Fault Tolerance using Whole-Process Migration and Speculative Execution

Thesis by

Justin David Smith

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

2003
(Submitted May 30, 2003)

© 2003
Justin David Smith
All Rights Reserved

Acknowledgements

This thesis would not have been possible without contributions and support from a number of individuals. It has been a great pleasure to interact with my colleagues in the course of my research. I am also truly amazed at the number of people who have supported me in achieving my dreams throughout my life, and to those people I offer my most sincere gratitude. Below I will acknowledge some of the major contributors and supporters of this thesis.

I am very grateful to my advisor, Professor Jason Hickey, who introduced me to the field of compilers and formal methods, and has offered constant inspiration and guidance throughout my graduate-level research. I also thank everyone in the PRL/Mojave group for their insight in the development of this work; in particular, Brian Aydemir, Adam Granicz, Nathaniel Gray, and Cristian Ţăpuş contributed to the formalization of the Mojave compiler that is described in this thesis.

My family and friends have always encouraged me to follow the road less traveled, no matter how intimidating the obstacles might seem. They have never faltered in their support of my aspirations, no matter how high, and they have always been there to offer guidance when I needed it. I offer my deepest thanks to my parents, Janet and David Smith, who have always been there for me, and who taught me the importance of seeking my dreams; their support has been the most valuable contribution to my life. I also offer deep thanks to everyone in my extended family, and to all of my friends — their love and support have given me the strength I needed to make it this far.

The teachers I've had throughout my life are responsible for my rich academic background that has allowed me to pursue studies at such a prestigious institute. I thank all of my teachers, and in particular I would like to thank Mrs. Dolores Veselka, who encouraged my early studies in mathematics, and Mr. Henry Veselka, who taught me my first programming languages, inspiring me to study computer science. The Veselkas have continued to follow my academic career and support my work, and for that I am very grateful.

This work was supported by Air Force grant F33615-98-C-3613 and the Department of Defense, Lawrence Livermore National Laboratory contract W-7405-ENG-48, subcontract B523297.

Abstract

This thesis examines programming language concepts that facilitate fault-tolerant distributed programming. New language primitives are introduced for whole-process migration, which allows an active process to be transferred from one machine to another, and speculative execution, which enables optimistic computing based on an unverified assumption. These primitives are developed in the context of the Mojave Compiler Collection, a multi-language multi-architecture compiler with ties to the MetaPRL theorem prover.

The new primitives are first discussed from a theoretical perspective. The primitives are implemented as part of a functional intermediate language in the Mojave compiler, which has a formal operational semantics and complete typing rules. The operational semantics and typing rules are extended to accommodate whole-process migration and speculative execution, and the implications these primitives have for program safety are discussed.

The primitives are implemented as part of the Mojave compiler. The runtime safety checks that are required to ensure these primitives are safe are presented, along with runtime invariants used to justify the safety of the system. The primitives are also integrated with a novel compacting, generational garbage collector whose algorithm is presented.

Contents

List of Figures

Chapter 1

Introduction

We propose a new model for addressing fault tolerance in distributed computation, by introducing two new language primitives for whole-process migration and speculative execution. Both primitives are implemented as extensions to the Mojave Compiler Collection (MCC), a compiler under development which compiles multiple source languages into a functional intermediate representation. This chapter will formulate problems in distributed computing which these primitives are designed to address, and will give a general introduction to migration and speculation. The remainder of the thesis will address the theoretical properties and implementation of these primitives in the Mojave compiler.

1.1 The distributed application problem

Consider an application that is distributed across many machines, or *nodes*, that are connected by a network backbone. Each node runs one or more processes that perform a set of *tasks*. In this distributed system, there are a number of potential failures that can disrupt the application. Hardware failures with individual machines or with the underlying network can disrupt coordination of the processes in the application, and software failures can cause the processes to enter an inconsistent state. An unanticipated failure at any of these points causes the entire application to fail. If the application spans tens of thousands of nodes, or if it is run over an unreliable network such as the Internet, then the odds of experiencing a failure somewhere in the system are significant.

Traditionally, distributed application developers implement a substantial amount of error handling code to respond to each of these faults. Since error detection and recovery code must be written for every operation where the process communicates with another node, this can lead to "spaghetti code," where the error handling code is tightly intertwined with code for the computation being performed. Programmers have difficulty analyzing this code for correctness, since it is difficult to isolate the computation from the error handling code.

This situation can be improved by providing reliable communication protocols. For example, a total order communication protocol guarantees that messages will be delivered in a well-defined order to every node in the distributed system that is alive. When a node fails, the total-order protocol ensures that surviving nodes continue to receive messages as expected, however it does not provide a mechanism for recovering the computation that was being performed on the failed node. Therefore the programmer must still write a substantial amount of code to recover the lost computation or to reassign the failed node's tasks to surviving nodes. This is a significant improvement over the naive implementation since much of the error recovery code is isolated in the communication primitives, but it is only a partial solution as a fault recovery model.

In the error recovery paths for distributed systems, the programmer is often attempting to revert the system's state to a consistent state immediately before the failure. A large portion of the error recovery code is devoted to identifying the failed node, determining what tasks were running on the failed node, and reassigning those tasks to a surviving node. The programmer may have to undo many computations that were already performed on the surviving nodes, because the computations relied on results from the failed node that were not saved to a persistent store and must be recomputed. All of this recovery code emphasizes the restoration of the distributed system to a consistent state that is prior to the failure.

Tasks evaluated in a distributed system must be capable of moving from one node to another; otherwise a failure on the node a task is running on will block all tasks that are dependent on it, which potentially covers the entire distributed system. Since tasks may need to be reassigned to surviving nodes, either each node in the system must contain code to evaluate each possible task, or else we must have a mechanism for migrating executable code into an arbitrary node. In grid computations, where every node runs the same algorithm but on a different data set, task migration simply involves copying the data set to a new node; however, in a heterogeneous distributed system, where each task is running a different algorithm, a mechanism to migrate executable code from one node to another will greatly simplify the programmer's work.

1.1.1 Checkpoints in distributed applications

One approach that addresses some of the issues of distributed fault-tolerance is to provide languages that have built-in mechanisms for taking a snapshot, or *checkpoint*, of a distributed system, and record these checkpoints on a persistent storage device. Each checkpoint records the complete state of every process in the system. The checkpoint is taken in such a way that all process states are logically consistent — the collection of process states represents a reachable state in the distributed system. A simple approach to record a self-consistent checkpoint is to suspend every process in the system and begin recording the checkpoint once every process has stopped. Because every process is recorded by the checkpoint, the entire distributed system can be restored to the point it is at when the checkpoint is recorded.

```
function Node-Computation:
    while true do
        — Mark a synchronization point for error recovery.
        checkpoint()
        — Now, compute. This computation is free of error-recovery code. If an error is
        — detected, an exception is raised that automatically restores the checkpoint.
        compute for one iteration
    end while
```

Figure 1.1: Example of grid computation using checkpoints

Checkpoints can be provided by a service library or using primitives in a language specialized for distributed computation. When checkpoints are embedded in the language, the compiler is able to optimize checkpoint performance by choosing internal representations of the data that are easy to save and restore. The compiler is well-suited to generating checkpoint code, since it has a record of the internal representation of each process's state.

By encapsulating all of a process's state into a single object, it becomes possible to migrate entire tasks from one machine to another, allowing the system to automatically revive tasks lost because of a node failure onto a surviving node. If architecture-independent data representations are used, then migration is feasible even for heterogeneous distributed networks. With checkpoints as a language primitive, much of the work of transmitting the process state can be automated by the compiler.

Specialized languages can also include automatic detection of failed nodes, so the programmer simply marks which parts of the distributed system reflect a consistent state. On a failure, built-in language primitives can rollback all process and I/O state using the most recent checkpoint of the system. This approach puts the burden of fault detection and recovery on the compiler itself, and with proper language design, error handling code can be clearly separated from the algorithms.

1.1.2 Grid computation

Grid computation is a class of distributed applications where a data set is divided and distributed among nodes in the system, and each node performs the same computation on its data set. Typically, this is used in scientific computing for physical simulations, where an iterative computation is run in parallel on a large n-dimensional data space, with each node responsible for a particular region of the space. These computations are often performed on high-performance distributed clusters containing tens of thousands of nodes connected by a dedicated, high-speed network backbone. These clusters have a low mean time between failure, on the order of one day, and as a result any long-term computation must be prepared for several failures during its lifetime.

In this realm, it is relatively easy to take a checkpoint of the distributed system because the synchronization points are easy to identify. Currently, programmers mix the algorithm and error handling code, but with checkpoints, the overhead in the source language for such a computation can be as minimal as is illustrated in Figure 1.1, where the error handling is encapsulated in a single language primitive, and is clearly separated from the computation itself. On a failure, the runtime which supports the language can automatically revert surviving processes to the last successful checkpoint and resurrect tasks that were lost on surviving machines. All of the fault recovery is transparent to the programmer.

1.2 Capturing snapshots of a process state

We develop language primitives for checkpointing as part of the Mojave compiler, which is described in more detail in Section 1.3. The Mojave compiler utilizes two tools to take snapshots of the process state. The first is *whole-process migration*, where an entire process state is migrated from one machine to another, or recorded to a persistent storage device for later execution. The second mechanism is an adaptation of transactions, which we refer to as *speculative execution*. Speculative execution allows a process state to be rolled back, but unlike whole-process migration, speculative execution is not persistent.

1.2.1 Whole-process migration

Whole-process migration can be used to migrate an active process from one machine to another, and to record a process's state to a persistent storage device. Migration provides persistence of computations, which is necessary to accommodate machine failures.

Whole-process migration has been widely studied [6, 19]. The JoCaml system [4] provides process mobility for OCaml programs based on the join calculus [7]. The Mojave compiler's approach to process migration is heavily influenced by Cardelli's work on the Ambient Calculus [2, 3], however the Mojave compiler only supports whole-process migration at this time, and does not yet support the fine-grained mobility described in the ambient calculus.

This thesis emphasizes migration's role in the simplification of fault-tolerant computing. Each node in a distributed computation can use migration to take periodic persistent checkpoints, which are used to recover the computation on another machine if the node fails. Migration can also be used to improve the performance and simplify development of distributed systems.

1.2.1.1 Load balancing

Load-balancing is an important issue in a distributed system, especially if the nodes or tasks are heterogeneous. A system monitor can detect nodes in the system that are overloaded relative to other nodes, and migrate some of the processes to balance the load. Processes do not need to observe the result of a migration, since the distributed system will provide a consistent environment across

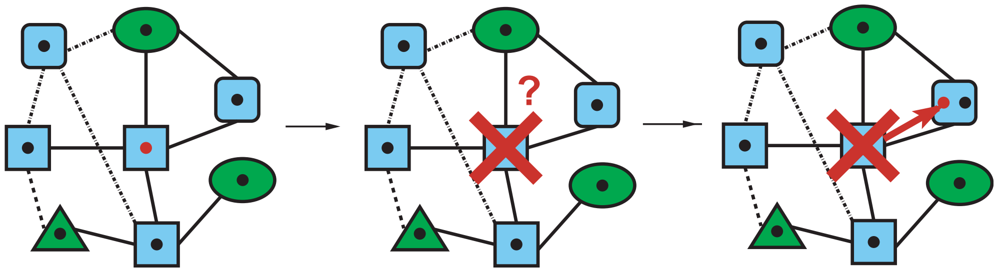


Figure 1.2: Fault tolerance using migration in a heterogeneous network

all machines. Therefore load balancing can be performed without any special support from the processes.

