]\!] \\
\mathcal{T}'[\![\texttt{ref}\ e]\!] & = & \texttt{ref}\ (\mathcal{T}'[\![e]\!]) \\
\mathcal{T}'[\![\texttt{!}e]\!] & = & \texttt{!}(\mathcal{T}'[\![e]\!]) \\
\mathcal{T}'[\![e_1\ \texttt{:=}\ e_2]\!] & = & (\mathcal{T}'[\![e_1]\!])\ \texttt{:=}\ (\mathcal{T}'[\![e_2]\!])
\end{array}
$$

Figure 4.3: Translation into target language

4.3 Lock-based implementation

In this work, we have used a lock-based implementation. This implementation strategy allows us to preserve the semantics required by the two execution models (serial and transactional) without implementing any form of logging related to the reads and writes performed by a task. Unfortunately, this strategy reduces the amount of parallelism available in the program, as it will dynamically prohibit expressions that may perform conflicting mutations of state from doing so, rather than allowing them and only penalizing performance in the case of conflicts.

4.3.1 Ticket lock

In the Manticore scheduler, we use simple spin locks because they have a single owner and rarely more than one writer [63]. This balance of threads is not present in our implementation of these parallel features, where many tasks will access the same lock. Additionally, under some of our execution models we would like to statically determine the dynamic order in which the locks should be acquired. For these reasons, we have chosen to use a *ticket lock* implementation [54].

Ticket locks consist of a pair of values. These values correspond to the number of requests for the lock and the number of times that it has been unlocked. When a thread wishes to acquire the lock, it performs an atomic operation to both increment and retrieve the number of requests for the lock. Then, that thread waits for the number of unlocks to reach the number of requests returned. After it has completed its work, the thread unlocks by incrementing the unlock count. Thus, the number returned from the atomic increment serves as a *ticket* and defines the order in which threads acquire and release the lock.

Our implementation in Manticore is written in a compiler-specific dialect of Standard ML called inline BOM [23]. This dialect directly corresponds to one of our intermediate representations and allows access to primitives not available at the Standard ML level, at the expense of lacking nearly all of the syntactic sugar of the surface language (*e.g.*, polymorphism, functors,

datatypes). The basic ticket lock implementation is shown in Figure 4.4. As described above, the type of a raw `ticket_lock` is a mutable data structure consisting of two long (64-bit) integers. The `@create` function allocates this structure and calls `promote` to ensure that it has been copied into a globally visible heap page. By default in Manticore, newly allocated values are only locally visible and access from other threads is prohibited [5].

The `@lock` function acquires a ticket using an atomic operation (`I64FetchAndAdd`) that retrieves and increments the number of requests and then spins until the number of unlocks is equal to the request number retrieved. In the case where the current ticket is not the same as the held one, the loop sleeps. As in other ticket lock implementations, we sleep for a constant multiple of the minimum possible hold time for the lock. Since this lock has ordered acquisitions, there is a risk of starvation if a thread uses another strategy, such as exponential backoff [54]. Finally, the `@unlock` method sets the number of releases to one more than the current ticket. We do not need to read the value that was previously in there, since we always know it will be the same as our ticket, avoiding one atomic operation.

Figure 4.5 contains more utility code for ticket locks that is used by this implementation of state. The function `@get_ticket` returns a ticket without spinning to acquire the lock. The `@current_ticket` function returns the number associated with the current lock holder. This function allows clients to poll for the lock at the surface language level using a custom strategy. Finally, the `@lock_with_ticket` function allows consuming code to attempt to take the lock based on some other ticket allocation strategy. As will be shown later when we discuss preserving static program order, this function allows us to statically ensure a dynamic lock acquisition order.

```
structure TicketLock = struct
  _primcode (

  typedef ticket_lock =
                ![
                   long,       (* num unlocks *)
                   long        (* num requests *)
                 ];

  define @create (_ : unit / exh : exh) : ticket_lock =
      let l : ticket_lock = alloc (0:long, 0:long)
      let l : ticket_lock = promote(l)
      return(l)
    ;
  define @lock (l : ticket_lock / exh : exh) : ml_long =
      let ticket : long = I64FetchAndAdd (&1(l), 1:long)
      let p : ml_long = alloc(ticket)
      fun spinLp () : ml_long =
          let difference : long = I64Sub(ticket,#0(l))
          if I64Eq(difference, 0:long)
          then
              return (p)
          else
              let t : long = I64Mul(difference, 100000:long)
              do SchedulerAction.@sleep (t)
              apply spinLp ()
      apply spinLp()
    ;
  define @unlock (l : ticket_lock, t : ml_long / exh : exh) : unit =
       let t : long = #0(t)
       let t : long = I64Add(t,1:long)
       do #0(l) := t
       return (enum(0):PrimTypes.unit)
    ;
  )
  type ticket_lock = _prim(ticket_lock)
  val create : unit -> ticket_lock = _prim(@create)
  val lock : ticket_lock -> long = _prim(@lock)
  val unlock : ticket_lock * long -> unit = _prim(@unlock-w)
end
```

Figure 4.4: Basic ticket lock implementation.

```
define @get_ticket (l : ticket_lock / exh : exh) : ml_long =
    let ticket : long = I64FetchAndAdd (&1(l), 1:long)
    let p : ml_long = alloc(ticket)
    return (p)
  ;

define @current_ticket (l : ticket_lock / exh : exh) : ml_long =
    let ticket : long = #0(l)
    let p : ml_long = alloc(ticket)
    return (p)
  ;

define @lock_with_ticket (l : ticket_lock, ticket : ml_long
/ exh : exh) : unit =
    let ticket : long = #0(ticket)
    fun spinLp () : unit =
        let difference : long = I64Sub(ticket,#0(l))
        if I64Eq(difference, 0:long)
        then
            return (enum(0):PrimTypes.unit)
        else
            let t : long = I64Mul(difference, 100000:long)
            do SchedulerAction.@sleep (t)
            apply spinLp ()
    apply spinLp()
  ;
```

Figure 4.5: Utility code for more advanced ticket lock usage.

4.4 Serial execution

In the serial execution model, the dynamic execution of the program must respect the static program order of the expressions. We use the utility code shown in Figure 4.5 to ensure a serial execution while still permitting parallelism elsewhere in the program. In the translation shown in Figure 4.3, expressions executed in parallel are augmented with their static program order, generating expressions of the form:

$$\textbf{atomic}\,(lock, n, e)$$

Where $lock$ is the dynamically-created lock, n is the static program order of the `atomic` within the parallel tuple, and e is the expression. Our implementation generates locking code directly from this information. For the above code segment, we will generate the following code:

```
let val _ = TicketLock.lockWithTicket (lock, n)
    val result = e
    val _ = TicketLock.unlock (lock, n)
in
    result
end
```

4.4.1 Correctness

In this implementation, we are using a ticket lock with a static acquisition order provided by the original source translation, which annotated the `atomic` expressions with their sequential order within the parallel constructs. To show that this translation is correct, we need to prove that for any two expressions e_1 and e_2 that both mutate state where e_1 is evaluated before e_2 in the serial program order, under any parallel execution with this serial implementation, e_1 will acquire and release a lock before e_2. Consider the history corresponding to the serial execution and any history corresponding to a parallel execution under this model. We need to show that for all parallel

executions, e_1 will still precede e_2. In this naïve translation, that property is straightforward, as this locking strategy, as described, removes all parallelism and forces serial and in-order execution of all expressions in a parallel tuple.

Consider an extension of this translation where expressions that do *not* mutate state are allowed to proceed in parallel with any other expression. We now have a set of potential parallel executions and many potential histories for a given program execution. But, in all of these executions, the only operations that may be reordered within the history are those that do not mutate state. Therefore, we have preserved the serial execution order on expressions that mutate state.

4.4.2 Deadlock freedom

A deadlock occurs when two parallel threads or tasks have cyclic dependencies. In this system, such a dependency could be created if they were each waiting on locks that the other holds. One way to avoid deadlock is to *order* all locks so that if any subset of them are acquired, they must be acquired in the same order [41]. In the serial execution model, we ensure deadlock freedom by limiting the static scope of our locks. Any given lock will be acquired only by the expressions within the original parallel tuple. Upon entry to the lock's allocation, the individual task will own a (possibly empty) list of locks. There is no operation performed between the creation of the lock and its acquisition, eliminating the possibility of acquiring another lock. Further, no other task may acquire that newly created lock, as its scope is restricted to the single parallel execution of tasks. Therefore, each new lock will always be acquired by tasks with an identical set of preceding locks, guaranteeing that any lock was acquired using a fixed ordering of preceding locks.

4.5 Transaction-based implementation

While our transactional implementation relies on the ticket locks described in Section 4.3.1 above, we use a straightforward implementation of transactional memory that is further simplified by the

Manticore execution model and constraints of the PML language. Our software-based implementation of transactional memory is similar to the approach used in Haskell [37]. Upon entry to a `atomic` block, a transaction record is created and associated with the task (nesting, if needed). All reads and writes to atomic objects are stored in that transaction record. At the end of the `atomic` block, the code will attempt to *commit* those changes. A commit is designed with two forms in our system:

- If the transaction occurred at the toplevel, then locks are taken on the atomic objects. If those atomic objects have the same original values as read by the transaction, then the logged changes are written and transaction is done. If not, then the transaction is *aborted* and we retry starting at the beginning of the `atomic` block.

- If the transaction is *nested*, then its parent transaction is similarly locked and checked for local changes to the read or written objects. If there are none, then the nested changes are logged to the parent and the nested transaction is done. If there were conflicting changes, then the transaction is *aborted* and we retry starting at the beginning of the `atomic` block.

Nested transactions are not currently implemented, though we describe their integration with the rest of the system here and their semantics in Chapter 5.

There are two types of transactional objects tracked in our system. They are the analogs of the `ref` cell and `array` datatype. The *transactional reference cell*, or `tref`, is the underlying implementation of a reference cell and contains a lock and a global ordering number. The *transactional array* is a standard `array`, except that it has both a global ordering number and a set of locks, defaulting to one lock per element. While locking each entry requires additional memory for those locks, as we show in the account transfer benchmark results in Section 4.8.1, if the program has large numbers of interfering concurrent operations, reducing the number of locks reduces performance. The global ordering numbers are shared between the two types of transactional objects and are used to establish a lock order that prevents deadlock between committing transactions.

4.6 Transactional execution

In the transactional execution model, the only semantic constraint that we have is that two expressions that modify the same piece of mutable state must see either all or none of the changes made by one another. So, in this case, we again use the ticket lock interface shown in Figure 4.4, but require significant additional code for the implementation of read/write logging, starting transactions, and committing them. Again, as shown in the translation shown in Figure 4.3, expressions executed in parallel are augmented with their static program order, generating expressions of the form:

$$\texttt{atomic}\,(lock, n, e)$$

Where $lock$ is the dynamically-created lock, n is the static program order of the `atomic` within the parallel tuple, and e is the expression.

4.6.1 Starting a transaction

A transaction record contains both the read and write logs. These logs record all of the values read or written during the execution of the transaction. In the case of a nested transaction system, the record also maintains a pointer to the parent transaction, which is empty in the case of a top-level transaction.