1.2.1.2 Active messages

Migration can be used to pass *active messages*, or messages that include executable code and data. To pass a message from one process to another, the process may start a new thread, and migrate the thread (including all necessary state) to the machine hosting the other process.

This can be particularly useful for database queries. When the database is hosted on a remote machine, it is highly inefficient to run a query on the local machine that transfers the entire database over the network to extract the few records which the query is interested in. Mainstream database systems work around this by supporting a query language used to describe the query. The local machine sends the text of the query, which is then run on the server. However, these languages are highly specialized and are not sufficiently expressive for many types of inquiries. Process migration allows arbitrary programs to migrate to the remote machine, allowing far greater expressiveness using a more general mechanism.

Similar to active messages, migration can be used to create active files, or files that contain data but also code to manipulate the data, similar to existing file formats which include a macro or scripting language. A simple example would be a multimedia file that includes the program required to play the media.

1.2.1.3 Fault tolerance in distributed programming

Migration provides a mechanism for persistent storage of the process state and the ability to resurrect the process on any machine. Therefore, when a node in a distributed network fails, migration can be used to revive all tasks that were running on the system prior to the failure. With a machine-independent representation of the code and data for each process, tasks can be resurrected on an arbitrary node in a heterogeneous distributed network, as illustrated in Figure 1.2.

Since the checkpoints may be recorded infrequently, the revived processes will be at an earlier point in the computation than the surviving processes. They can replay the original computation up to the point of failure, given the same inputs as the original process.

1.2.2 Speculative execution

In a distributed computation, the processes involved must synchronize periodically. Ideally, a process would speculatively continue computing across a synchronization point, even if it has not received confirmation that all other processes have reached the same synchronization point. In this case, a process can enter a new speculation as it passes the synchronization point and continue computing. If a process in the system fails, the remaining processes can agree on the last known-good synchronization point using a consensus algorithm, and then roll back to that synchronization point to continue the computation.

Speculations share many traits with traditional distributed transactions, one of the earliest and simplest abstractions for reliable concurrent programming [9]. Transactions provide source-level fault isolation: from a process's point of view, a failure cannot occur during a transaction; if a failure occurs, it must occur before or after. While transactional models are ubiquitous in the database community, they have not been frequently applied to traditional programming languages. As part of the Venari project, Haines et.al. [10] implement a transaction mechanism as part of Standard ML, utilizing a mutation log produced by a generational garbage collector to implement undoability.

Speculations share the same operations as transactions, but speculations provide weaker properties. For one, transactions are atomic; a process that is not involved in the transaction is not allowed to view the intermediate state of the processes that are participating in the transaction. This external process must either block until the transaction is completed, or else it must access an older copy of the state from before the transaction was entered. Speculations do not need to be atomic; if another process depends on a value generated during a speculation, the other process may simply join the existing speculation, so the processes will be rolled back in unison in the event of a failure.

Speculations may also be committed in the opposite order of transactions. For fault-tolerance, the process may speculatively execute past a synchronization point, and commit older speculations as it receives confirmations that the other nodes have reached the synchronization point. Since the confirmations typically arrive in temporal order, the process will usually commit the *oldest* speculation in the system, which is opposite the behavior a traditional distributed transaction would provide. In a speculation, if multiple speculations are active when a particular speculation is committed, the committed speculation is simply folded into the next-oldest speculation. This gives speculations more flexibility than traditional transactions provide.

Conventional code	Speculative code
1: **function** TRANSFER($file_1, file_2, k$)	1: **function** TRANSFER($file_1, file_2, k$)
2: $i :=$ **read**($file_1$)	2: **speculate**
3: **if read** failed **then return** failure	3: $i :=$ **read**($file_1$)
4: $j :=$ **read**($file_2$)	4: $j :=$ **read**($file_2$)
5: **if read** failed **then return** failure	5: **write**($file_1, i - k$)
6: **if write**($file_1, i - k$) failed **then return** failure	6: **write**($file_2, j + k$)
7: **if write**($file_2, j + k$) failed **then**	7: **return** success
8: — *Undo* $file_1$	8: **catch**
9: **write**($file_1, i$)	9: **return** failure
10: — *Unrecoverable if this write fails*	
11: **return** failure	
12: **return** success	

Figure 1.3: File transfer operation using lightweight transactions

1.2.2.1 Fault isolation and transaction-like behavior

Like transactions, speculations allow for fault isolation. Frequently, error-recovery code in a program exists to revert the state to a known-consistent state. As a simple example, consider a banking program that wishes to transfer a sum of money k from one account to another. If a failure occurs during the transfer, the program must take steps to ensure that the amount k remains credited to the original account. Code to perform this operation is illustrated in Figure 1.3.

In the traditional case, the programmer must manually revert the state of all modified files if an I/O failure occurs. Note that the error-recovery code itself requires some form of error-recovery; it is entirely possible that line 9 will fail, at which point it is not clear how the program should proceed. By contrast, speculations allow the programmer to automatically roll back the entire process state to the state on entry to this operation, without requiring the programmer to explicitly recover the state.

Speculations also offer a unique form of debugging; a process that takes periodic speculations can be debugged by rolling back to a prior speculation, and re-evaluating the code. This allows debuggers to support commands for stepping back one or more instructions, similar to support they already provide for stepping forward in the program. The ability to move in both forwards and backwards in the execution of a program can make it easier to isolate buggy code in a program.

1.2.2.2 Fault tolerance in distributed programming

Speculative execution enables a "monitor and recover" programming model [15], where processes monitor in the background for failures in the distributed system, and roll back their computation on discovery of a failure. The surviving processes will need to roll back their state in order to return to a consistent state for the overall distributed system.

It is possible to implement rollback of a surviving process using whole-process migration, since all processes are periodically written to persistent storage. However, this would require recovery of the full process state from the relatively slow persistent storage device. By comparison, speculations can utilize the current state of the process to roll back more efficiently, by keeping track only of the changes to the state since the last speculation. This mutation log is likely to be much smaller than the full process state, and therefore more efficient to utilize.

1.2.2.3 Generalization of speculative evaluation

In general, a speculative execution is based on an assumption A that may or may not be true, but is not immediately evaluated. The speculative execution is committed if A is verified to be true, and aborted if the process later determines A is false. For fault-tolerance, A represents the assumption that no machine in the distributed network will fail, and for fault isolation, A represents the assumption that no exception will be raised. In these cases, A cannot be assessed in advance, since A is dependent on the code that will be evaluated during the speculation.

There are other situations where it may be possible to evaluate A in advance, but for performance reasons it is more advantageous to defer the evaluation of A until part of the speculative code has been executed. In these situations, A itself may be dependent on information from other computers, and may be a slow computation due to network latencies. If A is likely to be true, and the cost of a speculation is low, then it is in the process's best interests to assume A and rollback if the assumption proves to be false later.

This approach is utilized in shared memory systems by general message predictors and pattern-based predictors to learn and predict the memory activity in distributed shared memory systems for performance improvement. Two such systems are Cosmos [14], which predicts future coherence operations and performs them speculatively, and the Memory Sharing Predictor [13], which detects patterns in memory request messages and sends read-only pages of memory to the predicted requesters.

1.3 The Mojave Compiler Collection (MCC)

We implement whole-process migration and speculative execution as part of the Mojave Compiler Collection (MCC)[1]. MCC is a multi-language compiler that compiles C, Pascal, ML, Java, and other languages to a common, functional intermediate language (FIR). The FIR is a concise semi-functional representation of the code, where variables in the FIR are immutable, but heap values can be modified. Looping constructs are expressed using recursion, and all internal function calls are expressed as tail-calls. The FIR also contains primitive features to support its source-level languages in a type-safe manner. MCC can be integrated with MetaPRL [12], a theorem prover capable of reasoning about programs written in the FIR language. The MCC architecture is illustrated in

[1] The latest version of MCC is available for download from `http://mojave.caltech.edu./`.

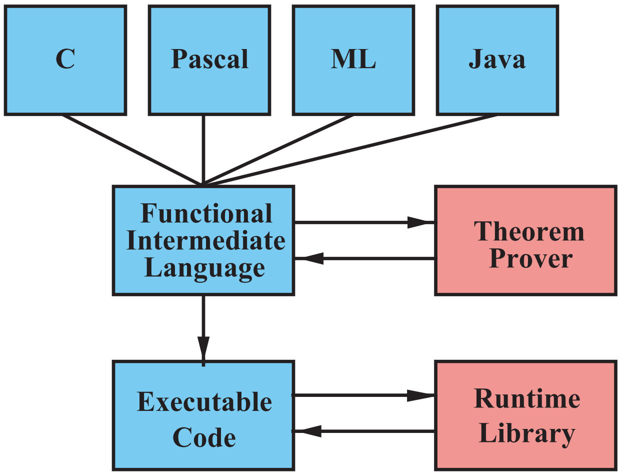


Figure 1.4: Design of the MCC compiler

Figure 1.4.

MCC utilizes formal reasoning of programs and runtime safety checks to ensure that programs compiled using MCC are *safe* — that is, they will not attempt to access illegal areas of memory, or use values with inappropriate types. MCC's safety properties allow mutually untrusting systems to exchange and evaluate code, and reduce the need for explicit security boundaries in distributed systems. They also allow programmers to develop modules for execution directly in kernel space, with the assurance that no MCC-compiled module will errantly tamper with other kernel modules.

MCC is able to impose an architecture-independent representation of data and can migrate code using the architecture-independent FIR representation. This makes it an ideal platform for developing whole-process migration primitives in an efficient manner. Also, MCC's heap design is easy to extend to support speculative execution models.

MCC provides an active test bed for research in several areas of distributed systems research. Several projects are conducted by members of the Mojave group to include a distributed filesystem [8] and distributed shared memory [5] with MCC, taking advantage of the features offered by the compiler. These projects extend the concepts of migration and speculation to the design of persistent storage and distributed communication technologies.

1.4 Organization of this thesis

Chapter 2 begins with a formal introduction of the language and primitives used to implement process migration and speculative execution. It covers the syntax and operational semantics of the language, and describes basic properties of process migration and speculation. Chapter 3 demonstrates that programs using process migration and speculations will be safe, even after migration from one system to another. Chapter 4 focuses on the implementation of these primitives, including data structures, invariants, and algorithms for implementing the primitives. Chapter 5 discusses integration of these primitives with a compacting garbage collector. Chapter 6 presents the conclusions of this work and future directions.

Chapter 2

Semantics of Process Migration and Speculation

Process migration and speculation are implemented as extensions to MCC. The semantics of migration and speculation is defined as part of the FIR, the intermediate language used internally by MCC. The complete FIR language, including the FIR operational semantics, is described in detail in a technical report by Hickey, Smith, et al. [11]. The FIR presented here is a subset of the full FIR language; only the properties of the FIR that are relevant to this thesis are discussed here.

The formal discussion of migration and speculation will be with respect to the FIR. This chapter introduces basic FIR syntax and provides the operational semantics for migration and speculation. Safety properties for migration and speculation are given in Chapter 3, and the runtime support required to implement these primitives is discussed beginning in Chapter 4.

2.1 The FIR syntax

The following conventions are used in the descriptions below. In general, the meta-variables i, j, k, and l to refer to arbitrary integers, and the meta-variables m and n to refer to arbitrary nonnegative integers. The meta-variable v refers to program variables. The meta-variables t and u refer to program types, while the Greek letters α, β, γ and the meta-variable tv refer to type variables.

In most cases, the meta-variable m enumerates type parameters $\alpha_1, \ldots, \alpha_m$, and the meta-variable n enumerates actual parameters $v_1, \ldots, v_n$. The notation $\lceil v_i \rceil_1^n$ is a shorthand denoting the vector $v_1, \ldots, v_n$, where the index variable is implicitly i. An alternate notation $\lceil v_j \rceil_{j=1}^n$ explicitly uses a different index variable j.

The basic FIR base terms are shown in Figure 2.1. MCC supports several forms of numbers, including integers of various signedness and precision, and floating-point values of various precisions. Sets of integers are used in integer pattern matching expressions. The sets are represented by lists of closed intervals $[i_1, i_2]$.

		Entity	Description
		$v_1, v_2, \ldots$	Variable names
		$tv_1, tv_2, \ldots$	
		$\alpha, \beta, \ldots$	Type variables
i	::=	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots$	Integer constants
set	::=	$[i_1^1, i_2^1], \ldots, [i_1^n, i_2^n]$	Integer interval set
pre	::=	$8 \mid 16 \mid 32 \mid 64$	Integer precisions
$sign$	::=	**signed** | **unsigned**	Integer signedness
r	::=	$(i, pre, sign)$	Raw integer constants

Figure 2.1: FIR base terms

2.1.1 FIR type system

The FIR has two classes of types, the basic types, shown in Figure 2.2, and the type definitions, which are parameterized types of the form $\Lambda\alpha_1, \ldots, \alpha_m.t$.

The type $\mathbb{Z}_{box}$ refers to tagged, signed integers. Native integers with bit precision *pre* and signedness *sign* are represented using $\mathbb{Z}_{pre}^{sign}$, and must be boxed when stored into memory. Floating-point values are also supported by the implementation, but are not described here.

Memory values are represented by several types. The tuple type $\langle t_1, \ldots, t_n \rangle$ represents a tuple $\langle v_1, \ldots, v_n \rangle$, where each value v_i has type t_i. The array type t **array** resembles a tuple, but the elements of an array all have the same type, and an array has arbitrary nonnegative dimension. Both tuples and arrays are seen as safe types — the compiler guarantees that values in these memory areas have the appropriate types. MCC also supports tagged tuples, or unions, but they are omitted here for simplicity.

The unsafe type **data** represents arbitrary data. Values of type **data** are normally used to represent data aggregates for imperative programming languages, such as C, that allow the assignment of values to the data area without regard for the data type. Data areas with the **data** type have no explicit substructure.

The function type $(t_1, \ldots, t_n) \to t$ includes the functions that return a value of type t, given arguments of types $t_1, \ldots, t_n$.

For polymorphic types, MCC provides type application $tv[t_1, \ldots, t_m]$, which applies arguments to a type definition. If the definition is a parameterized type $\Lambda\alpha_1, \ldots, \alpha_m.t$, the type $tv[t_1, \ldots, t_m]$ is defined as the type $t[t_1/\alpha_1, \ldots, t_m/\alpha_m]$. For example, in a context containing the definition $\gamma = \Lambda\alpha, \beta.\langle \alpha, \beta \rangle$, the type $\gamma[\mathbb{Z}_{box}, \mathbb{Z}_{32}^{\mathbf{signed}} \to \mathbb{Z}_{box}]$ is the same as the type $\left\langle \mathbb{Z}_{box}, \mathbb{Z}_{32}^{\mathbf{signed}} \to \mathbb{Z}_{box} \right\rangle$.

The universal type $\forall\alpha_1, \ldots, \alpha_m.t$ defines a polymorphic type, where t must be a function type. The existential type $\exists\alpha_1, \ldots, \alpha_m.t$ defines a type abstraction. The values in an existential type have the form **ty_pack**$[\exists\alpha_1, \ldots, \alpha_m.t](v, t_1, \ldots, t_m)$, where v has type $t[t_1/\alpha_1, \ldots, t_m/\alpha_m]$. The type projection $v.i$ is used for values having existential type $\exists\alpha_1, \ldots, \alpha_m.t$. If a value

		Type	Description
t	$::=$	$\mathbb{Z}_{box}$	Boxed integers
	$\mid$	$\mathbb{Z}_{pre}^{sign}$	Native integers
	$\mid$	**void**	Void type
	$\mid$	$\langle t_1, \ldots, t_m \rangle$	Tuple data
	$\mid$	t **array**	Array data
	$\mid$	**data**	Unsafe data
	$\mid$	$(t_1, \ldots, t_m) \rightarrow t$	Function type
	$\mid$	$\alpha, \beta, \ldots$	Polymorphic type vars
	$\mid$	$tv[t_1, \ldots, t_m]$	Type application
	$\mid$	$\forall \alpha_1, \ldots, \alpha_m.t$	Universal types
	$\mid$	$\exists \alpha_1, \ldots, \alpha_m.t$	Existential types
	$\mid$	$v.i$	Abstract type
tydef	$::=$	$\Lambda \alpha_1, \ldots, \alpha_m.t$	Parameterized types

Figure 2.2: FIR type system

$v = \textbf{ty\_pack}[\exists \alpha_1, \ldots, \alpha_m.t](v', t_1, \ldots, t_m)$ has type $\exists \alpha_1, \ldots, \alpha_m.t$, then $v.i$ is equivalent to t_i.

Type definitions define parameterized types. A type definition *tydef* has the form $\Lambda \alpha_1, \ldots, \alpha_m.t$, where t is a type abstracted over type parameters $\alpha_1, \ldots, \alpha_m$. Type definitions will appear as part of the program context.

2.1.2 FIR statements

Statements in the FIR are divided into two classes: the atoms a and general expressions e shown in Figure 2.3.

2.1.2.1 Atoms

The atoms a represent values, including numbers, variables, and basic arithmetic. Atoms are functional. Apart from arithmetic exceptions[1], the order of atom evaluation does not matter. The atoms include the following.

The boxed integers $\textbf{int}(i)$ have type $\mathbb{Z}_{box}$. The raw integers $\textbf{rawint}(r)$ are native, unboxed integer constants with type $\mathbb{Z}_{pre}^{sign}$. There are two forms for arithmetic: unary operations *unop* a, and binary operations a_1 *binop* a_2. The operators, shown in Figure 2.4, include the normal operations for arithmetic.

The variables v represent values defined in the program environment, described in Section 2.2. Variables are immutable: the FIR does not include a variable assignment operation.

There are three kinds of polymorphic operations. The $\textbf{ty\_apply}[t](v, t_1, \ldots, t_m)$ atom is a type application of a polymorphic value v to type arguments $t_1, \ldots, t_m$. For the application

[1] Notable arithmetic exceptions include division by zero and floating point overflow/underflow.

		Definition	Description
a	$::=$	$\mathbf{int}(i)$	Boxed integers
	$\mid$	$\mathbf{rawint}(r)$	Raw integers
	$\mid$	v	Variables
	$\mid$	$\mathbf{ty\_apply}[t](a, t_1, \ldots, t_m)$	Type application
	$\mid$	$\mathbf{ty\_pack}[t](v, t_1, \ldots, t_m)$	Existential pack
	$\mid$	$\mathbf{ty\_unpack}(v)$	Existential unpack
	$\mid$	$\mathit{unop}\, a$	Unary operation
	$\mid$	$a_1\ \mathit{binop}\ a_2$	Binary operation
e	$::=$	$\mathbf{let}\, v \colon t = a\ \mathbf{in}\ e$	Basic operations
	$\mid$	$\mathbf{let}\, v \colon t_v = (\text{“}s\text{”} : t_s)(a_1, \ldots, a_n)\ \mathbf{in}\ e$	Calls to the runtime
	$\mid$	$a(a_1, \ldots, a_n)$	Tail-call
	$\mid$	$\mathbf{special}\, \mathit{tailop}$	Special tail-call
	$\mid$	$\mathbf{match}\, a\ \mathbf{with}\ \lceil s_i \mapsto e_i \rceil_1^n$	Case analysis
	$\mid$	$\mathbf{let}\, v = \mathit{alloc}\ \mathbf{in}\ e$	Allocation
	$\mid$	$\mathbf{let}\, v \colon t = a_1[a_2]\ \mathbf{in}\ e$	Load from heap
	$\mid$	$a_1[a_2] \colon t \leftarrow a_3;\ e$	Store into heap

Figure 2.3: FIR atoms and expressions

to be well-formed, the variable v must have universal type $\forall \alpha_1, \ldots, \alpha_m.u$; the atom has type $t = u[t_1/\alpha_1, \ldots, t_m/\alpha_m]$. The $\mathbf{ty\_pack}[t](v, t_1, \ldots, t_m)$ atom performs type abstraction. It has type $t = \exists \alpha_1, \ldots, \alpha_m.u$ when v has type $u[t_1/\alpha_1, \ldots, t_m/\alpha_m]$. The $\mathbf{ty\_unpack}(v)$ atom is the elimination form for type abstraction. If v has existential type $\exists \alpha_1, \ldots, \alpha_m.u$, the atom has type $u[v.1/\alpha_1, \ldots, v.m/\alpha_m]$. The type $v.i$ represents the type parameter t_i in the original pack operation.

2.1.2.2 Expressions

The $\mathbf{let}\, v \colon t = a\ \mathbf{in}\ e$ expression forms a new scope, where the variable v is bound to the value of the atom expression a in the expression e. For the expression to be well-formed, the atom must have type t, and the expression e must be well-formed for an arbitrary value v of type t.

The external-call expression $\mathbf{let}\, v \colon t_v = (\text{“}s\text{”} : t_s)(a_1, \ldots, a_n)\ \mathbf{in}\ e$ is used to provide access to the runtime and operating system. The string “s” represents the name of a runtime function to be called with arguments $a_1, \ldots, a_n$. For the expression to be well-formed, the runtime function “s” must have type $t_s = (u_1, \ldots, u_n) \rightarrow t_v$, each atom a_i must have type u_i, and e must be well-formed given a value v with type t_v.

The tail-call $a(a_1, \ldots, a_n)$ represents a function call to the function a with arguments $a_1, \ldots, a_n$. Functions are statically defined as part of the program context, discussed in Section 2.2.1, and function definitions may not be nested. The atom a in a tail-call is always a variable v or a type-application $v[t_1, \ldots, t_m]$ where v is defined in the context. For the tail-call to be well-formed, the function a must have some type $(u_1, \ldots, u_n) \rightarrow \mathbf{void}$, and each argument a_i must have type u_i. The return type of the function is the empty type $\mathbf{void}$. There is no syntactic mechanism for using the return value of a function, and functions never return.

The special-call **special** *tailop* represents a call for process migration, or one of the speculation operations. The *tailop* operations are shown in Figure 2.4.

- The operator **migrate** $[i, a_p, a_o]\ a_{fun}(a_1, \ldots, a_n)$ defines a process migration. The argument (a_p, a_o) specifies the destination (for example, the name of a migration server), and the argument $a_{fun}(a_1, \ldots, a_n)$ specifies a function to be called once the process migrates, along with its arguments.

- The operator **speculate** $a_{fun}(a_{const}, a_1, \ldots, a_n)$ specifies entry into a speculation. This operation effectively takes a process checkpoint, which can be restored if the speculation is aborted. Once the speculation is entered, the function a_{fun} is called with arguments $a_{const}, a_1, \ldots, a_n$.