4.6.2 Reading and writing values

When reading a value from a piece of transactional memory, the transaction cannot just query the object directly. First, it must check its transactional structure to determine if a new value has been written to that object. Then, it must follow the parent chain of transactional structures to see if any of them have written a new value. Finally, if none of them have, it may read the value from the object in memory. This value is stored into the current transactional structure's read log and then returned to the executing program. Writing is simpler; the value is written directly into the current

transactional structure's write log.

4.6.3 Commit

As mentioned earlier in this section, the procedure used by commit is different depending upon whether the current transaction is the top-level transaction or a nested transaction. In a top-level transaction, first we take locks on all read and written objects, using their global numbering as the order for lock acquisition. Then, we check for any changes that occurred in global memory to values read or written by the transaction. If there were none, then we write our changes and the transaction is done. If there were conflicts, then we remove our transactional record and retry the operation from the beginning of the `atomic` block. In either case, all locks are released.

In a nested transaction, we only need to check our parent transaction for read or write conflicts. If there are none, we simply concatenate our read and write log entries to those of the parent transaction. If there were conflicts, then we remove our transactional record and retry the nested operation from the beginning of the `atomic` block. Note that during this operation, we needed to lock our parent transaction, as otherwise we could conflict with a sibling transaction when reading or editing the logs. Other than these log edits, the parent transaction is unaffected.

4.7 Removing unnecessary `atomic` wrappers

Each `atomic` wrapper corresponds to an additional transaction record and set of dynamic locking operations. When semantically valid, removing these wrappers will reduce the transactional overhead. There are two cases where `atomic` wrappers may be removed without altering the semantics of the program.

- If the wrapped expression is pure; that is, it does not modify mutable state.
- If the wrapped expression modifies mutable state, but there are no concurrent expressions that modify the same regions of memory.

In this section, we address only the first case. We note in the analysis where this approach may be extended to the second case, but the first suffices for the primary design goal — we do not want to slow down mutation-free code that previously was executed in parallel and might have performance penalties under the transformation presented in Figure 4.3.

4.7.1 Effects

Effects are the computational actions performed during the evaluation of code, beyond simply the type of the value resulting from the evaluation [31]. Languages that treat effects explicitly commonly make one or more of mutable state, I/O, and concurrency explicitly tracked and verified by the implementation [30]. In this work, we track only mutable state.

There are two broad categories of effect systems. The first — and most popular — is *annotation*-based systems. In these languages, the programmer is required to annotate their programs with additional information about the effects performed by expressions as well as possibly how the runtime should handle these effects. Typically, the compiler then implements a validator that ensures the effect annotations are provably correct.

The other major approach is *effect inference*. Either as an extension of type inference or as a separate analysis pass of the compiler, the system itself will determine the effects associated with each expression. Our system uses this approach, as it fits more cleanly with the Manticore project design goal of minimizing programmer burden.

4.7.2 Effect language

Our analysis computes effects after type inference has been performed, against the BOM direct-style intermediate representation in Manticore, as described in Section 2.4. The effect analysis computes the effects related to mutable state usage for each expression and each function definition in the entire program (including all libraries). These effects are represented by values from the grammar shown in Figure 4.6.

$$
\begin{array}{rcll}
\varepsilon & \in & \mathit{effect} & ::= \quad PURE \\
& & & \mid \quad MUTATE \\
& & & \mid \quad ATOMIC \\
& & & \mid \quad \top
\end{array}
$$

Figure 4.6: Effects analyzed for removal of redundant atomics.

These effects are a conservative characterization of the possible effects of each expression. The $PURE$ effect corresponds to an expression guaranteed not to manipulate mutable state. We also mark expressions in unreachable code that has not yet been eliminated by the dead code elimination optimization with this effect. The effect $MUTATE$ describes allocation, reading, or writing mutable state. $ATOMIC$ indicates an expression that performs a synchronization operation (i.e., the `atomic` wrapper described earlier in the transformation). Finally, $\top$ is the catch-all effect that indicates anything could happen. This value is introduced when, owing to the imprecision of our analysis, we are unable to reason about a given expression. We use this value instead of simply promoting to $MUTATE$ in order to be able to track the analysis imprecision directly, though the two distinct effect values are treated identically by the optimization phase. Since the goal of this analysis is to provide sufficient information to remove `atomic` wrappers when they appear around an expression that is guaranteed to be mutation-free, this coarsening suffices. If we wish to also allow non-overlapping uses of mutable state, then we need to additionally condition the $MUTATE$ operation with either a static or dynamic value associated with each piece of mutable state. We do not perform this more detailed analysis and optimization in this work for two reasons. First, the analysis is significantly more complicated, as useful optimizations — such as allowing parallel recursive decomposition of mutable structures — require also reasoning about subregions of arrays, multiple dynamic allocation paths, and understanding partitioning of data in the recursive calls of divide-and-conquer algorithms. Recursive calls and support for higher-order functions are both common usage patterns in functional languages and more complicated than **`for`** loops to analyze. Second, code that is sufficiently simple in its partitioning for a practical static analysis to infer non-overlapping regions has proven to perform well and scale through translation to pure

functional code in previous work on Manticore [10].

4.7.3 Effect analysis

The result of effect analysis is a finite map, $\mathcal{E}$, that maps from program locations — also known as program points — to effects. In order to build up this map, we need to augment the intermediate representation from Figure 4.2 with locations for each expression. Figure 4.7 shows the new intermediate representation, which differs from the original by changing the original expression nodes, e, into terms, t, which are now paired with a location at every occurrence. These locations are from a countable set and are unique for each expression in the program.

e	::=	(ℓ, t)	pair of a location and a term
t	::=	x	variables
	|	c	constants
	|	**()**	unit value
	|	**fun** $f\ x = e$ **in**	functions
	|	$x\ e$	application
	|	**(**e_1, e_2**)**	tuples (pairs)
	|	**(|**e_1, e_2**|)**	parallel tuples (pairs)
	|	**let** $p = e_1$ **in** e_2	binding
	|	**atomic** (e, c, e)	isolated execution
	|	**lock ()**	allocate a new lock
	|	**ref** e	create mutable state
	|	**!**e	read mutable state
	|	e_1 **:=** e_2	update mutable state
p	::=	x	
	|	c	
	|	**()**	
	|	**(**p_1, p_2**)**	

Figure 4.7: Location-augmented target language

This analysis builds up a finite map, $\mathcal{E}$, from locations to their effects, as defined in Figure 4.6 over the language in Figure 4.7. It also tracks the latent effects associated with the invocation of a function in a second finite map, $\mathcal{F}$, from variables (restricted to function names from `fun` bindings) to latent effects.

$$
\begin{array}{rcl}
\mathcal{E} & : & \ell \xrightarrow{fin} \varepsilon \\
\mathcal{F} & : & var \xrightarrow{fin} \varepsilon \\
\mathcal{C} & : & var \xrightarrow{fin} var
\end{array}
$$

Figure 4.8: Finite maps produced and used by the analysis

The analysis is defined as a dataflow analysis, augmented with control-flow information, as shown in Figure 4.10. The overall effect for each expression is defined in terms of the effects of each of its subexpressions. At the start of the analysis, all locations and variables in the maps $\mathcal{E}$ and $\mathcal{F}$ are initialized with the value $PURE$, with the exception of the library functions that perform mutation operations, which are initialized as follows:

$$
\begin{array}{rcl}
\mathcal{F}[Array.array] & = & MUTATE \\
\mathcal{F}[Array.sub] & = & MUTATE \\
\mathcal{F}[Array.update] & = & MUTATE
\end{array}
$$

Arrays are the only portion of the library that need to be initialized separately, as references cells are treated as a special language form.

$$
\begin{array}{rcl}
\mathcal{M} & : & \mathcal{E} \times \mathcal{E} \to \mathcal{E} \\
\mathcal{M}[PURE, x] & = & x \\
\mathcal{M}[x, PURE] & = & x \\
\mathcal{M}[ATOMIC, MUTATE] & = & MUTATE \\
\mathcal{M}[MUTATE, ATOMIC] & = & MUTATE \\
\mathcal{M}[\_, \top] & = & \top \\
\mathcal{M}[\top, \_] & = & \top
\end{array}
$$

Figure 4.9: Effect merge rules

Effects across multiple subexpressions are merged by the $\mathcal{M}$ function, defined in Figure 4.9. Finally, we use $\mathcal{C}$ to denote the possible function targets of a given variable, as returned by the Manticore control-flow analysis [8].

Variables and constants are pure expressions. Function definitions update both of the effect

Expression form	Update rule
(ℓ, x)	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], PURE]$
(ℓ, c)	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], PURE]$
$(\ell, \texttt{fun}\ f\ x = e_1\ \texttt{in}\ e_2)$	$\begin{cases} \mathcal{F}[f] = \mathcal{M}[\mathcal{F}[f], \mathcal{E}[e_1]] \\ \mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], \mathcal{E}[e_2]] \end{cases}$
$(\ell, x\ e)$	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], \varepsilon']$ $\quad where\ \varepsilon' = \top$ $\quad\quad if \mathcal{C}[x] = \top$ $\quad where\ \varepsilon' = \mathcal{E}[e]$ $\quad\quad \forall_{f \in \mathcal{C}[x]} (\varepsilon' = \mathcal{M}[\mathcal{F}[f], \varepsilon'])$ $\quad\quad otherwise$
$(\ell, (e_1, e_2))$	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], \varepsilon'']$ $\quad where\ \varepsilon' = \mathcal{E}[e_1]$ $\quad\quad \varepsilon'' = \mathcal{M}[\varepsilon', \mathcal{E}[e_2]]$
$(\ell, (\lvert e_1, e_2 \rvert))$	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], \varepsilon'']$ $\quad where\ \varepsilon' = \mathcal{E}[e_1]$ $\quad\quad \varepsilon'' = \mathcal{M}[\varepsilon', \mathcal{E}[e_2]]$
$(\ell, \texttt{let}\ p = e_1\ \texttt{in}\ e_2)$	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], \varepsilon'']$ $\quad where\ \varepsilon' = \mathcal{E}[e_1]$ $\quad\quad \varepsilon'' = \mathcal{M}[\varepsilon', \mathcal{E}[e_2]]$
$(\ell, \texttt{atomic}\ (e_1, c, e_2))$	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], ATOMIC]$
$(\ell, \texttt{lock ()})$	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], PURE]$
$(\ell, \texttt{ref}\ e)$	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], \mathcal{M}[MUTABLE, \mathcal{E}[e]]]$
$(\ell, \texttt{!}e)$	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], \mathcal{M}[MUTABLE, \mathcal{E}[e]]]$
$(\ell, e_1\ \texttt{:=}\ e_2)$	$\mathcal{E}[\ell] = \mathcal{M}[\mathcal{E}[\ell], \mathcal{M}[MUTABLE, \mathcal{M}[\mathcal{E}[e_1], \mathcal{E}[e_2]]]]$

Figure 4.10: Dataflow-based effect analysis

maps — the effect associated with the function is updated with the effect associated with the execution of its body, and the binding itself is $PURE$ (since capturing a closure is a pure operation). Application is the most complicated rule and the place where control-flow analysis influences the result. During the earlier translation in Figure 4.3, we normalized applications so that a variable is always the target of the application. This normalization allows control-flow analysis to build a mapping from these variables to all of the potential target functions. To compute the overall effect of the application, we merge the effects of all of those potential target functions and the effect associated with evaluating the argument to them.

Tuples, parallel tuples, and **`let`** bindings are all perform a straightforward merge of the effects of their subexpressions. The `atomic` expression always has the same effect. Creation of a lock is pure, and all operations on reference cells have the mutable effect.

This analysis is iterated over the entire program until there are no longer any changes to the effect set. Since there is an upper-bound on the values (`T`) and a fixed number of locations and function binding variables, this analysis is guaranteed to terminate. The `T` value results when the control-flow analysis is unable to determine the potential target of an invocation, due to the lack of precision in that analysis. In practice, this algorithm takes $< 0.1\%$ of compilation time even on our largest benchmarks. For example, on Barnes-Hut, the entire compilation process took 19.4s but the analysis and transformation (discussed in the next section) took only 0.022s.

4.7.4 Transformation

In Figure 4.11, we describe the translation, $\mathcal{R}$, that removes redundant `atomic` operations from the code. The only expression that is modified is the `atomic` expression. In the case where the effect associated with the program point of the expression to execute is $PURE$, we can remove the expression from the wrapper, leaving around a stub that increments the ticket lock but otherwise leaving the expression alone. This stub is required for the serial execution model, which relies on the ticket numbers assigned to the ticket lock. In the transactional execution model, we can omit the `atomic` expression entirely.

4.7.5 Limitations of this analysis

In addition to the restrictions listed earlier in this section, this analysis does not distinguish encapsulated uses of state. The code below exhibits a common idiom in otherwise mutation-free PML code. Rather than passing along a boolean flag to indicate whether any information was updated during complex recursive operations (e.g., computing the effect analysis discussed in this chapter), programs frequently have a single `ref` cell that is reset on each iteration that tracks whether an

$$
\begin{array}{rcl}
\mathcal{R}[\![x]\!] & = & x \\
\mathcal{R}[\![c]\!] & = & c \\
\mathcal{R}[\![\texttt{fun}\, f\, x = e_1\, \texttt{in}\, e_2]\!] & = & \texttt{fun}\, f\, x = \mathcal{R}[\![e_1]\!]\, \texttt{in}\, \mathcal{R}[\![e_2]\!] \\
\mathcal{R}[\![x\ e]\!] & = & x\ (\mathcal{R}[\![e_1]\!]) \\
\mathcal{R}[\![\texttt{(|}e_1, e_2\texttt{|)}]\!] & = & \texttt{(|}\mathcal{R}[\![e_1]\!], \mathcal{R}[\![e_2]\!]\texttt{|)} \\
\mathcal{R}[\![\texttt{atomic}\ (e_1, c, e_2)]\!] & = & \texttt{let}\ () = \texttt{atomic}\ ((), c, ())\ \texttt{in}\ \mathcal{R}[\![e_2]\!] \\
 & & \quad \text{when } \mathcal{E}[\ell] = PURE \\
 & & \quad \text{where } \ell \text{ is the location of } e_2 \\
 & & \texttt{atomic}\ (\mathcal{R}[\![e_1]\!], c, \mathcal{R}[\![e_2]\!]) \\
 & & \quad \text{otherwise} \\
\mathcal{R}[\![\texttt{let}\ p = e_1\ \texttt{in}\ e_2]\!] & = & \texttt{let}\ p = \mathcal{R}[\![e_1]\!]\ \texttt{in}\ \mathcal{R}[\![e_2]\!] \\
\mathcal{R}[\![\texttt{ref}\ e]\!] & = & \texttt{ref}\ (\mathcal{R}[\![e]\!]) \\
\mathcal{R}[\![\texttt{!}e]\!] & = & \texttt{!}(\mathcal{R}[\![e]\!]) \\
\mathcal{R}[\![e_1\ \texttt{:=}\ e_2]\!] & = & (\mathcal{R}[\![e_1]\!])\ \texttt{:=}\ (\mathcal{R}[\![e_2]\!])
\end{array}
$$

Figure 4.11: Translation to remove atomics based on effect analysis.

update occurred. This mutable state does not escape the function and should not prohibit multiple calls to this function from proceeding in parallel, but our current analysis is unable to handle this case.

```
fun compute i = let
  val updated = Ref.new false
in
   ...
end

val result = (| compute 1, compute 2 |)
```

In order to handle this case, we would need to implement a *region-based* effect system, along the lines of the work done in Deterministic Parallel Java [13], though inferring those regions using an extension to our effect analysis instead of their static annotations.

4.8 Evaluation

The experimental setup is identical to that described in the experiments in the previous chapter, in Section 3.7.1. The effects optimization was performed, but since there were no `atomic` operations to remove in these benchmarks, there is no difference in the resulting code.

4.8.1 Account transfer

Our account transfer benchmark is a microbenchmark designed to test the overhead associated with the implementation of parallel mutable state. It is shown in Figure 4.12. As shown in the `transfer` function, this benchmark has almost no work — just one subtraction and one addition. Because of this design, the workload is entirely a measure of the results of heavy contention on the locking infrastructure and parallelism.

We have compiled and tested this benchmark in four configurations. First, we disable all locking and allow random and unprotected access to the array. This result provides an estimate of how fast this particular task could go, if correctness is not a concern. Second, we disable parallelism and locking, simply running in the single-processor sequential configuration. Third, we use *serial* execution, which causes the transfers to occur in the same dynamic order as the single-processor sequential configuration. Finally, we use *transactional* execution, which will run operations concurrently, retrying any operation whose `transfer` method conflicts with another. Results are shown graphically in Figure 4.13 and numerically in Table 4.1.

The results of this benchmark across both 1,000,000 transactions and 10,000,000 transactions have similar patterns. On a single processor, the sequential version (which does not use any of the parallel features) is the fastest implementation, but is quickly overtaken by the unprotected version at even 2 cores. The performance of the sequential version continues to decline because of scheduler overhead in Manticore. When only one thread is performing useful work, the other threads in the system will attempt to steal work from it, interrupting it from making progress. As

Strategy	Number of Processors 1	4	8	16	24	32	40	48
Unprotected	0.176	0.0930	0.0530	0.0330	0.0310	0.0350	0.0420	0.0565
Sequential	0.127	0.128	0.130	0.136	0.145	0.158	0.176	0.208
Serial	0.235	0.401	0.629	0.942	1.00	0.945	1.17	1.78
Transactional	0.328	0.153	0.085	0.0522	0.0463	0.0475	0.0571	0.0712

(a) 1M transactions

Strategy	Number of Processors 1	4	8	16	24	32	40	48
Unprotected	2.79	1.40	0.780	0.422	0.307	0.257	0.236	0.247
Sequential	2.00	2.00	2.01	2.02	2.04	2.07	2.10	2.19
Serial	3.66	5.65	7.91	12.3	13.7	13.1	15.8	19.2
Transactional	5.13	2.24	1.23	0.684	0.514	0.447	0.425	0.603

(b) 10M transactions

Table 4.1: Comparison of mutable state implementation strategies for the account transfer benchmark. Execution times in seconds.

the number of cores increase, so do those attempts to steal work, degrading performance.

As should be expected, the unprotected execution model is the fastest at all other numbers of processors. While it is not a practical execution model (since the unprotected version is not guaranteed to compute a correct answer), it does help us to measure the overhead of the transaction model. The serial execution model, where lock order is explicitly specified, is faster than the transactional model only where there is a single processor. This behavior is because under the serial model there are both no contended locks and none of the logging behavior and additional value checking that is required in the transactional version. Once there are many processors, though, the transactional model is faster than both the serial and sequential versions.

Graphically in Figure 4.14 and numerically in Table 4.2, we investigate the performance trade-off in the size of the array chunk that we track for edits. As mentioned earlier, the smaller the size, the more memory is taken (in number of locks allocated), but it also reduces the odds of a conflict. When the size is larger, the transaction commit time may increase, as lock contention increases when concurrent tasks attempt to commit their changes to elements that share the same

	Number of Processors							
Locking size	1	4	8	16	24	32	40	48
1	5.13	2.24	1.23	0.684	0.514	0.447	0.425	0.603
10	5.06	2.26	1.27	0.727	0.561	0.502	0.502	0.826
50	4.99	2.34	1.39	0.910	0.915	1.14	1.45	2.23
100	5.00	2.46	1.55	1.23	1.64	2.11	2.91	4.60

Table 4.2: Comparison of array lock range sizes in the account transfer benchmark. Execution times in seconds.

chunk. As is shown in the table, for this benchmark — which has a very large ratio of transactional operations to work — smaller chunk sizes result in the best performance. This performance comes at the cost of memory, as there is one lock object per chunk, and in the degenerate case (one lock per account), the size of the lock array is larger than the size of the account array.

4.8.2 Minimax with time limit

In Section 3.7.4, we introduced the minimax benchmark. In this benchmark, we performed an α-β search over a game tree, trying to find a move that optimized the expected outcome. In that benchmark, we limited our depth of investigation to 4 moves, and had corresponding times based on the CPU and processor. But what if we wanted to instead limit ourselves to a specific amount of time? One method for handling this problem is to spawn parallel searches from 1 move to the full game board (16 moves) and to cut off execution after a fixed amount of time. While we could return all of the results sort them, in this implementation we investigate the approach of having a single global variable with the result and updating that variable. In the following example, we have sample code that stores the best result so far into a reference cell, as shown below:

```
val best = ref EMPTY
fun update_best(new_move, score) =
  case !best
   of EMPTY => (best := MOVE(new_move, score))
    | MOVE(_, old_score) => (
```

	Number of Processors							
Timeout	1	4	8	16	24	32	40	48
100ms	7,200	22,000	38,000	60,000	69,000	66,000	61,000	40,000
250ms	18,000	55,000	95,000	160,000	181,000	190,000	180,000	150,000
500ms	36,000	108,000	190,000	310,000	380,000	390,000	380,000	340,000

Table 4.3: Comparison of the maximum tree number of moves searched versus timeout limit across several timeout values.

```
if (score > old_score)
then (best := MOVE(new_move, score))
else ()
```

We have augmented the minimax benchmark to do exactly the same thing, and Table 4.3 evaluates the number of unique board depths that can be evaluated versus the number of processors at a given time threshold using the transactional execution model. This evaluation is shown graphically in Figure 4.15. This facility not only allows more states to be evaluated than the default memoized strategy (at depth 4, counting from 0, we visit at most $16 * 15 * 14 * 13 * 12 = 524,160$ states, not including shared states eliminated by memoization), but also allows the user to set a fixed amount of time for their evaluation. Note that for the most part values increase up to around 40 processors, but past that point, it tends to diminish because of some extremely poor runs. Generally the best runs are the same, but the lowest quartile is significantly worse at higher processors because of occasional scheduling orders that cause limited sharing of memoized work before spawning off huge amounts of work that could ultimately have been shared.