- The operator **rollback** $[a_{level}, a_{const}]$ is used to abort a speculation. The atom a_{level} is the name of the speculation to be aborted (the speculation checkpoints are saved in a list). Speculations are restarted on failure, and the atom a_{const} is a new argument to be passed to the speculation entry point when the speculation is restarted.

  For example, if the speculation was entered with **speculate** $a_{fun}(a_{const}, a_1, \ldots, a_n)$, and the speculation is aborted with the **rollback** $[a'_{level}, a'_{const}]$, the speculation is *resumed* with the tail-call $a_{fun}(a'_{const}, a_1, \ldots, a_n)$.

- The operator **commit** $[a_{level}]\ a_{fun}(a_1, \ldots, a_n)$ commits the speculation identified by a_{level}. The speculation checkpoint identified by a_{level} is discarded, and the function a_{fun} is called with arguments $a_1, \ldots, a_n$.

The match statement **match** a **with** $\lceil s_i \mapsto e_i \rceil_1^n$ is a pattern match of an integer against multiple sets. Each match case $s_i \mapsto e_i$ specifies an integer (or raw integer) set s_i and an expression e_i to be evaluated if $a \in s_i$. Evaluation is ordered and total. Evaluation chooses the *first* match that succeeds, and the match statement is well-formed only if there is a match case for any possible value of a.

The aggregate data areas include tuples, arrays and raw data. The **let** $v = \mathit{alloc}$ **in** e expression allocates a data aggregate, using one of the *alloc* forms shown in Figure 2.4.

Values are projected from an aggregate data area using the **let** $v\colon t = a_1[a_2]$ **in** e expression. For the expression to be well-formed, a_1 must be an aggregate, and a_2 must be a valid index into the aggregate. All fields in aggregates are mutable. The $a_1[a_2]\colon t \leftarrow a_3;\ e$ expression assigns value a_3 to field a_2 in aggregate a_1.

		Definition	Description
unop	::=	$- \mid ! \mid \cdots$	Unary operations
binop	::=	$+ \mid - \mid * \mid / \mid \cdots$	Binary operations
alloc	::=	$\langle a_1, \ldots, a_n \rangle : t$	Tuple allocation
	$\mid$	$\textbf{array}(a_{size}, a_{init}) : t$	Array allocation
	$\mid$	$\textbf{malloc}(a) : t$	Rawdata allocation
tailop	::=	$\textbf{migrate}\ [i, a_p, a_o]\ a_{fun}(a_1, \ldots, a_n)$	System migration
	$\mid$	$\textbf{speculate}\ a_{fun}(a_{const}, a_1, \ldots, a_n)$	Speculation entry
	$\mid$	$\textbf{rollback}\ [a_{level}, a_{const}]$	Speculation rollback
	$\mid$	$\textbf{commit}\ [a_{level}]\ a_{fun}(a_1, \ldots, a_n)$	Speculation commit

Figure 2.4: FIR operators

2.2 Judgments

All judgments, including type and well-formedness judgments, are defined with respect to an environment Γ, also called a *context*. The environment contains both variable declarations of the form $v\colon t$, and variable definitions of the form $v\colon t = b$. The declaration $v\colon t$ specifies that a variable v has an unspecified value of type t. The definition $v\colon t = b$ specifies that variable v has the value b, and b has type t.

2.2.1 Heap and store values

The definitions in the context use values of two sorts: heap values h, and store values b, shown in Figure 2.5. The *heap values* represent atoms that have been fully evaluated. In general, an atom evaluates to a number, a label, or a variable that represents a store value. The *store values* are the values in a program store. These include heap values, functions, "packed" values with existential type, and data in each of the aggregate data types: tuples, arrays, and rawdata.

Functions are universally quantified, with type parameters $\alpha_1, \ldots, \alpha_m$ and actual parameters $v_1, \ldots, v_n$. Elements of type **data** are represented abstractly using the form $\langle \mathrm{C} \rangle$; the elements in the data area are not explicitly described.

2.2.2 Kinds

The program types are also defined as part of the context Γ. For presentation purposes, the program types are classified with *kinds*, which have the following form.

$$\begin{array}{rcl} k_s & ::= & \omega \mid \Omega \\ k & ::= & \omega^m \to k_s \end{array}$$

		Definition	Description
h	$::=$	$\mathbf{int}(i)$	Boxed integers
	$\vert$	$\mathbf{rawint}(r)$	Raw integers
	$\vert$	v	Variables
b	$::=$	h	Heap values
	$\vert$	$\Lambda\alpha_1, \ldots, \alpha_m.\lambda v_1, \ldots, v_n.e$	Functions
	$\vert$	$\mathbf{ty\_pack}[t](v, t_1, \ldots, t_m)$	Type packing
	$\vert$	$\langle h_1, \ldots, h_n \rangle$	Tuples
	$\vert$	$\langle \mathrm{C} \rangle$	Raw data

Figure 2.5: Heap and store values

Definition			Description
def	$::=$	$v\colon t$	Variable declaration
	$\vert$	$v\colon t = b$	Variable definition
	$\vert$	$tv\colon k$	Type declaration
	$\vert$	$tv\colon k = tydef$	Type definition
Γ	$::=$	ϵ	Empty environment
	$\vert$	Γ, def	Adding a definition
$\Gamma \vdash \diamond$			Context Γ is well-formed
$\Gamma \vdash tydef_1 = tydef_2 : k$			$tydef_1$ and $tydef_2$ are equal type definitions (or types)
$\Gamma \vdash tv_1 = tv_2 : k$			tv_1 and tv_2 are equal type definitions
$\Gamma \vdash a : t$			Atom a has type t
$\Gamma \vdash b : t$			Store value b has type t
$\Gamma \vdash e : t$			Program e has type t

Figure 2.6: Program contexts and judgments

The kind k_s classifies the type definitions *tydef* as “small” types ω and “large” untagged types Ω, primarily to support efficient garbage collection. The general kind $k = \omega^m \to k_s$ represents a parameterized type definition *tydef*. The number of parameters m may be any nonnegative integer. If $m = 0$, the type parameters may be omitted.

2.2.3 Contexts and judgments

A program context Γ is defined as a set of mutually-recursive declarations and definitions, as shown in Figure 2.6. There are two forms of definitions. The type definition $tv\colon k = tydef$ defines a type named tv, having kind k, and value *tydef*. The variable definition $v\colon t = b$ defines a variable named v, with type t and store value b. For each definition form there is a corresponding *declaration* form.

This thesis assumes each variable and type variable in a context is defined/declared at most once, and uses alpha-renaming to rename variables as appropriate. The meta-variable d is used to represent a definition or declaration *def*.

The judgment $\Gamma \vdash \diamond$ specifies that the context Γ is well-formed. A context is well-formed if all of its declarations and definitions are well-formed. For each declaration $v\colon t$ and definition $v\colon t = b$, the term t must be a well-formed type, and the value b must have type t. Similarly, all type definitions in Γ must be well-formed.

The type system includes an equational theory of types. The judgment $\Gamma \vdash \mathit{tydef}_1 = \mathit{tydef}_2 : k$ is a type definition equality judgment. When the judgment is true, tydef_1 and tydef_2 have the specified kind, and they are equal. There is no separate membership judgment $\Gamma \vdash \mathit{tydef} : k$.

The judgments $\Gamma \vdash a : t$, $\Gamma \vdash b : t$, $\Gamma \vdash e : t$ express typing relations for atoms, store values, and expressions, respectively.

2.3 FIR operational semantics

Evaluation is defined on *programs*, which include three parts: the current environment Γ, a checkpoint environment $\mathcal{C}$, which is an *ordered list* of checkpoints, and an expression e to be evaluated. Checkpoints are required for speculations, which are discussed in Section 2.3.3. A checkpoint $\langle \Gamma, f(\diamond, a_1, \ldots, a_n)\rangle$ contains a context Γ, and a function $f(\diamond, a_1, \ldots, a_n)$, where $\diamond$ is a special speculation parameter. The function is called if evaluation is resumed from the checkpoint.

$$\begin{array}{rcll} C & ::= & \langle \Gamma, f(\diamond, a_1, \ldots, a_n)\rangle & \text{Single checkpoint} \\ \mathcal{C} & ::= & C_m; \ldots; C_1 & \text{Checkpoint environment} \end{array}$$

Definition 2.3.1 Fully-defined contexts

A context Γ *is said to be* fully-defined *if every variable* v *in the context is defined with the form* $v\colon t = b$ *and every type variable* tv *is defined with the form* $tv\colon k = \mathit{tydef}$.

Definition 2.3.2 Programs

A program *is either the special term* **error***, or it is a triple* $(\Gamma \mid \mathcal{C} \mid e)$ *that satisfies the following conditions.*

- Γ *is fully-defined and* $\Gamma \vdash e : \mathbf{void}$*,*
- *For each* $\langle \Gamma', f(\diamond, a_1, \ldots, a_n)\rangle \in \mathcal{C}$*, the context* Γ' *is fully-defined, and the judgment* $\Gamma', v_{const}\colon \mathbb{Z}_{32}^{\mathbf{signed}} \vdash f(v_{const}, a_1, \ldots, a_n) : \mathbf{void}$ *holds.*

Intuitively, the **error** term specifies a runtime error during program evaluation (such as an out-of-bounds array access, or a runtime type error during a subscripting operation). The **error** term is similar to an exception, and may assume any type.

Evaluation is a relation on programs. The relation

$$(\Gamma \mid \mathcal{C} \mid e) \rightarrow (\Gamma' \mid \mathcal{C}' \mid e')$$

specifies that the program $(\Gamma \mid \mathcal{C} \mid e)$ evaluates in one step to the program $(\Gamma' \mid \mathcal{C}' \mid e')$. The relation

$$(\Gamma \mid \mathcal{C} \mid e) \rightarrow \textbf{error}$$

specifies that evaluation of the program $(\Gamma \mid \mathcal{C} \mid e)$ results in a runtime error in one step. This thesis only describes the rules for external calls, special-calls, and several subscripting operations on the heap. The complete operational semantics is described in the technical report [11].

2.3.1 External calls

For external calls, we defer to a semantic interpretation function $[\![\text{“s”}]\!]$ that specifies the interpretation of the built-in function named “s”.

$$\begin{array}{l}(\Gamma \mid \mathcal{C} \mid \textbf{let}\, v\colon t = (\text{“}s\text{”} : (u_1, \ldots, u_n) \rightarrow t)(h_1, \ldots, h_n)\ \textbf{in}\ e) \rightarrow \\ \quad (\Gamma, v\colon t = [\![\text{“s”}]\!](h_1, \ldots, h_n) \mid \mathcal{C} \mid e) \end{array} \qquad \textrm{RED-LETEXT}$$

It is important to note that the semantic interpretation function $[\![\text{“s”}]\!]$ may reference data both within and external to the context Γ. This is significant in determining what state is included in checkpoints.

2.3.2 Process migration

When a process migrates, the entire process (including Γ, $\mathcal{C}$, and the continuation function) migrates to a new location, which is just another runtime environment. The operational definition of migration is transparent by design. The program context must not change during process migration. This decouples the process execution from the process location, allowing transparent fault recovery in distributed systems. This decoupling is fairly significant, since otherwise a process might have to spend considerable time reacquiring machine-specific resources and rebuilding its state when it is transferred to another machine.