4.8.3 STMBench7

The STMBench7 benchmark was created in 2007 to measure the performance of transactional memory system implementations [33]. This benchmark simulates random workloads over a large number of objects, simulating an in-memory object graph for a CAD/CAM system. We evaluate our implementation of transactional memory through a benchmark that re-creates these objects and

	Number of Processors							
Strategy	1	4	8	16	24	32	40	48
Unprotected	0.685	0.688	0.694	0.715	0.750	0.807	0.892	1.01
Transactional	0.964	0.969	0.975	1.07	1.04	1.09	1.31	1.61

Table 4.4: Comparison of array lock range sizes in the STMBench7 benchmark on 1,000,000 transactions. Execution times in seconds.

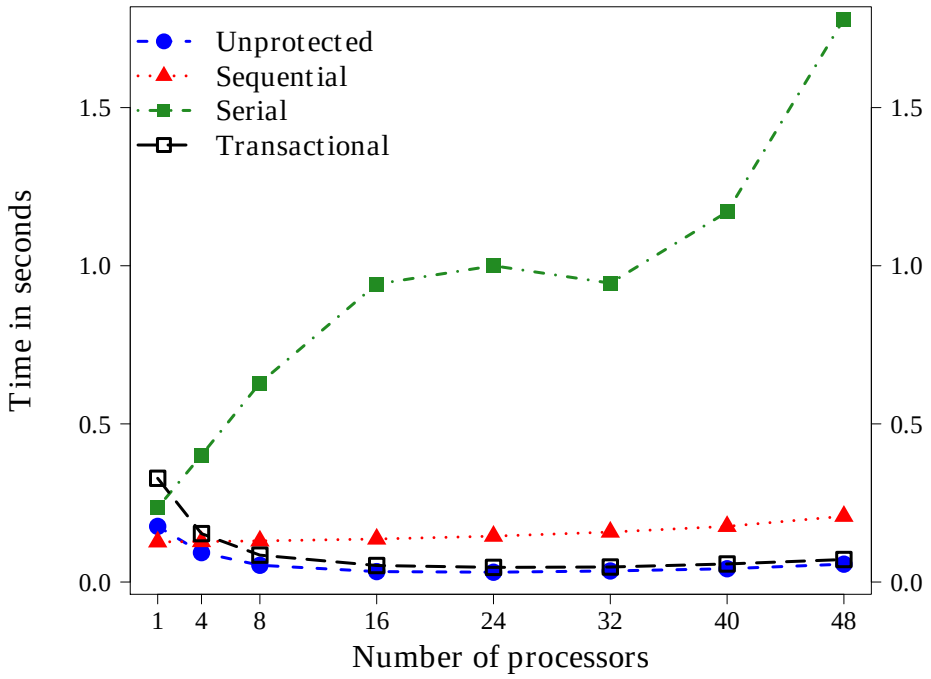
uses a workload similar to their balanced workload — 60% transactions that just read state, and 40% transactions that mutate state.

In this benchmark, there are a large number of objects. At the top, there are 500 composite objects. Below each of these composites are 3 local objects, each of which has 200 elements. There are additionally 600 connections between these elements. Different ports of this program have used a variety of implementations, from object-oriented representations (in the original Java version) to a set-based version in Ziarek's Concurrent ML-based implementation [69]. Since our transactional memory implementation is based on large arrays and reference cells, we have simply used arrays to hold the data and indexes into those arrays for the various items.

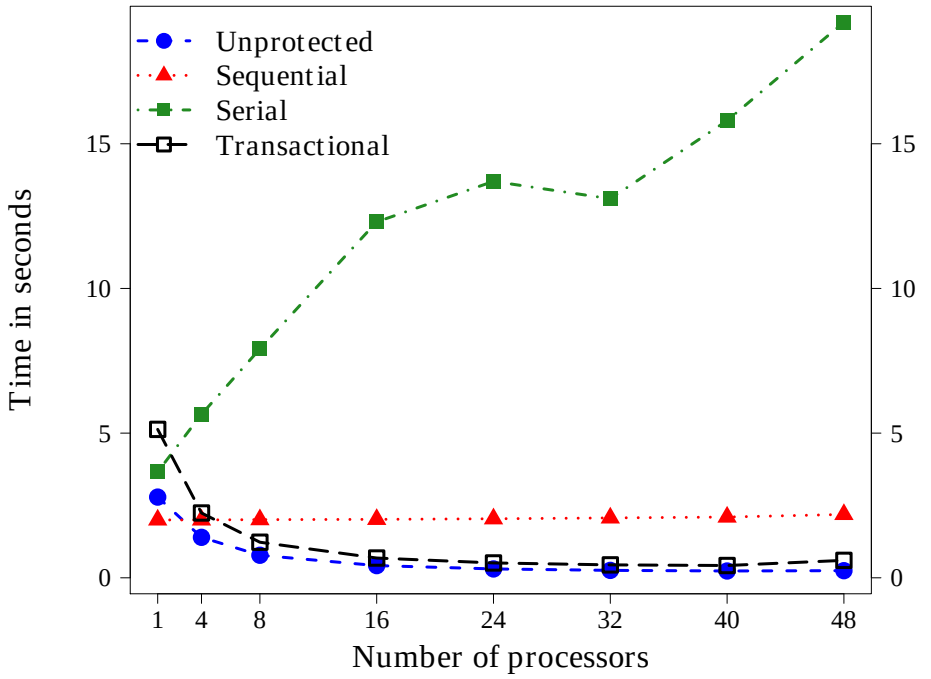
In Table 4.4, we provide an evaluation of our transactional implementation against a lock-free, unprotected implementation, to understand how much overhead our system imposes. We ran against the number of objects as described in the previous paragraph, with 1,000,000 parallel transactions, on a varying number of processors from 1 to 48. Even with a very simple atomic strategy with coarse locking — global transactional locks for updates — we incur less than a factor of two slowdown in this benchmark, with significantly worse performance at larger number of processors because of the additional interference between tasks.

```
structure Transfer = struct
  val numAccounts = 10000
  val accounts = Array.tabulate (numAccounts, fn i => 1000)
  val transfers = Array.tabulate (10001,
                        fn i => Rand.inRangeInt(0, numAccounts))

  fun transfer (index) = let
      val index = index mod Global.numAccounts
      val v = 100
      val i = Array.sub (transfers, index)
      val j = Array.sub (transfers, (index+1))
      val i' = Array.sub (accounts, i)
      val j' = Array.sub (accounts, j)
  in
      Array.update (accounts, i, i'-v);
      Array.update (accounts, j, j'+v)
  end
end

fun doTransfers (start, n) =
    if (n = 1)
    then (Transfer.transfer (start))
    else let
        val half = n div 2
        val _ = (| doTransfers (start, half),
                   doTransfers (start + half, half) |)
    in
        ()
    end

val _ = doTransfers (0, problemSize)
```

Figure 4.12: Parallel account transfer code.



(a) 1M transactions



(b) 10M transactions

Figure 4.13: Comparison of mutable state implementation strategies for the account transfer benchmark.

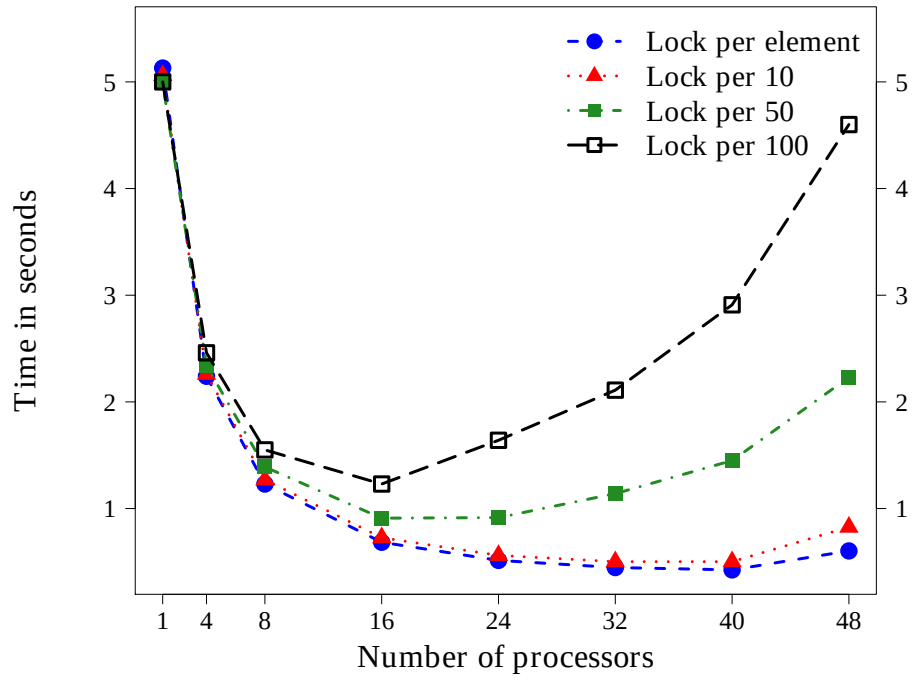


Figure 4.14: Comparison of array lock range sizes in the account transfer benchmark.

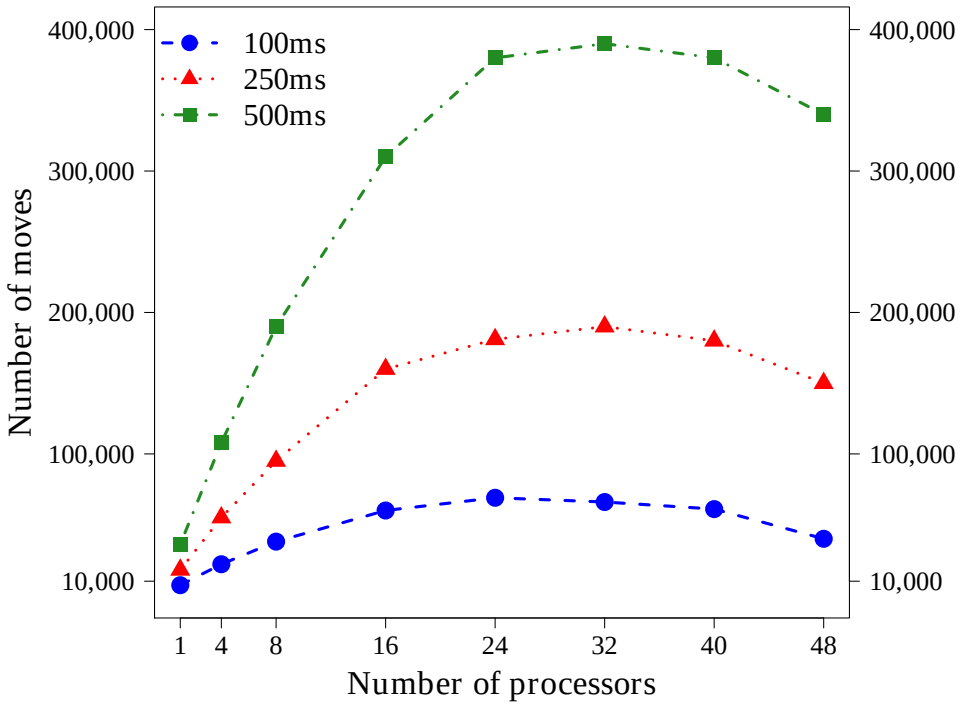


Figure 4.15: Comparison of the maximum tree number of moves searched versus timeout limit across several timeout values.

4.9 Local reasoning

Local reasoning refers to the ability to reason about the behavior of a function and its usage of the heap in isolation from any other functions — including concurrent functions — that mutate the heap [59]. In order to preserve local reasoning, we have based our mutable state design around the model of linearizability [42, 41]. In this model, every function invocation appears to the rest of the system as if it occurred instantaneously — none of its internal uses of state are visible to any concurrently executing code. In a sequential context without concurrency, this linearizability property is trivial. Callers cannot distinguish intermediate states of their callees because of the call/return discipline. Further, there are no parallel or concurrent invocations, so there is no other code executing that could discern intermediate states.

From the point of view of shared memory and parallelism, linearizability requires that any two concurrent operations provide results consistent with their shared memory operations having occurred either entirely before or after one another. This requirement is handled directly in both of our execution models.

Serial In the serial state model, each parallel expression will be executed in sequential program order if it uses mutable state. The sequential program order is the same order as the sequential execution of the program without any parallelism. This execution model is stricter than the linearizable model, as it provides a guarantee of sequential in-order execution with respect to changes in mutable state, at the cost of parallelism.

Transactional In this model, the first expression to complete its work will successfully commit its changes to shared memory, irrespective of program order. But, any expressions executed in parallel will either see all of the changes or none of the changes performed by that expression. Further, any conflicting changes will cause the second writer to abort and retry their changes. This model preserves local reasoning because within the scope of a sequential function, only global

changes committed before the call can change its behavior. No changes made concurrently will affect its behavior, other than to cause a semantically invisible abort and re-execution.

In both of these models, the implementation enforces these properties and there are no annotations required in the code.

4.9.1 Example

In games such as Pousse [7], the problem is to come up with the best possible move in a limited amount of time. One key part of the inter-thread communication in this benchmark is to update the "best known move." In the example below, the code `update_best` checks the best move so far to determine if it should be updated and, if so, replaces the value in the global mutable state. In all execution models, the programmer can reason locally about the correctness of the function `update_best` without worrying about either other writers to the variable `best` or concurrent executions.

```
val best = ref EMPTY
fun update_best(new_move, score) =
  case !best
   of EMPTY => (best := MOVE(new_move, score))
    | MOVE(_, old_score) => (
      if (score > old_score)
      then (best := MOVE(new_move, score))
      else ()
```

The code below that calls `update_best` has different potential executions under the two execution models, but in both cases the result will be the same.

```
(| update_best(move1, 27), update_best(move2, 30) |)
```

Serial In this model, the two parallel subexpressions both perform mutations of state, so they must run in serial program order. This code will execute as if there were no parallel expressions, simply executing them in sequence. That is, it will execute the same as if the code had been written:

```
(update_best(move1, 27), update_best(move2, 30))
```

Transactional In the transactional model, the second expression may begin to execute before the first one. There are four permissible interleavings of these expressions under this model, which are handled semantically in the following manner:

1. The first expression runs to completion before the second begins. In this case, the second expression will then run to completion.

2. Both run concurrently, with the first committing before the second. Upon attempt to complete the second expression, there will be a conflict on the reference cell corresponding to `best`, so the second expression will be aborted and re-executed.

3. The first and second run concurrently, with the second committing before the first. Upon attempt to complete the first expression, there will be a conflict on the reference cell corresponding to `best`, so the first expression will be aborted and re-executed.

4. The second expression runs to completion before the first begins. In this case, there is no work for the system to do.

These examples under the different execution models describe the full set of permissible behaviors and their expected runtime behavior. The system, though, may limit those possible executions in order to simplify or remove the need for conflict handling.

4.10 Conclusion

Mutable state is an extension to the language of Parallel ML (PML) that simply adds back features that were originally in Standard ML. By preserving the syntax familiar to Standard ML programmers and providing an execution model that preserves local reasoning, we have provided a mechanism that does not burden the programmer with annotations or type system hurdles.

The evaluation demonstrates that the transactional implementation of this feature introduces little overhead and scales well across a range of processors. We have shown this behavior both on a microbenchmark and on a larger one, additionally providing better performance than some other implementations.

This programming model does not allow unrestricted access to mutable state. Because we implicitly provide atomic guarantees around any concurrent expressions that access mutable state, for programs whose correctness does not change in the absence of locking our model introduces unnecessary overhead. For example, an implementation of memoization as presented in Chapter 3 could be written using mutable state, but rather than sharing results between parallel threads to reduce re-execution, this implementation would require parallel threads to complete and commit before their results are shared with other threads. That implementation of memoization would both dramatically reduce sharing, due to the restriction of computed results until transaction commit, and increase synchronization overhead.

CHAPTER 5

SEMANTICS

In this chapter, we define an operational semantics that specifies the behavior of a subset of the Parallel ML (PML) language, including parallelism, memoization, and mutable state. This formalism is in the style of Moore and Grossman's work on a small-step operational semantics for transactions [56]. We have several major differences from their presentation:

- Memoization is added, using the syntax of Harper *et al.* [1], and its interaction with transactional memory is specified.
- Rather than their three forms of threads, we have a single fork-join model of parallelism, which substantially simplifies their thread pool representation, though it reduces the flexibility of the programming model.
- We do not provide a type system and associated type-safety proofs, leaving that to future work.

5.1 Syntax

Figure 5.1 contains the expression forms in the reduced version of the PML language that we use in this formal system. This language is further simplified from those presented in Figure 3.1 and Figure 4.1 in order to reduce the number of purely mechanical rules. Variables, x, refer to bound variables in **`let`** expression and parameters of functions, which are distinguished from function identifiers, f. We assume that variables names are distinct. Locations are the value that results from the allocation of a reference cell, and serve as the argument to both retrieve and set the current value of a reference cell. The **`atomic`** keyword wraps any parallel expression that may use mutable state to ensure that it is executed as if it had exclusive access to the heap. The **`let`** form is used to introduce fork-join parallelism and bind the results to a pair of variables.

Memoization is provided through the **mfun** keyword which otherwise has the same syntax as a normal function declaration. This syntax is identical to that used in the recent work on selective memoization by Harper *et al.* [1] and that presented in Chapter 3. In this language, function definitions are closed (*i.e.*, they have no free variables) and may not be recursive.

The **inatomic** and **memo** forms are not surface syntax, but are administrative expressions introduced by the rules in the operational semantics when entering an **atomic** block and invoking a memoized function that does not have a previously recorded value, respectively.

e	::=	x	variables
	|	c	constants
	|	f	function identifiers
	|	l	locations
	|	**fun** $f(x).e$	functions
	|	**mfun** $f(x).e$	memoized functions
	|	$e_1\, e_2$	application
	|	**seq**$(e_1,\, e_2)$	sequencing
	|	**ref** e	reference cell allocation
	|	$!e$	access the value of a reference cell
	|	$e_1 := e_2$	update the value of a reference cell
	|	**atomic** e	enter an atomic block
	|	**inatomic**$(a,\, e)$	evaluation within an atomic block
	|	**memo**$(f,\, e_1,\, e_2)$	evaluation within a memoized function, `f`
	|	**let** $(x,\, y) = (\|e_1,\, e_2\|)$ **in** e_3	parallel evaluation with binding
v	::=	c	constants
	|	l	locations
	|	**fun** $f(x).e$	functions
	|	**mfun** $f(x).e$	memoized functions

Figure 5.1: Syntax of the reduced Parallel ML language

$$\begin{aligned} \mathrm{H} &: l \xrightarrow{\mathit{fin}} v \\ \mathrm{M} &: (f * v) \xrightarrow{\mathit{fin}} v \end{aligned}$$

Figure 5.2: Finite maps

These rules build up two finite maps, shown in Figure 5.2. The first, H, is a map from from

each of the allocated heap locations l to a value, v. The second, M, is a memo table from a function identifier and input value to a cached output value. A value is retrieved with the syntax $\mathrm{H}(l)$ and is updated by $\mathrm{H}(l) \leftarrow v$. If a value is unset in the finite map, we denote that with $\mathrm{H}(l) \uparrow$.

5.2 Operational semantics

This semantics ensures two dynamic constraints on the execution of programs:

- The mutable heap, H, may only be accessed by one thread at a time.
- During the evaluation of a memoized function, the mutable heap is inaccessible.

Program states are made of three parts, separated by semicolons:

$$\mathrm{m}; \mathrm{a}; e$$

The first part, m, indicates whether we are currently evaluating an expression that is within a memoization function context, ▪, or outside of one, with the ▫ symbol. The second is a similar indicator for **`atomic`** tracking. In this case, the • symbol indicates that there is a concurrent thread currently executing in an **`atomic`** block, and ∘ indicates that there is no other thread in one. When the current program context is set to ∘, any rules that access the heap may be performed, as it is the case that there may be other transactions executing but the current evaluation context has exclusive access to the heap. These values are also shown in Figure 5.3. Finally, e is an expression, from the language in Figure 5.1.

$$\begin{array}{rcll} \mathrm{a} & ::= & \circ \mid \bullet & \text{atomic operation tracing} \\ \mathrm{m} & ::= & \square \mid \blacksquare & \text{memoization operation tracing} \end{array}$$

Figure 5.3: Program states

Figure 5.4 provides the rules for function application, both for normal functions and memoized functions. BETA-FUN simply performs capture-avoiding substitution in the body of the function

$$\frac{}{\mathrm{m};\mathrm{a};(\textbf{fun}\, f(x).e)\, v \to \mathrm{m};\mathrm{a};e[v/x]} \quad \text{(BETA-FUN)}$$

$$\frac{\mathrm{M}(f, v_1) = v_2}{\mathrm{m};\mathrm{a};(\textbf{mfun}\, f(x).e)\, v_1 \to \mathrm{m};\mathrm{a};v_2} \quad \text{(BETA-MFUN-1)}$$

$$\frac{\mathrm{M}(f, v_1) \uparrow}{\mathrm{m};\mathrm{a};(\textbf{mfun}\, f(x).e)\, v_1 \to \mathrm{m};\mathrm{a};\textbf{memo}(f,\, v_1,\, e[v_1/x])} \quad \text{(BETA-MFUN-2)}$$

$$\frac{\blacksquare;\mathrm{a};e \to \blacksquare;\mathrm{a}';e'}{\mathrm{m};\mathrm{a};\textbf{memo}(f,\, v,\, e) \to \mathrm{m}';\mathrm{a}';\textbf{memo}(f,\, v,\, e')} \quad \text{(MEMO-1)}$$

$$\frac{}{\mathrm{m};\mathrm{a};\textbf{memo}(f,\, v_1,\, v_2) \to \mathrm{m};\mathrm{a};v_2 \qquad \mathrm{M}(f, v_1) \leftarrow v_2} \quad \text{(MEMO-2)}$$

Figure 5.4: Rules for function application including memoization.

with the argument value. The syntax $e[v/x]$ means to replace all occurrences of x in the expression e with the value v. The first memoized function rule, BETA-MFUN-1, covers the case where the memo table associated with the function f already has a result value associated with v_1. In that case, the rule does not execute the body expression but instead returns the cached value. In the second rule, BETA-MFUN-2, there is no value associated with the argument in the memo table, so the function body will be evaluated. First, though, we wrap the body of the function in the **memo** form, which both provides the name of the function and the original argument value for later caching purposes and supports tracking the evaluation of expressions within a memoized function.