In the **migrate** $[j, a_{ptr}, a_{off}]\ a_{fun}(a_1, \ldots, a_n)$ special-call expression, the atoms a_{ptr} and a_{off} specify a string (as a rawdata block and offset) that describes the migration protocol and target (for example, a machine name). The number j is a unique identifier used by the runtime. Operationally, evaluation of the expression leads to process migration followed by the evaluation of the tail-call $a_{fun}(a_1, \ldots, a_n)$.

$$(\Gamma \mid \mathcal{C} \mid \textbf{special}\,\textbf{migrate}\ [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n)) \rightarrow (\Gamma \mid \mathcal{C} \mid f(h_1, \ldots, h_n)) \quad \textrm{RED-SYSMIGRATE}$$

The implementation of these semantics is discussed in Section 4.3.

All state recorded in $\mathcal{C}$ is migrated in the system migration call. Note that state that is external to $\mathcal{C}$, such as the machine state accessed through external calls, may not necessarily be migrated. Currently, the memory heap and program registers (both explicitly represented in $\mathcal{C}$), and the process code are migrated. I/O to files and devices is not supported directly by the FIR at this time, and must use the external call interface. The I/O state is not currently migrated.

2.3.3 Speculations

Speculations are entered with the special-call **speculate** $a_{fun}(a_{const}, a_1, \ldots, a_n)$. The runtime adds a process checkpoint to the checkpoint environment $\mathcal{C}$. This checkpoint can be restored later if the speculation is aborted. Evaluation proceeds with a tail-call $a_{fun}(a_{const}, a_1, \ldots, a_n)$, and the speculative call is treated identically to this tail-call for typing purposes. For technical reasons, a_{const} must have type $\mathbb{Z}_{32}^{\mathbf{signed}}$.

$$\begin{array}{ll}(\Gamma \mid \mathcal{C} \mid \mathbf{special}\,\mathbf{speculate}\ f(h_{const}, h_1, \ldots, h_n)) \rightarrow & \\ \quad (\Gamma \mid \langle \Gamma, f(\diamond, h_1, \ldots, h_n)\rangle\,; \mathcal{C} \mid f(h_{const}, h_1, \ldots, h_n)) & \textsc{Red-Spec}\end{array}$$

The **rollback** $[a_{level}, a_{const}]$ special-call aborts a speculation. It is possible to enter several speculations simultaneously (in the source program, speculations are typically nested). The atom a_{level} is an integer that identifies the speculation level, and a_{const} is a speculation parameter.

When a speculation checkpoint $\langle \Gamma, a_{fun}(\diamond, a_1, \ldots, a_n)\rangle$ is rolled back with the **rollback** $[i, j]$ special-call expression to level i with speculation parameter j, evaluation proceeds as a tail-call $a_{fun}(j, a_1, \ldots, a_n)$, using the original process context Γ and the truncated checkpoint environment $C_i; \ldots; C_1$. All checkpoints with level higher than i are discarded. The level that was rolled back is re-entered by this primitive; in effect, the state that is restored is the state captured immediately after the level was entered.

$$\begin{array}{ll}(\Gamma' \mid C_m; \ldots; C_i = \langle \Gamma, f(\diamond, \lceil h_k \rceil_{k=1}^{n})\rangle\,; \ldots; C_1 \mid \mathbf{special}\,\mathbf{rollback}\ [i, j]) \rightarrow & \\ \quad (\Gamma \mid C_i; C_{i-1}; \ldots; C_1 \mid f(j, \lceil h_k \rceil_{k=1}^{n})) & \\ \quad \mathbf{when}\, i \in \{1 \ldots m\} & \textsc{Red-Spec-Rollback}\end{array}$$

In most cases, rollback operates on the most recently entered level. As a short-hand, when $i = 0$ the runtime will roll back only the most recent level.

$$\begin{array}{ll}(\Gamma' \mid C_m = \langle \Gamma, f(\diamond, h_1, \ldots, h_n)\rangle\,; \ldots; C_1 \mid \mathbf{special}\,\mathbf{rollback}\ [0, j]) \rightarrow & \\ \quad (\Gamma \mid C_m; \ldots; C_1 \mid f(j, h_1, \ldots, h_n)) & \textsc{Red-Spec-Rollback-2}\end{array}$$

Also, it is an error to specify a checkpoint that does not exist.

$$\begin{array}{ll}(\Gamma \mid C_m; \ldots; C_1 \mid \mathbf{special}\,\mathbf{rollback}\ [i, j]) \rightarrow \mathbf{error} & \\ \quad \mathbf{when}\, i \notin \{0 \ldots m\} \vee m = 0 & \textsc{Red-Spec-Rollback-Error}\end{array}$$

Speculations are committed with the special-call **commit** $[i]$ $a_{fun}(a_1, \ldots, a_n)$. Operationally, the checkpoint is deleted from the checkpoint context and evaluation continues with a tail-call to the function $a_{fun}(a_1, \ldots, a_n)$.

$$\begin{array}{ll}(\Gamma \mid C_m; \ldots; C_1 \mid \mathbf{special}\,\mathbf{commit}\ [i]\ f(h_1, \ldots, h_n)) \rightarrow & \\ \quad (\Gamma \mid C_m; \ldots; C_{i+1}; C_{i-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n)) & \\ \quad \mathbf{when}\, i \in \{1 \ldots m\} & \textsc{Red-Spec-Commit}\end{array}$$

In most cases, commit operates on the most recently entered level. As a short-hand, when $i = 0$ the runtime will commit the most recent level.

$$\begin{array}{ll}(\Gamma \mid C_m; \ldots; C_1 \mid \textbf{special}\,\textbf{commit}\ [0]\ f(h_1, \ldots, h_n)) \rightarrow & \\ \quad (\Gamma \mid C_{m-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n)) & \textsc{Red-Spec-Commit-2}\end{array}$$

Also, it is an error to specify a checkpoint that does not exist.

$$\begin{array}{ll}(\Gamma \mid C_m; \ldots; C_1 \mid \textbf{special}\,\textbf{commit}\ [i]\ f(h_1, \ldots, h_n)) \rightarrow \textbf{error} & \\ \quad \textbf{when}\, i \notin \{0 \ldots m\} \vee m = 0 & \textsc{Red-Spec-Commit-Error}\end{array}$$

The FIR does not syntactically require entry and commit operations to be balanced. Instead, programs that attempt to rollback to or commit a checkpoint that does not exist evaluate to the **error** term.

The implementation of these semantics is discussed in Section 4.4.

2.3.3.1 Speculations and external state

All state recorded in $\mathcal{C}$ can be rolled back by a speculation. However, state that is external to $\mathcal{C}$, such as the machine state accessed through external calls, including I/O to files and devices, is not currently rolled back.

2.3.3.2 Speculations and transactions

Speculations bear a certain resemblance to database transactions. Indeed, on a single-process system there is little difference, as both mechanisms accommodate rollback to a known-consistent state when an abnormal condition occurs. However, in the presence of other processes, speculations can be more flexible than traditional transactions, particularly with regard to atomicity. For a process in a traditional transaction, any state modifications it makes must be opaque to processes not involved in the transaction. An external process wishing to utilize the state modifications must block until the transaction is committed or aborted. With speculations, we can conceive other processes dynamically joining a speculation that is in progress, optimistically assuming that the speculation will complete with the new state in place.

2.3.4 Subscripting operations

The subscripting operations correspond to the aggregate values: tuples, arrays, and rawdata.

The projection for a tuple $\langle h_0, \ldots, h_{n-1}\rangle$ and index j is the element h_j. A tuple operation is always successful because the offset is a constant, and the type system discussed in Chapter 3 ensures that the offset is within bounds.

$$\begin{array}{ll}((\Gamma', v_2\colon u' = \langle h_0, \ldots, h_{n-1}\rangle)\ \textbf{as}\ \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1\colon u = v_2[j]\ \textbf{in}\ e) \rightarrow & \\ \quad (\Gamma, v_1\colon u = h_j \mid \mathcal{C} \mid e) & \textsc{Red-LetSub-Tuple}\end{array}$$

$$((\Gamma', v\colon u' = \langle h_0, \ldots, h_{n-1}\rangle) \text{ \textbf{as} } \Gamma \mid \mathcal{C} \mid v[j]\colon u \leftarrow h;\ e) \rightarrow (\Gamma', v\colon u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1}\rangle \mid \mathcal{C} \mid e) \qquad \textsc{Red-SetSub-Tuple}$$

Arrays are similar to tuples when the array index is in bounds.

$$((\Gamma', v_2\colon u' = \langle h_0, \ldots, h_{n-1}\rangle) \text{ \textbf{as} } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \rightarrow (\Gamma, v_1 : u = h_j \mid \mathcal{C} \mid e) \textbf{ when}\, j \in \{0 \ldots n-1\} \qquad \textsc{Red-LetSub-Array}$$

$$((\Gamma', v\colon u' = \langle h_0, \ldots, h_{n-1}\rangle) \text{ \textbf{as} } \Gamma \mid \mathcal{C} \mid v[j]\colon u \leftarrow h;\ e) \rightarrow (\Gamma', v\colon u' = \langle h_0, \ldots, h_{j-1}, h, h_{j+1}, \ldots, h_{n-1}\rangle \mid \mathcal{C} \mid e) \textbf{ when}\, j \in \{0 \ldots n-1\} \qquad \textsc{Red-SetSub-Array}$$

It is an error for an array subscript to be out-of-bounds.

$$((\Gamma', v_2\colon t_2 = \langle h_0, \ldots, h_{n-1}\rangle) \text{ \textbf{as} } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : t_1 = v_2[j] \textbf{ in } e) \rightarrow \textbf{error} \textbf{ when}\, j \notin \{0 \ldots n-1\} \qquad \textsc{Red-LetSub-Array-Error}$$

$$((\Gamma', v\colon t_1 = \langle h_0, \ldots, h_{n-1}\rangle) \text{ \textbf{as} } \Gamma \mid \mathcal{C} \mid v[j]\colon t_2 \leftarrow h;\ e) \rightarrow \textbf{error} \textbf{ when}\, j \notin \{0 \ldots n-1\} \qquad \textsc{Red-SetSub-Array-Error}$$

For the unsafe aggregates, the subscript operations perform a runtime type check. Rather than specify the operations here, we rely on an operational semantics provided by the runtime, described in Chapter 4.