The MEMO-1 rule steps the evaluation within the body of a memoized function. Note that when evaluating the body expression, we require that only rules that are safe during memoization, *i.e.*, allowing ▪, are permitted. This condition is what prevents allocating, reading, or setting mutable state within a memoized function. See Figure 5.6 and rules ALLOC, GET-2, and SET-3, along with their administrative forms in Figure 5.8, all of which require that the ▫ flag is set in the program state.

$$\frac{\mathrm{m}; \mathrm{a}; e_1 \to \mathrm{m}'; \mathrm{a}'; e_1'}{\mathrm{m}; \mathrm{a}; \mathbf{let}\,(x,\,y) = (|e_1,\,e_2|)\,\mathbf{in}\,e_3 \to \mathrm{m}'; \mathrm{a}'; \mathbf{let}\,(x,\,y) = (|e_1',\,e_2|)\,\mathbf{in}\,e_3} \quad \text{(LET-1)}$$

$$\frac{\mathrm{m}; \mathrm{a}; e_2 \to \mathrm{m}'; \mathrm{a}'; e_2'}{\mathrm{m}; \mathrm{a}; \mathbf{let}\,(x,\,y) = (|e_1,\,e_2|)\,\mathbf{in}\,e_3 \to \mathrm{m}'; \mathrm{a}'; \mathbf{let}\,(x,\,y) = (|e_1,\,e_2'|)\,\mathbf{in}\,e_3} \quad \text{(LET-2)}$$

$$\frac{}{\mathrm{m}; \mathrm{a}; \mathbf{let}\,(x,\,y) = (|v_1,\,v_2|)\,\mathbf{in}\,e_3 \to \mathrm{m}; \mathrm{a}; e_3[v_1/x, v_2/y]} \quad \text{(LET-3)}$$

Figure 5.5: Rules for parallel language features, with non-deterministic evaluation order of subexpressions.

$$\frac{l \notin \mathrm{dom}(\mathrm{H})}{\square; \mathrm{a}; \mathbf{ref}\,v \to \square; \mathrm{a}; l \qquad \mathrm{H}(l) \leftarrow v} \quad \text{(ALLOC)}$$

$$\frac{}{\square; \mathrm{a}; !l \to \square; \mathrm{a}; \mathrm{H}(l)} \quad \text{(GET-2)}$$

$$\frac{}{\square; \mathrm{a}; l := v \to \square; \mathrm{a}; v \qquad \mathrm{H}(l) \leftarrow v} \quad \text{(SET-3)}$$

Figure 5.6: Rules for mutable state features.

Parallelism is introduced in Figure 5.5. This feature combines the parallel tuple expression from Parallel ML with a binding form. Unlike all of the other rules in this semantics, the LET-1 and LET-2 rules can both be eligible for evaluation on the same expression and introduce non-determinism. Once both of the expressions have been full reduced to values, we substitute them for their bound variables in the body expression with LET-3.

The operation of mutable state is shown in Figure 5.6. The rule ALLOC allocates a new label in the heap, associates the value with it, and then returns that label. This rule may not be evaluated during evaluation of a memoized function. GET-2 reads a value from the heap and returns it, whereas SET-3 updates the heap with the newly assigned value.

Finally, Figure 5.7 describes the operation of this language with respect to the atomic keywords. Their design — and the atomic markers $\bullet$ and $\circ$ — provides exclusive access to the heap for a single expression at a time, while still supporting nested **`atomic`** usage and parallelism. The first rule,

$$\frac{}{\mathrm{m}; \circ; \texttt{atomic}\, e \to \mathrm{m}; \bullet; \texttt{inatomic}(\circ, e)} \quad \text{(ATOMIC)}$$

$$\frac{\mathrm{m}; \mathrm{a}; e \to \mathrm{m}'; \mathrm{a}'; e'}{\mathrm{m}; \bullet; \texttt{inatomic}(\mathrm{a}, e) \to \mathrm{m}; \bullet; \texttt{inatomic}(\mathrm{a}', e')} \quad \text{(INATOMIC-1)}$$

$$\frac{}{\mathrm{m}; \bullet; \texttt{inatomic}(\circ, v) \to \mathrm{m}; \circ; v} \quad \text{(INATOMIC-2)}$$

Figure 5.7: Rules for the features related to atomicity.

ATOMIC, can only step when there are no other expressions concurrently operation with atomic access to the heap, which is denoted by $\circ$. When this rule steps, it changes the outer atomic marker to $\bullet$, indicating that there is a subexpression with atomic access to the heap. This rule wraps the expression in the **`inatomic`** keyword and the atomic marker $\circ$, indicating that it has access to the heap.

The rule INATOMIC-1 steps the expression within the **`inatomic`** expression. Note that it uses the outer memoization, heap, and function caches, but the *inner* atomic marker. This feature is also what enables nested entry into an **`atomic`** block. When a nested block is entered, the outer program context is then instead the *inner* context of the **`inatomic`** block, which prevents any other expressions in a parallel executions from gaining exclusive access to the heap until the current nested block is finished. The last rule, INATOMIC-2, handles termination of the atomic expression. When it terminates, it also resets the outer atomic marker to $\circ$, indicating that another expression may now gain access to the heap.

Figure Figure 5.8 contains all of the administrative rules for this semantics. Each of these rules wrap an evaluation step with its context.

$$\frac{\mathrm{m}; \mathrm{a}; e_1 \to \mathrm{m}'; \mathrm{a}'; e_1'}{\mathrm{m}; \mathrm{a}; e_1\, e_2 \to \mathrm{m}'; \mathrm{a}'; e_1'\, e_2} \quad \text{(APP-1)}$$

$$\frac{\mathrm{m}; \mathrm{a}; e_2 \to \mathrm{m}'; \mathrm{a}'; e_2'}{\mathrm{m}; \mathrm{a}; v\, e_2 \to \mathrm{m}'; \mathrm{a}'; v\, e_2'} \quad \text{(APP-2)}$$

$$\frac{\square; \mathrm{a}; e \to \square; \mathrm{a}'; e'}{\square; \mathrm{a}; \mathbf{ref}\, e \to \square; \mathrm{a}'; \mathbf{ref}\, e'} \quad \text{(REF)}$$

$$\frac{\square; \mathrm{a}; e \to \square; \mathrm{a}'; e'}{\square; \mathrm{a}; !e \to \square; \mathrm{a}'; !e'} \quad \text{(GET-1)}$$

$$\frac{\square; \mathrm{a}; e_1 \to \square; \mathrm{a}'; e_1'}{\square; \mathrm{a}; e_1 := e_2 \to \square; \mathrm{a}'; e_1' := e_2} \quad \text{(SET-1)}$$

$$\frac{\square; \mathrm{a}; e_2 \to \square; \mathrm{a}'; e_2'}{\square; \mathrm{a}; v := e_2 \to \square; \mathrm{a}'; v := e_2'} \quad \text{(SET-2)}$$

$$\frac{\mathrm{m}; \mathrm{a}; e_1 \to \mathrm{m}'; \mathrm{a}'; e_1'}{\mathrm{m}; \mathrm{a}; \mathbf{seq}(e_1, e_2) \to \mathrm{m}'; \mathrm{a}'; \mathbf{seq}(e_1', e_2)} \quad \text{(SEQ-1)}$$

$$\frac{}{\mathrm{m}; \mathrm{a}; \mathbf{seq}(v, e_2) \to \mathrm{m}; \mathrm{a}; e_2} \quad \text{(SEQ-2)}$$

Figure 5.8: Administrative rules.

CHAPTER 6

RELATED WORK

The challenge of creating a programming model and language implementation that supports parallelism, communication between threads, and high performance has been the subject of research for as long as there has been parallel computing hardware. In this work, we focused on two smaller problems: adding memoization and explicit mutable variables.

Both memoization and hash tables have been the subject of significant prior work. Our primary contribution is in combining memoization with hash tables and improving the performance of both by taking advantage of the looser semantic guarantees required by memoization.

Mutable state in the context of parallel languages has been widely studied and is available in a large number of different forms in various languages. Our work is unique in its combination of no required annotations, a linearizable execution model, and scalable parallel performance.

6.1 Memoization

Memoization of functions was originally proposed by Donald Michie [55]. In that work, he used the example of factorial, remembering a fixed number of applications, and discarding based on the least recently used value. William Pugh extended this work to an incremental evaluator, in the context of using it to incrementalize computation and evaluation [62]. Pugh investigated the use of a fixed-size hash table and focused on cache replacement strategies that work well in that context and are smarter than LRU, though not applicable for all problems [61]. Our work differs from that by relying instead on dynamically-sized hash tables, which avoid some of the pathological cases that lead Pugh to investigate smarter replacement strategies for individual problems.

Hughes extended memoization to the context of lazy languages [43], relying on pointer equality and lazy evaluation to determine equality when performing a lookup within the hashed memo functions. We do not handle complex arguments to functions in this work.

Ziarek, Sivaramakrishnan, and Jagannathan investigated memoization in the context of message-passing concurrency [69]. Unlike our work, which prohibits access to mutable state within a memoized function, in addition to the function arguments they track and record the actions related to threads and communication performed by the function. Their system will return the same value when they determine that the call site satisfies all of those same constraints with respect to the concurrency features.

An extension of their earlier work on self-adjusting computation, Acar, Blelloch, and Harper specified an extension of Standard ML with *selective memoization* [1]. Their work differs from ours and is more similar to the work by Ziarek *et al.* in that it also tracks the control-flow path through the function in order to potentially increase the memoization cache hit ratio. They suggest that this approach would be able to ignore unrelated arguments and provide further gains, whereas our work specifies that the argument to the function is identical to the value in the memoization cache.

6.2 Hash tables

The idea of a dynamically growing hash table for in-memory storage of data originates with Larson [48]. In that work, he extended the earlier research on external hashing in file systems and external data storage. His system used fixed-sized segments that were pointers to linked lists of elements with incremental growth of the table by a single fixed segment when the number of records in the table exceeded a percentage of the current capacity. It also incrementally updated elements by expanding the table one bucket at a time into the newly-allocated space.

The implementation of dynamic hashing in Icon extended that approach for use in a language where nearly everything is either a table or a set — and both of those use a dynamic hash table for their implementation [32]. They adapted Larson's scheme to start with a smaller initial table and segment size, but then grow it by a power of two each time the table needs to increase. This change supported their need to have many tables in memory at once, some of which might remain small.

Shalev and Shavit extended this implementation in a concurrent context with the idea of *recursive split-ordering* to order the elements in each bucket so that they are in binary digit order of their hash value [65]. That order means that if the sizes of the underlying table are always grown by powers of two, then splitting an entry only involves a single split for each list in each bucket. Their implementation is also linearizable and lock-free, which makes it both semantically and performance-size ready for use in a parallel setting. Larson recently extended that work to implement table contraction and to stabilize performance across a variety of work sets by returning to the incremental bucket splitting technique of the original work [68]. Our implementation differs from both of these concurrent implementation by removing both locking and atomic operations. Since the surface language feature that uses these hash tables — memoization — does not require correctness from the underlying table, we avoid synchronization between concurrent operations.

6.3 Mutable State

Research in the area of combining mutation and parallelism has focused either on fully automatic approaches that preserve determinism at the cost of performance or completely manual approaches that provide good performance but require significant annotations. Our approach to mutable state is automatic, but relaxes determinism to something still better than unrestricted non-determinism without requiring manual annotations or locking. This strategy — providing a programming model without explicit transactions, locks, or annotations while still providing a linearizable semantics and scalable performance — is the fundamental difference between our approach to mutable state and the prior art.

6.3.1 Speculation

Pingali and his colleagues have described the *Galois Approach* [46], which extends imperative, object-oriented programming with two optimistic iterators: one for unordered collections and one

for partially-ordered collections. These iterators are hints to the system that multiple instances of the loop bodies may be executed in parallel. The correctness of this approach depends on the libraries that implement the collection abstractions. If two operations on a collection are commutative, then they may be executed in parallel, since the order in which they are executed does not affect the result. Since Galois optimistically executes the loop bodies in parallel, it may have to undo the effects of a loop body when a commutative conflict is detected. Thus the collection abstractions must define what operations commute (*e.g.*, `add`(x) commutes with `add`(y) when $x \neq y$), and provide undo operations. This approach to parallel execution in the presence of mutable data structures is one implementation of *speculation*.

The main drawback to the Galois approach is that implementing the collection abstractions is challenging, since they must correctly detect interference and provide a rollback mechanism. These collection abstractions provide a form of parallelism similar to our mutable state, as each of the operations that are marked as commutative are similar to our atomic regions. If commutativity annotations are used, a programmer can explicitly write similar code to that performed by our translation and effect analysis for mutable state that will reproduce the behavior of a program in Parallel ML.

6.3.2 Imperative parallel languages

The JAVA memory model defines a consistency model that is relaxed (weaker) compared to sequential consistency [47, 52]. This model is based on a set of constraints derived from program order and data type. One goal of this memory model is to provide sequential consistency to correctly-synchronized programs and define the behavior of programs with data races to make the allowable executions clear to language implementers and programmers. These data races require explicit barriers on memory operations. This requirement forces programmers implementing concurrency-safe objects manually verify that the methods are linearizable to avoid race conditions, as there is no explicit language or tool support currently available to verify object safety.

The proposed C++ memory model is similar to the JAVA memory model in starting from a weaker model than sequential consistency [14]. While this model also provides sequential consistency for atomic operations, it differs from the JAVA model by providing no semantics for data races. Programs written in this memory model are non-deterministic and, again, require significant programmer effort to ensure correctness without, in general, the aid of static verification.

Single-Assignment C (SAC) was extended by Herhut *et al.* with a mechanism for extending data-parallel loops with side effects [39]. Their design serializes the effects in any given iteration and serializes the effects between iterations, but does not prescribe a specific order of the iterations. This implementation has similar semantics to our implementation of arrays and mutable state.

The FX language introduced the use of region and effect inference to identify regions of a program that could be executed in parallel because they were guaranteed to be non-interfering [29, 30]. In their approach, users annotate the program with regions and the compiler infers the effects using a unification-based approach (similar to type inference). They then produced a dataflow graph, identifying the parts of the program that could potentially be executed in parallel. In our work we do not use regions to identify disjoint portions of the heap. We do perform a similar effect inference, however, and rely on control-flow analysis to improve the precision of our effect analysis over the unification-based approach.

Deterministic Parallel Java (DPJ) provides a strictly data-race-free, explicitly parallel language [13]. This system relies on a type-based effect system in concert with method-level parallelism annotations (requiring annotations in an average of 12.3% of the lines of code) to guarantee a parallel execution that is the same as some sequential execution. While that approach of sequential execution is similar to ours, we omit the need for user annotations, at the cost of less scalable parallelism. For example, in DPJ, the user can mark parallel subtasks as working on disjoint portions of the heap, allowing parallelism without synchronization. In our implementation, we require a check of the transaction log at commit. However, our fundamental approaches to non-deterministic behavior are similar — in both DPJ and our system, non-deterministic behavior

is still sequentially consistent.

6.3.3 Transactional memory

Software Transactional Memory (STM) provides an interface that implements a memory consistency model on reads and writes across multiple threads of execution [66]. This model is usually exposed as a new language feature — an **`atomic`** block, along with failure and retry handlers. In the space of declarative programming, Harris *et al.* showed how to cleanly integrate STM with the type system of a functional language [37] and integrated this work in the Glasgow Haskell Compiler (GHC). One challenge with this work is that since the user defines the size of the atomic regions, there is a granularity problem associated with identifying the transaction size that provides the best results [60], similar to our investigation of the performance tradeoffs in lock sizes in Section 4.8.1. While our work does not provide a full solution to this problem, we have attempted to engineer our system to support small transaction sizes and to rely on the effect analysis to reduce the size of transactions.

When compared to arbitrary locking, the transactional model limits the performance of STM in situations with heavily-shared objects. Even in cases where two transactions against an object could commute in theory, as when adding distinct items to a set, the shared writes to the same object will cause one of those two transactions to be aborted. Transactional boosting is a technique for annotating objects with commutativity and inverses for methods, enabling those objects to be used concurrently when the standard transactional model would not normally permit it without aborting one [40]. As in the Galois approach above, significant expertise and work are required from the implementer of these *boosted objects*. Coarse-grained transactions provide a formalism proving the preservation of atomic semantics when transactional models are extended to reduce false sharing as in models like transactional boosting [45]. This work does not support transactional boosting.

6.3.4 Logic programming

Logic programming provides one of the earliest examples of implicit parallelization implementation techniques and their integration with explicit threading and shared state [34]. Three forms of implicit parallelism are readily apparent in logic programs:

- And-parallelism is over the different goals to solve
- Or-parallelism is over the different clauses to solve
- Unify-parallelism is between the sets of terms to be instantiated[1]

Different logic programming language implementations have provided a variety of language constructs for explicit parallelism and integrated these constructs with the implicit parallelism. The **`cut`** and cavalier **`commit`** operations cause the search of a logic program to terminate early along branches of exploration. The **`cut`** operation is similar to the explicit threading, shared state design because it requires synchronization of all prior **`cut`** threads to return the first one that would have returned in the sequential execution [50]. The cavalier **`commit`** model, which does not require synchronization with prior work, is an example of the explicit threading, shared state design.

In the presence of shared variables, logic programming implementations, such as Ciao, analyze the goals for independence of variables and only evaluate them in parallel if the goals are both independent and the parallel execution does not change the search space [26]. The blackboard language feature allows shared state between parallel threads, with access synchronized in implementation-specific ways [34].

6.3.5 Automatic

The language Id and its successor pH, a dialect of Haskell, provide automatic parallelization of user programs [57, 58]. This approach generates fine-grained parallel work, conceptually creating

1. Unify-parallelism is not often used in practice due to the extremely fine-grained parallelism [34].

a new thread for each available reducible expression except for those guarded by a conditional. The Manticore approach relies on user annotations for identification of potential parallelism.

These languages introduced the I-structure [3] and M-structure [6]. I-structures are shared state that is restricted to a single write operation but an unlimited number of reads. A second write to an I-structure is a dynamic error, resulting in an invalid store. Reading an I-structure before its data has been written causes the reading thread to block until the data is available. M-structures provide shared state with paired readers and writers. The read operation blocks until data is available, clearing and returning the value once it is available. The write operation blocks if the M-structure is full, until a read has been performed and the cell cleared. Our system does not provide direct analogs to either of these communication mechanism.

The Parcel Scheme compiler and runtime also provide automatic parallelization [38]. That work performs static analysis of Scheme programs to determine lifetimes of mutable variables. The compiler then determines the available parallelization that respects the dependencies between the pieces of code. While that work also performs a flow-based effect analysis, it predates the control-flow analysis techniques that we use, limiting its ability to handle higher-order functions as generally as we can in identifying where `atomic` annotations are not needed because the computation is guaranteed to be pure.

6.3.6 Explicit parallelism

One of the most popular implementations of explicit parallelism is the use of POSIX threads (pthreads) in C [16]. This language and library combination offers great performance, but requires the programmer to handle all issues related to the number of threads, granularity of work, locking, synchronization, and data consistency. This proposal aims to capture a significant portion of the performance achievable with this combination while reducing the programmer burden.

Glasgow Parallel Haskell (GPH) supports explicit **`par`** and **`pseq`** constructs for parallel execution of work. Obtaining good performance from these keywords requires careful programming and

machine-specific tuning using advanced profiling tools[44]. Unlike the PML implicitly-threaded language features, this approach requires that the runtime create a lightweight parallel thread (spark) for any work explicitly occurring within the **`par`** or **`pseq`** constructs, though depending on the scheduling that work may be executed on the original processor. Since these constructs require pure computations, any use of impure language features must occur within the **`IO`** monad, explicitly defining a sequential order of operation.

GPH also has explicit thread creation routines. These routines operate in the context of the **`IO`** monad, so all synchronization between the threads must be explicitly performed by the programmer through shared state. For example, if two threads perform **`putStrLn`** operations, the order in which those strings appear is not defined. But, if a data structure — such as an MVar, which is based on the M-structures from Section 6.3.5 — is used by the programmer to sequence the operations, then the operations will be performed in sequence. Shared data structures at risk of concurrent access are implemented with explicit atomic operations or locking, where necessary to ensure consistency.

CHAPTER 7

CONCLUSION

The addition of features that enable communication between threads in a parallel functional program without halting parallel work to share intermediate results is both important and challenging. There are algorithms that are significantly faster when they are permitted to share intermediate results if the synchronization penalties for doing so are minimized. These algorithms are currently implemented in other programming languages using low-level locking operations, atomic memory instructions, or software transactional memory libraries, all of which require both specific knowledge of the target hardware and application workload to achieve high performance.

Our approach extends the Parallel ML (PML) language with two features, memoization and mutable state, that provide this sharing without requiring per-machine or per-workload reasoning from the programmer. Memoization exploits the pure nature of functions to achieve synchronization-free sharing between threads. Mutable state requires synchronization, but when paired with our analysis and implementation techniques, those synchronization costs are reduced. Our operational semantics precisely describes the allowable sharing and execution model of programs that use memoization and mutable state.

7.1 Future Work

This work shows promising results, but also suggests many potential further avenues of research. The largest current limitation is due to the state of our compiler — the team is in the process of rewriting the entire frontend, so use of both of these features currently requires direct calls to the underlying library functions rather than the high-level syntax proposed in this dissertation. Additionally, while this work was integrated with many of the implicitly-threaded parallel constructs of PML, it has not been integrated with parallel case or explicitly-threaded parallelism.

7.1.1 Memoization

While this replacement policy addresses the issue of providing an implementation that supports a fixed table size, it does not address the issue of holding on to entries that are no longer needed. The garbage collector for Manticore supports custom, datatype-specific functions during collection [5]. This support could be used to automatically prune any elements from the table whose time had exceeded a threshold. Implemented during global collection, this change would simply copy an empty entry (`NONE`) value and abandon the original value.