For aggregates of **data** type, we assume existence of a runtime function $\textbf{runtime}(\Gamma \mid \langle \textsc{c} \rangle\, [j] : t)$ that projects a valid value h of type t from data area $\langle \textsc{c} \rangle$, or results in an error. We also assume the existence of a runtime function $\textbf{runtime}(\Gamma \mid \langle \textsc{c} \rangle\, [j] : t \leftarrow h)$ to store a value in a rawdata aggregate, returning a new value $\langle \textsc{c}' \rangle$ or producing an error.

$$\textbf{runtime}(\Gamma \mid \langle \textsc{c} \rangle\, [j] : t) = \begin{cases} h & \text{if } \Gamma \vdash h : t \\ \textbf{error} & \text{otherwise} \end{cases}$$

$$\textbf{runtime}(\Gamma \mid \langle \textsc{c} \rangle\, [j] : t \leftarrow h) = \begin{cases} \langle \textsc{c}' \rangle & \text{assignment succeeds} \\ \textbf{error} & \text{otherwise} \end{cases}$$

Given these runtime functions, a rawdata subscript operation returns the value given by the runtime, or produces an error.

$$((\Gamma', v_2\colon u' = \langle \textsc{c} \rangle) \text{ \textbf{as} } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \rightarrow (\Gamma, v_1 : u = h \mid \mathcal{C} \mid e) \textbf{ when}\, \textbf{runtime}(\Gamma \mid \langle \textsc{c} \rangle\, [j] : u) = h \qquad \textsc{Red-LetSub-RawData}$$

$$((\Gamma', v_2\colon u' = \langle \textsc{c} \rangle) \text{ \textbf{as} } \Gamma \mid \mathcal{C} \mid \textbf{let}\, v_1 : u = v_2[j] \textbf{ in } e) \rightarrow \textbf{error} \textbf{ when}\, \textbf{runtime}(\Gamma \mid \langle \textsc{c} \rangle\, [j] : u) = \textbf{error} \qquad \textsc{Red-LetSub-RawData-Error}$$

$$((\Gamma', v\colon u' = \langle \textsc{c} \rangle) \text{ \textbf{as} } \Gamma \mid \mathcal{C} \mid v[j]\colon u \leftarrow h;\ e) \rightarrow (\Gamma', v\colon u' = \langle \textsc{c}' \rangle \mid \mathcal{C} \mid e) \textbf{ when}\, \textbf{runtime}(\Gamma \mid \langle \textsc{c} \rangle\, [j] : u \leftarrow h) = \langle \textsc{c}' \rangle \qquad \textsc{Red-SetSub-RawData}$$

$$((\Gamma', v\colon u' = \langle \text{C} \rangle)\ \mathbf{as}\ \Gamma \mid \mathcal{C} \mid v[j]\colon u \leftarrow h;\ e) \rightarrow \mathbf{error}$$
$$\mathbf{when\,runtime}(\Gamma \mid \langle \text{C} \rangle\ [j] : u \leftarrow h) = \mathbf{error} \qquad \textsc{Red-SetSub-RawData-Error}$$

Chapter 3

Safety

The FIR features a strong typing system that is used to ensure the safety of programs. By showing that a FIR program is well-typed, we can prove that the program is *safe* — it will not perform an illegal memory operation, and it will only branch to an execution point that is an entry point for either a function within the program, or a predetermined external function. With this property, system migration can be performed in a safe manner, even among machines that are not mutually trusting.

This chapter describes the properties of the FIR type system that contribute to demonstrating the safety of programs. The type rules give rise to preservation and progress theorems, which ensure that a program's type is preserved during evaluation and that evaluation will not stall. This chapter also introduces the runtime safety checks that are required to enforce certain typing rules in the presence of unsafe aggregate types (such as **data** for C programs).

3.1 Typing rules

The typing rules for FIR terms are presented in this section as a set of inference rules. Only syntactically valid terms are covered by these rules. Each rule consists of a judgment from Figure 2.6 as the conclusion, and a list of judgments and other "side-conditions" (such as $i \in \mathbb{Z}_{box}$) as premises.

The shorthand $\Gamma \vdash \lceil J_i \rceil_1^n$ denotes a list of judgments. This is equivalent to listing all judgments $\Gamma \vdash J_1, \ldots, \Gamma \vdash J_n$. When $n = 0$, this notation expands to no judgments. This thesis only describes the rules for external calls, special-calls, and subscripting operations. The complete type system is described in the technical report [11].

3.1.1 External call typing rule

External calls are defined by the runtime environment to access the runtime or the operating system. For example, the runtime typically exports functions to gather statistics and control the garbage collector, and it also exports a set of system calls. The description of these functions is specific to the runtime and beyond the scope of this thesis. For our purposes, we assume there is some interpretation $[\![\text{“s”}]\!]$ that defines a function that corresponds to the runtime operation.

$$\frac{\Gamma \vdash \lceil a_i : u_i \rceil_1^n \quad \Gamma, v\colon t_1 \vdash e : t_2 \quad \Gamma \vdash [\![\text{“s”}]\!] : (u_1, \ldots, u_n) \to t_1}{\Gamma \vdash \mathbf{let}\, v\colon t_1 = (\text{“}s\text{”} : (u_1, \ldots, u_n) \to t_1)(a_1, \ldots, a_n)\ \mathbf{in}\ e : t_2} \qquad \textsc{Ty-LetExt}$$

3.1.2 Migration typing rule

For simplicity, the expression typing rule Ty-Special-Call uses a common form for all of the special-calls. If *tailop* is a special-call, then **special** *tailop* is an expression.

$$\frac{\Gamma \vdash \mathit{tailop} : \mathbf{special}\, t}{\Gamma \vdash \mathbf{special}\, \mathit{tailop} : t} \qquad \textsc{Ty-Special-Call}$$

Process migration is specified with the **migrate** $[j, a_{ptr}, a_{off}]\ a_f(a_1, \ldots, a_n)$ special-call expression, which is described in Section 4.3. The atoms a_{ptr} and a_{off} specify a string (as a rawdata block and offset) that describes the migration protocol and target (for example, a machine name). Section 4.3.1 describes the possible protocols available for system migration. The $a_f(a_1, \ldots, a_n)$ is a tail-call to be performed once the process has migrated. The number j is a unique identifier used by the runtime.

$$\frac{\begin{array}{cc} & \Gamma \vdash a_{ptr} : \mathbf{data} \\ j \in \mathbb{Z}_{box} & \Gamma \vdash a_{off} : \mathbb{Z}_{32}^{\mathbf{signed}} \\ \Gamma \vdash \lceil a_i : t_i \rceil_1^n & \Gamma \vdash a_f : (t_1, \ldots, t_n) \to \mathbf{void} \end{array}}{\Gamma \vdash \mathbf{migrate}\ [j, a_{ptr}, a_{off}]\ a_f(a_1, \ldots, a_n) : \mathbf{special\, void}} \qquad \textsc{Ty-SysMigrate}$$

3.1.3 Speculation typing rules

The speculation special-calls are described in Section 4.4. The **speculate** $a_f(a_{const}, a_1, \ldots, a_n)$ expression specifies entry into a new speculation which may be later committed or rolled back. The speculate call instructs the runtime to establish a process checkpoint that may be used for rollback (or discarded if the speculation is committed). Otherwise, the speculate call acts exactly like a tail-call to a function a_f with arguments $(a_{const}, a_1, \ldots, a_n)$. For technical reasons, the first argument is currently restricted to have type $\mathbb{Z}_{32}^{\mathbf{signed}}$.

$$\frac{\Gamma \vdash a_{const} : \mathbb{Z}_{32}^{\mathbf{signed}} \quad \Gamma \vdash \lceil a_i : t_i \rceil_1^n \quad \Gamma \vdash a_f : (\mathbb{Z}_{32}^{\mathbf{signed}}, t_1, \ldots, t_n) \to \mathbf{void}}{\Gamma \vdash \mathbf{speculate}\ a_f(a_{const}, a_1, \ldots, a_n) : \mathbf{special\, void}} \qquad \textsc{Ty-Spec}$$

It is possible to enter several speculations simultaneously. While speculations are typically nested in the source program, the type system does not enforce this structural constraint. The type system also does not require entry and commits to be balanced. This error condition is handled by a runtime check in the operational semantics. Each program checkpoint is identified by an integer a_{level} known as a *speculation level*; speculation levels are described in further detail in section 4.4.1. When a speculation at level a_{level} is aborted, or "rolled back," all speculations with more recent identifiers are discarded and the program state is restored to the state immediately *after* entry into the speculation. If the speculation was initiated with **speculate** $a'_{fun}(a'_{const}, a'_1, \ldots, a'_n)$, the **rollback** $[a_{level}, a_{const}]$ special-call expression resumes execution with a tail-call to $a'_{fun}(a_{const}, a'_1, \ldots, a'_n)$.

$$\frac{\Gamma \vdash a_{level} : \mathbb{Z}_{32}^{\mathbf{signed}} \quad \Gamma \vdash a_{const} : \mathbb{Z}_{32}^{\mathbf{signed}}}{\Gamma \vdash \mathbf{rollback}\ [a_{level}, a_{const}] : \mathbf{special\,void}} \quad \textsc{Ty-SpecRollback}$$

Speculations can be committed in any order. The expression **commit** $[a_{level}]\ a_f(a_1, \ldots, a_n)$ specifies that speculation a_{level} be committed. Committing a speculation instructs the runtime to discard the program checkpoint identified by a_{level}. Once committed, the commit call acts like the tail-call $a_f(a_1, \ldots, a_n)$.

$$\frac{\Gamma \vdash a_{level} : \mathbb{Z}_{32}^{\mathbf{signed}} \quad \Gamma \vdash \lceil a_i : t_i \rceil_1^n \quad \Gamma \vdash a_f : (t_1, \ldots, t_n) \to \mathbf{void}}{\Gamma \vdash \mathbf{commit}\ [a_{level}]\ a_f(a_1, \ldots, a_n) : \mathbf{special\,void}} \quad \textsc{Ty-SpecCommit}$$

3.1.4 Subscripting typing rules

Elements in the aggregate types (tuples, arrays, and rawdata) are accessed using the $\mathbf{let}\, v\colon t_1 = a_1[a_2]\ \mathbf{in}\ e$ expression. All entries in aggregate blocks are mutable; an entry is replaced using the $a_1[a_2]\colon t_1 \leftarrow a_3;\ e$ expression.

In general, $\mathbf{let}\, v\colon t_1 = a_1[a_2]\ \mathbf{in}\ e$ is a well-formed expression if all of the following are true: the atom a_1 has an aggregate type, the atom a_2 is a valid index into a_1, the element of a_1 at location a_2 has type t_1, and expression e is well-formed assuming v has type t_1.

Similarly, $a_1[a_2]\colon t_1 \leftarrow a_3;\ e$ is well-formed if the following are true: the atom a_1 has aggregate type, the atom a_2 is a valid index into a_1, the element of a_1 at location a_2 has type t_1, the atom a_3 has type t_1, and expression e is well-formed (with no extra assumptions).

For the subscripting operation $a_1[a_2]$ on tuple aggregates, the index a_2 must be a constant j. If the tuple a_1 has type $\langle u_0, \ldots, u_{n-1} \rangle$, the element has type u_j.

$$\frac{\Gamma \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle \quad \Gamma, v_1\colon u \vdash e : t}{\Gamma \vdash \mathbf{let}\, v_1\colon u = a_1[j]\ \mathbf{in}\ e : t} \quad \textsc{Ty-LetSub-Tuple}$$

$$\frac{\Gamma \vdash a_1 : \langle u_0, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{n-1} \rangle \quad \Gamma \vdash a_3 : u \quad \Gamma \vdash e : t}{\Gamma \vdash a_1[j]\colon u \leftarrow a_3;\ e : t} \quad \textsc{Ty-SetSub-Tuple}$$

Unlike tuples, all the elements in an array have the same type. The subscripting operation $a_1[a_2]$ is well-formed if the index a_2 is a valid index, the aggregate a_1 has array type u **array**, and the element being accessed has type u.

$$\frac{\Gamma \vdash a_1 : u\ \mathbf{array} \quad \Gamma \vdash a_2 : \mathbb{Z}_{box} \quad \Gamma, v\colon u \vdash e : t}{\Gamma \vdash \mathbf{let}\, v\colon u = a_1[a_2]\ \mathbf{in}\ e : t} \quad \textsc{Ty-LetSub-Array}$$

$$\frac{\Gamma \vdash a_1 : u\ \textbf{array} \quad \Gamma \vdash a_2 : \mathbb{Z}_{box} \quad \Gamma \vdash a_3 : u \quad \Gamma \vdash e : t}{\Gamma \vdash a_1[a_2] \colon u \leftarrow a_3;\ e : t} \quad \textsc{Ty-SetSub-Array}$$

For **data** blocks used by C programs, the load and store operations may manipulate values in memory with arbitrary types. Therefore, the typing rules cannot determine the type of the value being accessed. For safety, the runtime environment generates a runtime safety check to verify that the value being accessed has a compatible type. Runtime safety checks are discussed in more detail in Section 4.1.5.

$$\frac{\Gamma \vdash a_1 : \textbf{data} \quad \Gamma \vdash a_2 : \mathbb{Z}_{box} \quad \Gamma, v \colon u \vdash e : t}{\Gamma \vdash \textbf{let}\, v \colon u = a_1[a_2]\ \textbf{in}\ e : t} \quad \textsc{Ty-LetSub-RawData}$$

$$\frac{\Gamma \vdash a_1 : \textbf{data} \quad \Gamma \vdash a_2 : \mathbb{Z}_{box} \quad \Gamma \vdash a_3 : u \quad \Gamma \vdash e : t}{\Gamma \vdash a_1[a_2] \colon u \leftarrow a_3;\ e : t} \quad \textsc{Ty-SetSub-RawData}$$

3.2 Preservation

The proof of type-safety has two parts. The PRESERVATION theorem 3.2.1 shows that types are preserved during program reduction. The PROGRESS theorem 3.3.1 shows that for well-typed programs, if the expression e being evaluated is not a value, then there is a reduction rule that can be used to evaluate the program one more step. These safety properties allow for provably safe migration of code and data from one system to another, in networks where the machines are not mutually trusting.

Theorem 3.2.1 PRESERVATION *If* $(\Gamma_r \mid \mathcal{C}_r \mid e_r)$ *is a program and* $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \rightarrow (\Gamma_c \mid \mathcal{C}_c \mid e_c)$*, then* $(\Gamma_c \mid \mathcal{C}_c \mid e_c)$ *is a valid program.*

This can be proven by a case analysis on the reduction. The proof for the full FIR language is given in the technical report [11]. This thesis gives a sketch of the proof for the special calls only. Note that if the program $(\Gamma_r \mid \mathcal{C}_r \mid e_r)$ is valid, we can construct a typing proof that $\Gamma_r \vdash e : \textbf{void}$.

SysMigrate Suppose the reduction uses the rule RED-SYSMIGRATE.

$$(\Gamma_r \mid \mathcal{C}_r \mid \textbf{special}\,\textbf{migrate}\ [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n)) \rightarrow (\Gamma_r \mid \mathcal{C}_r \mid f(h_1, \ldots, h_n))$$

The proof of typing must use the rule TY-SYSMIGRATE.

$$\frac{\begin{array}{cc} & \Gamma_r \vdash h_{ptr} : \textbf{data} \\ j \in \mathbb{Z}_{box} & \Gamma_r \vdash h_{off} : \mathbb{Z}_{32}^{\textbf{signed}} \\ \Gamma_r \vdash \lceil h_i : t_i \rceil_1^n & \Gamma_r \vdash f : (t_1, \ldots, t_n) \rightarrow \textbf{void} \end{array}}{\Gamma_r \vdash \textbf{migrate}\ [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n) : \textbf{special}\,\textbf{void}}$$

From the premises $\Gamma_r \vdash \lceil h_i : t_i \rceil_1^n$ and $\Gamma_r \vdash f : (t_1, \ldots, t_n) \rightarrow \textbf{void}$, we can infer that $\Gamma_r \vdash f(h_1, \ldots, h_n) : \textbf{void}$. Since Γ_r and $\mathcal{C}_r$ are unaltered by this rule, the proof case is complete.

Spec Suppose the reduction uses the rule RED-SPEC.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \textbf{special}\,\textbf{speculate}\ f(h_{const}, h_1, \ldots, h_n)) \to \\ \quad (\Gamma_r \mid \langle \Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle\,; \mathcal{C}_r \mid f(h_{const}, h_1, \ldots, h_n))\end{array}$$

The proof of typing must use the rule TY-SPEC.

$$\frac{\Gamma_r \vdash h_{const} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \Gamma_r \vdash \lceil h_i : t_i \rceil_1^n \quad \Gamma_r \vdash f : (\mathbb{Z}_{32}^{\textbf{signed}}, t_1, \ldots, t_n) \to \textbf{void}}{\Gamma_r \vdash \textbf{speculate}\ f(h_{const}, h_1, \ldots, h_n) : \textbf{special}\,\textbf{void}}$$

From the premises $\Gamma_r \vdash h_{const} : \mathbb{Z}_{32}^{\textbf{signed}}$, $\Gamma_r \vdash \lceil h_i : t_i \rceil_1^n$, and $\Gamma_r \vdash f : (t_1, \ldots, t_n) \to \textbf{void}$, we can infer that the checkpoint $\langle \Gamma_r, f(\diamond, h_1, \ldots, h_n)\rangle$ is well-formed, and that $\Gamma_r \vdash f(h_1, \ldots, h_n) :$ **void**.

SpecRollback Suppose the reduction uses the rule RED-SPEC-ROLLBACK.

$$\begin{array}{l}(\Gamma'_r \mid C_m; \ldots; C_i = \langle \Gamma_r, f(\diamond, \lceil h_k \rceil_{k=1}^n)\rangle\,; \ldots; C_1 \mid \textbf{special}\,\textbf{rollback}\ [i, j]) \to \\ \quad (\Gamma_r \mid C_i; C_{i-1}; \ldots; C_1 \mid f(j, \lceil h_k \rceil_{k=1}^n)) \\ \quad \textbf{when}\, i \in \{1 \ldots m\}\end{array}$$

By assumption, the checkpoint context $\mathcal{C}_r$ is well-formed, and $\Gamma_r \vdash f(j, h_1, \ldots, h_n) :$ **void**. Since the reduction only removes checkpoints from the context, the checkpoint context remains well-formed. The argument for RED-SPEC-ROLLBACK-2 is similar.

SpecCommit Suppose the reduction uses the rule RED-SPEC-COMMIT.

$$\begin{array}{l}(\Gamma_r \mid C_m; \ldots; C_1 \mid \textbf{special}\,\textbf{commit}\ [i]\ f(h_1, \ldots, h_n)) \to \\ \quad (\Gamma_r \mid C_m; \ldots; C_{i+1}; C_{i-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n)) \\ \quad \textbf{when}\, i \in \{1 \ldots m\}\end{array}$$

The proof of typing must use the rule TY-SPECCOMMIT.

$$\frac{\Gamma_r \vdash i : \mathbb{Z}_{32}^{\textbf{signed}} \quad \Gamma_r \vdash \lceil h_i : t_i \rceil_1^n \quad \Gamma_r \vdash f : (t_1, \ldots, t_n) \to \textbf{void}}{\Gamma_r \vdash \textbf{commit}\ [i]\ f(h_1, \ldots, h_n) : \textbf{special}\,\textbf{void}}$$

From the premises $\Gamma_r \vdash \lceil h_i : t_i \rceil_1^n$ and $\Gamma_r \vdash f : (t_1, \ldots, t_n) \to \textbf{void}$, we can infer that $\Gamma_r \vdash f(h_1, \ldots, h_n) : \textbf{void}$. The argument for RED-SPEC-COMMIT-2 is similar.

3.3 Progress

Theorem 3.3.1 PROGRESS *If $(\Gamma_r \mid \mathcal{C}_r \mid e_r)$ is a program, and e_r is not a value h, then there is a program $(\Gamma_c \mid \mathcal{C}_c \mid e_c)$ such that $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \to (\Gamma_c \mid \mathcal{C}_c \mid e_c)$, or $(\Gamma_r \mid \mathcal{C}_r \mid e_r) \to$* **error**.

This can be proven by induction on the length of the proof $\Gamma_r \vdash e_r : t$. The proof for the full FIR language is given in the technical report [11]. This thesis gives a sketch of the proof for the special calls only, where $e_r = \textbf{special}\, S$.

SysMigrate Suppose $e_r = \textbf{special migrate}\ [j, h_p, h_o]\ h_{fun}(h_1, \ldots, h_n)$. The typing rule that applies is TY-SYSMIGRATE.

$$\frac{\begin{array}{cc} & \Gamma_r \vdash h_{ptr} : \textbf{data} \\ j \in \mathbb{Z}_{box} & \Gamma_r \vdash h_{off} : \mathbb{Z}_{32}^{\textbf{signed}} \\ \Gamma_r \vdash \lceil h_i : t_i \rceil_1^n & \Gamma_r \vdash h_f : (t_1, \ldots, t_n) \to \textbf{void} \end{array}}{\Gamma_r \vdash \textbf{migrate}\ [j, h_{ptr}, h_{off}]\ h_f(h_1, \ldots, h_n) : \textbf{special void}}$$

Since $h_f : (t_1, \ldots, t_n) \to \textbf{void}$, it must be defined in the context as a function $f = \lambda v_1, \ldots, v_n.e$. The RED-SYSMIGRATE rule applies.

$$(\Gamma_r \mid \mathcal{C}_r \mid \textbf{special migrate}\ [j, h_{ptr}, h_{off}]\ f(h_1, \ldots, h_n)) \to (\Gamma_r \mid \mathcal{C}_r \mid f(h_1, \ldots, h_n))$$

Spec Suppose $e_r = \textbf{special speculate}\ h_f(h_{const}, h_1, \ldots, h_n)$. The corresponding typing rule is TY-SPEC.

$$\frac{\Gamma_r \vdash h_{const} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \Gamma_r \vdash \lceil h_i : t_i \rceil_1^n \quad \Gamma_r \vdash h_f : (\mathbb{Z}_{32}^{\textbf{signed}}, t_1, \ldots, t_n) \to \textbf{void}}{\Gamma_r \vdash \textbf{speculate}\ h_f(h_{const}, h_1, \ldots, h_n) : \textbf{special void}}$$

Since $h_f : (h_{const}, h_1, \ldots, h_n) \to \textbf{void}$, it must be defined in the context as a function f. The RED-SPEC rule applies.

$$\begin{array}{l}(\Gamma_r \mid \mathcal{C}_r \mid \textbf{special speculate}\ f(h_{const}, h_1, \ldots, h_n)) \to \\ \quad (\Gamma_r \mid \langle \Gamma_r, f(\diamond, h_1, \ldots, h_n) \rangle ; \mathcal{C}_r \mid f(h_{const}, h_1, \ldots, h_n))\end{array}$$

SpecRollback Suppose $e_r = \textbf{special rollback}\ [h_{level}, h_{const}]$. The corresponding typing rule is TY-SPECROLLBACK.

$$\frac{\Gamma_r \vdash h_{level} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \Gamma_r \vdash h_{const} : \mathbb{Z}_{32}^{\textbf{signed}}}{\Gamma_r \vdash \textbf{rollback}\ [h_{level}, h_{const}] : \textbf{special void}}$$

The value h_{level} must defined in the context as a variable or a constant. In the former case, there is a rule in the full FIR, RED-ATOM-VAR, which will expand the variable. Otherwise, let $i = h_{level}$, and suppose the checkpoint context $\mathcal{C}_r$ contains m checkpoints $C_m, \ldots, C_1$.