Past some point, the work required to look up and store memoized data may outweigh the cost of simply recomputing the value. Previously in the Manticore project, we have had success performing static analysis of well-structured recursive parallel programs to determine when it might be more efficient to perform work sequentially instead of in parallel [4]. That analysis could be extended to recursive functions that perform memoization to determine when the expected work size is smaller than the overhead of using memoization. In those cases, memoization could be removed.

Currently, we only memoize functions whose parameter type is `int`. This design choice is extremely limiting, since it relies on both a user-provided and implemented hash function and the ability to avoid collisions. We would like to extend the signature in future work to something more general, such as the following:

```
signature MEMO_TABLE =
 sig
    type ('dom,'rng) table

    val insert : ('dom, 'rng) table * 'dom * int * 'rng -> unit
    val find : ('dom, 'rng) table * 'dom * int -> 'rng option
end
```

This implementation would require users to provide both a hashing function and equality for the

argument types (to handle the case where distinct arguments have the same hash value), but would dramatically increase the number of programs that could take advantage of this feature. The implementation of the memoization table would also be required to store the original argument type for the equality check.

Finally, currently our implementation of memoization does not guarantee that a function will be evaluated exactly once at any argument. At the cost of additional synchronization overhead and a table implementation that never discards values, we could provide an alternative implementation that does provide that guarantee.

7.1.2 Mutable state

Currently, we support only transactional variables. But, there are other interesting forms of mutable state. For example, some state may only ever be filled with a particular value (such as a per-index answer in an array). These *idempotent* values could be implemented with a mechanism similar to the I-structure of Id [3]. Additionally, some state may have *monotonic* behavior. That is, many threads might access it, but they would perform operations that strictly overwrite it with either increasing or decreasing values, such as performing a parallel `max` search over a large set of values using a single global update cell instead of a reduction-style implementation. Both of these features could be implemented using techniques that are more expensive than memoization but less than the transactional logging and rollback currently associated with mutable state.

Our programming model also prevents the programmer from writing their own high-performance implementations of features that do not require atomic behavior. For example, the memoization feature cannot be implemented using mutable state in a way that avoids locking, temporary additional storage for results, and synchronization, as the implementation of memoization does in this work.

Finally, our design of transactions memory is quite simple compared to any serious transactional memory implementation. There has been over a decade of work in optimizing transactional

memory implementations that would likely be useful in our system, particularly when we add support for nested transactions, which we have only sketched the design of but which are not currently implemented in the system.

7.1.3 Semantics

While we have defined an operational semantics that defines the dynamic behavior of programs written using these new features, we have defined neither a type system nor proved the semantics sound. Proving it sound would be a useful result because of the exclusion properties that surround both memoization and mutable state — disallowing mutable state within memoization and interference between two concurrent expressions that modify the heap are both dynamically ensured, but it would be more useful to the programmer to provide a static error along with a guarantee that programs that typecheck never get stuck.

We would like to show that our language is linearizable — that is, that each function call appears to take effect instantaneously [42]. While our system wraps all uses of mutable state into atomic regions, providing a guarantee that the effects of each expression evaluation will appear instantaneous with respect to all concurrent expressions, proving linearizability requires more an extension of our semantics that allows us to precisely characterize the potential execution histories.

REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *Conference Record of the 30th Annual ACM Symposium on Principles of Programming Languages (POPL '03)*, pages 14–25, New York, NY, 2003. ACM.

[2] A. W. Appel. Simple generational garbage collection and fast allocation. *Software – Practice and Experience*, 19(2):171–183, 1989.

[3] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, Oct. 1989.

[4] S. Auhagen. Hybrid Chunking. Master's thesis, University of Chicago, Apr. 2012. Available from `http://www.cs.uchicago.edu/phd/ms_completed`.

[5] S. Auhagen, L. Bergstrom, M. Fluet, and J. Reppy. Garbage Collection for Multicore NUMA Machines. In *MSPC 2011: Memory Systems Performance and Correctness*, New York, NY, June 2011. ACM.

[6] P. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture (FPCA '91)*, volume 523 of *Lecture Notes in Computer Science*, pages 538–568, New York, NY, Aug. 1991. Springer-Verlag.

[7] R. Barton, D. Adkins, H. Prokop, M. Frigo, C. Joerg, M. Renard, D. Dailey, and C. Leiserson. Cilk Pousse, 1998. Available from `http://people.csail.mit.edu/pousse/`.

[8] L. Bergstrom. Arity raising and control-flow analysis in Manticore. Master's thesis, University of Chicago, Nov. 2009. Available from `http://manticore.cs.uchicago.edu`.

[9] L. Bergstrom. Measuring NUMA effects with the STREAM benchmark. Technical Report TR-2012-04, Department of Computer Science, University of Chicago, May 2012.

[10] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy tree splitting. *Journal of Functional Programming*, 22(4-5):382–438, Sept. 2012.

[11] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, Mar. 1996.

[12] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '95)*, pages 207–216, New York, NY, July 1995. ACM.

[13] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot Topics in Parallelism*, pages 4–4, Berkeley, CA, 2009. USENIX Association.

[14] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, pages 68–78, New York, NY, 2008. ACM.

[15] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture (FPCA '81)*, pages 187–194, New York, NY, Oct. 1981. ACM.

[16] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, MA, 1997.

[17] M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and W. Pfannenstiel. Nepal – nested data parallelism in Haskell. In *Proceedings of the 7th International Euro-Par Conference on Parallel Computing*, volume 2150 of *Lecture Notes in Computer Science*, pages 524–534, New York, NY, Aug. 2001. Springer-Verlag.

[18] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, pages 10–18, New York, NY, Jan. 2007. ACM.

[19] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, New York, NY, 2001.

[20] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94)*, pages 70–83, New York, NY, Jan. 1994. ACM.

[21] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL '93)*, pages 113–123, New York, NY, Jan. 1993. ACM.

[22] K. Emoto, S. Fischer, and Z. Hu. Generate, test, and aggregate. In H. Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 254–273. Springer-Verlag, New York, NY, 2012.

[23] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML*, pages 15–24, New York, NY, Oct. 2007. ACM.

[24] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5–6):537–576, 2011.

[25] E. R. Gansner and J. H. Reppy, editors. *The Standard ML Basis Library*. Cambridge University Press, Cambridge, England, 2004.

[26] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP languages. *ACM Transactions on Programming Languages and Systems*, 22(2):296–339, 2000.

[27] L. George, F. Guillame, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *Fifth International Conference on Compiler Construction*, pages 83–97, Apr. 1994.

[28] GHC. The Glasgow Haskell Compiler. Available from `http://www.haskell.org/ghc`.

[29] D. K. Gifford, P. Jouvelot, and M. A. Sheldon. The FX-87 reference manual. Technical Report TR-407, Massachusetts Institute of Technology, Sept. 1987.

[30] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Report on the FX programming language. Technical Report TR-531, Massachusetts Institute of Technology, Feb. 1992.

[31] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Conference record of the 1986 ACM Conference on Lisp and Functional Programming*, pages 22–38, New York, NY, 1986. ACM.

[32] W. G. Griswold and G. M. Townsend. The design and implementation of dynamic hashing for sets and tables in Icon. *Software – Practice and Experience*, 23(4):351–367, Apr. 1993.

[33] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 315–324, New York, NY, 2007. ACM.

[34] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel execution of Prolog programs: A survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[35] R. H. Halstead Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, New York, NY, Aug. 1984. ACM.

[36] T. Harris, S. Marlow, and S. P. Jones. Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 Haskell Workshop*, pages 49–61, New York, NY, 2005. ACM.

[37] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '05)*, pages 48–60, New York, NY, June 2005. ACM.

[38] W. L. Harrison. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2:179–396, 1989.

[39] S. Herhut, S.-B. Scholz, and C. Grelck. Controlling chaos: On safe side-effects in data-parallel operations. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, pages 59–67, New York, NY, Jan. 2009. ACM.

[40] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '08)*, pages 207–216, New York, NY, Feb. 2008. ACM.

[41] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, New York, NY, 2008.

[42] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[43] J. Hughes. Lazy memo-functions. In *Functional Programming Languages and Computer Architecture (FPCA '85)*, pages 129–146, New York, NY, 1985. Springer-Verlag.

[44] D. Jones, Jr., S. Marlow, and S. Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2009 Haskell Workshop*, pages 81–92, New York, NY, 2009. ACM.

[45] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Conference Record of the 37th Annual ACM Symposium on Principles of Programming Languages (POPL '10)*, pages 19–30, New York, NY, 2010. ACM.

[46] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 211–222, New York, NY, June 2007. ACM.

[47] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Computer*, C-28(9):690 –691, Sep 1979.

[48] P.-A. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, Apr. 1988.

[49] V. Y. Lum, P. S. T. Yuen, and M. Dodd. Key-to-address transform techniques: A fundamental performance study on large existing formatted files. *Communications of the ACM*, 14(4):228–239, Apr. 1971.

[50] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. Warren, A. Calderwood, P. Szeredi, S. Haridi, and Others. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2):243–271, Feb. 1990.

[51] L. Mandel and L. Maranget. *The JoCaml Language Release 3.11 Documentation and User's Manual*, Dec. 2008. Available from `http://jocaml.inria.fr/manual/index.html`.

[52] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Conference Record of the 32nd Annual ACM Symposium on Principles of Programming Languages (POPL '05)*, pages 378–391, New York, NY, 2005. ACM.

[53] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 65–77, New York, NY, August–September 2009. ACM.

[54] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[55] D. Michie. "Memo" functions and machine learning. *Nature*, 218:19–22, Apr. 1968.

[56] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *Conference Record of the 35th Annual ACM Symposium on Principles of Programming Languages (POPL '08)*, pages 51–62, New York, NY, 2008. ACM.

[57] R. S. Nikhil. *ID Language Reference Manual*. Laboratory for Computer Science, MIT, Cambridge, MA, July 1991.

[58] R. S. Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.

[59] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, Apr. 2007.

[60] C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *Proceedings of the 5th Conference on Computing Frontiers (CF '08)*, pages 67–78, New York, NY, May 2008. ACM.

[61] W. Pugh. An improved replacement strategy for function caching. In *Conference record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 269–276, New York, NY, July 1988. ACM.

[62] W. W. Pugh. *Incremental Computation and the Incremental Evaluation of Functional Programs*. PhD thesis, Cornell University, Ithaca, NY, USA, 1988.

[63] M. Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. PhD thesis, University of Chicago, Aug. 2010. Available from `http://manticore.cs.uchicago.edu`.

[64] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

[65] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, May 2006.

[66] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, 1995. ACM.

[67] A. Shaw. Data parallelism in Manticore. Master's thesis, University of Chicago, July 2007. Available from `http://manticore.cs.uchicago.edu`.

[68] D. Zhang and P.-A. Larson. LHlf: Lock-free linear hashing (poster paper). In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '12)*, pages 307–308, New York, NY, Feb. 2012. ACM.

[69] L. Ziarek, K. Sivaramakrishnan, and S. Jagannathan. Partial memoization of concurrency and communication. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 161–172, New York, NY, 2009. ACM.