If $1 \leq i \leq m$, then the RED-SPEC-ROLLBACK rule applies.

$$\begin{array}{l}(\Gamma'_r \mid C_m; \ldots; C_i = \langle \Gamma_r, f(\diamond, \lceil h_k \rceil_{k=1}^n) \rangle ; \ldots ; C_1 \mid \textbf{special rollback}\ [i, j]) \to \\ \quad (\Gamma_r \mid C_i; C_{i-1}; \ldots; C_1 \mid f(j, \lceil h_k \rceil_{k=1}^n)) \\ \quad \textbf{when}\ i \in \{1 \ldots m\}\end{array}$$

If $i = 0 \wedge m > 0$, then the RED-SPEC-ROLLBACK-2 rule applies.

$$\begin{array}{l}(\Gamma'_r \mid C_m = \langle \Gamma_r, f(\diamond, h_1, \ldots, h_n) \rangle ; \ldots ; C_1 \mid \textbf{special rollback}\ [0, j]) \to \\ \quad (\Gamma_r \mid C_m; \ldots; C_1 \mid f(j, h_1, \ldots, h_n))\end{array}$$

Otherwise, the RED-SPEC-ROLLBACK-ERROR rule applies.

$$
\begin{array}{l}
(\Gamma_r \mid C_m; \ldots; C_1 \mid \textbf{special rollback } [i, j]) \rightarrow \textbf{error} \\
\quad\quad \textbf{when } i \notin \{0 \ldots m\} \vee m = 0
\end{array}
$$

SpecCommit Suppose $e_r = \textbf{special commit } [h_{level}]\ h_f(h_1, \ldots, h_n)$. The typing rule which applies is TY-SPECCOMMIT.

$$
\frac{\Gamma_r \vdash h_{level} : \mathbb{Z}_{32}^{\textbf{signed}} \quad \Gamma_r \vdash \lceil h_i : t_i \rceil_1^n \quad \Gamma_r \vdash h_f : (t_1, \ldots, t_n) \rightarrow \textbf{void}}{\Gamma_r \vdash \textbf{commit } [h_{level}]\ h_f(h_1, \ldots, h_n) : \textbf{special void}}
$$

The value h_{level} must be a variable or a constant. In the former case, the RED-ATOM-VAR rule applies. Otherwise, let $i = h_{level}$, and suppose the checkpoint context $\mathcal{C}_r$ contains m checkpoints $C_m, \ldots, C_1$.

If $1 \leq i \leq m$, then the RED-SPEC-COMMIT rule applies.

$$
\begin{array}{l}
(\Gamma_r \mid C_m; \ldots; C_1 \mid \textbf{special commit } [i]\ f(h_1, \ldots, h_n)) \rightarrow \\
\quad\quad (\Gamma_r \mid C_m; \ldots; C_{i+1}; C_{i-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n)) \\
\quad\quad \textbf{when } i \in \{1 \ldots m\}
\end{array}
$$

If $i = 0 \wedge m > 0$, then the RED-SPEC-COMMIT-2 rule applies.

$$
\begin{array}{l}
(\Gamma_r \mid C_m; \ldots; C_1 \mid \textbf{special commit } [0]\ f(h_1, \ldots, h_n)) \rightarrow \\
\quad\quad (\Gamma_r \mid C_{m-1}; \ldots; C_1 \mid f(h_1, \ldots, h_n))
\end{array}
$$

Otherwise, the RED-SPEC-COMMIT-ERROR rule applies.

$$
\begin{array}{l}
(\Gamma_r \mid C_m; \ldots; C_1 \mid \textbf{special commit } [i]\ f(h_1, \ldots, h_n)) \rightarrow \textbf{error} \\
\quad\quad \textbf{when } i \notin \{0 \ldots m\} \vee m = 0
\end{array}
$$

3.4 Runtime safety checks

Preservation and progress require some runtime support to be complete, notably for subscripting operations. Array bounds checks are explicit in the reduction rules, but type validation for values read from unsafe blocks (blocks with type **data**) is deferred to the runtime. These checks are performed during the $\textbf{runtime}(\Gamma \mid \langle \text{C} \rangle\ [i] : t)$ and $\textbf{runtime}(\Gamma \mid \langle \text{C} \rangle\ [i] : t \leftarrow h)$ operations, which are described in more detail in Section 4.1.5.

Chapter 4

Implementation of Process Migration and Speculation

Previous chapters focus on the formal properties of process migration and speculation. This chapter focuses on the implementation details, particularly of the construction of a runtime environment that satisfies the operational semantics and safety properties imposed in previous chapters. Process migration and speculation interact closely with a compacting garbage collector, which is described in Chapter 5.

4.1 Runtime implementation

The FIR is machine-independent, and the Mojave compiler architecture is designed to support multiple back-ends, including both native-code and interpreted runtime environments. Currently, our primary runtime implementation is a native-code runtime for the Intel IA32 architecture [1]. An additional runtime environment is available that simulates RISC architectures. Object code generation is performed in two stages: the FIR is first translated to a machine intermediate representation (MIR), which introduces runtime safety checks in a machine-independent form. The MIR language is not discussed in detail here; the language itself is similar to the FIR with a simpler type system, and the process of generating MIR code is a straightforward elaboration of the FIR code. In the second stage, the final object code is generated for the target architecture from the MIR program.

The runtime manages several tasks, including garbage collection, process migration, speculation, and runtime type-checking for heap operations. To complicate matters, a faithful C pointer semantics rules out direct use of data relocation, which occurs when a process migrates or during heap compaction. Several auxiliary data structures and invariants are introduced to address these matters.

4.1.1 Runtime data structures and invariants

The runtime consists of the following parts and invariants.

- A *heap*, containing the data for tuples, arrays, and rawdata. A data value in the heap is called a *block*, and the heap contains multiple (possibly non-contiguous) blocks. Values are stored in the heap in an architecture-independent format.

- A *text area*, containing the program code. This includes both native machine code and a representation of the FIR code. The FIR code is immutable at all times, and the native machine code is immutable at all times except during process migration. The native code is modified during process migration, when the machine code is regenerated from the FIR for the target architecture.

- A set of *registers*. Each variable in the FIR program is assigned to a register, which is usually a hardware register. Some architectures have a limited number of hardware registers, so some FIR variables may be stored in memory locations known as *register spills*.

  At any time during program execution, a register may contain a value in one of several machine types: a pointer into the heap, a function pointer, or a numerical value. The machine type is statically determined from the variable's type in the FIR. Register spills have the same properties as registers, and additional *temporary registers* are introduced during assembly code generation which have very short live ranges. Registers may represent values in a machine-dependent manner.

  Invariant: at each function boundary and garbage collection point: if a register contains a pointer, it contains the address of the beginning of a block in the heap; if a register contains a function pointer, it contains the address of a function entry point in the text area.

  Note that temporary registers that are generated during assembly generation may contain "hybrid" pointers, however these temporaries are never live at a function boundary or garbage collection point.

- A *pointer table*, containing pointers to all valid data blocks in the heap.

  Invariant: all non-empty entries in the pointer table contain pointers to valid blocks in the heap, and every valid block in the heap has an entry allocated for it in the pointer table.

  It is possible that there will be valid blocks in the heap whose pointer table entry refers to a different block; Section 4.4 discusses this special case.

- A *function table*, containing function pointers to all valid higher-order functions. The function table is immutable, except during process migration.

Unsafe block format

MSB	LSB
size : 30 (words)	0 R
index : 30 (words)	1 M
$data_0$	
$\vdots$	
$data_{size-1}$	

Safe block format

MSB		LSB
tag : 7	*size* : 23 (words)	0 R
index : 30 (words)		0 M
$data_0$		
$\vdots$		
$data_{size-1}$		

Figure 4.1: Data block header format

Invariant: all entries in the function table contain the address of a function entry point in the text area.

- A *checkpoint* record, containing descriptions of all live program checkpoints. Checkpoints are discussed in Section 4.4.

4.1.2 Data blocks and the heap

The heap represents the FIR store, and it contains the store values b defined in Figure 2.5, also called *blocks*. The runtime representation of a block contains two parts: a header that describes the size and type of information stored in the block, and a value area containing the contents of the block.

There are two types of data blocks in the heap. Unsafe data corresponds to values of type **data**; memory accesses within a **data** block are unrestricted. Safe data corresponds to the $\langle t_1, \ldots, t_n \rangle$ and t **array** types. Memory operations on safe data blocks can be statically validated, to ensure that all values are manipulated with the proper types. The contents of unsafe data are not explicitly typed in the FIR, and safety checks are required to ensure the data is interpreted properly. Any pointer read from an unsafe block must be checked to ensure it is a valid pointer, and the bounds must be checked any time an unsafe block is dereferenced. In contrast, the contents of safe block data are typed in the FIR, and many safety checks can be omitted. Note that safety checks cannot be omitted on safe data after a successful migration, unless the two machines are mutually trusting. The garbage collector can use explicit FIR types to identify pointers embedded in safe block data, but it must use a more conservative algorithm to determine which values are pointers in unsafe block data. As a consequence, in the presence of unsafe data, the garbage collector may consider certain blocks to be live beyond their actual live range.

A block header has three parts, as illustrated in Figure 4.1: it contains a tag that identifies the block type (unsafe or safe), and distinguishes among safe data types; an index into the pointer table identifying the pointer for this block, accessed with the indexof(b) function; and a nonnegative number that indicates the size of the block, accessed with the sizeof(b) function. The header also

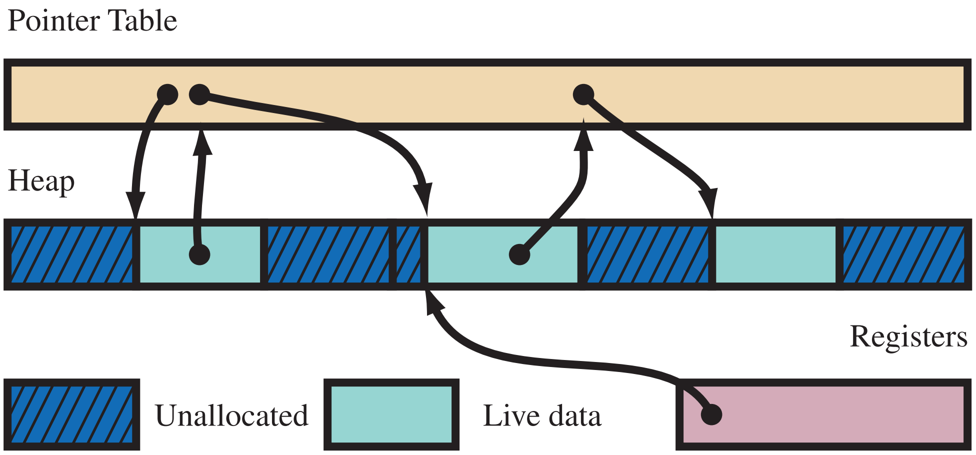


Figure 4.2: Pointer table representation

contains bits used by garbage collection and speculations. In the figure, R indicates the root bit, and M indicates the mark bit, both discussed in Chapter 5.

A null pointer is always a valid pointer to a zero-size block. A null pointer is allocated an entry in the pointer table and may be manipulated like any other base pointer, but any attempt to dereference it will result in a runtime exception. Certain source-level languages require distinct null pointers for various data types. For example, in Java there are different null pointers for each object type that are not considered equivalent. For these languages, a different zero-size block is allocated in advance for each type.

4.1.3 Pointer table

The pointer table *ptr_table* effectively acts as a segment table for the blocks (segments) in the heap. It supports several features, including migration and speculation, but its main purpose is to allow for relocation and safety for C data areas. The pointer table is implemented in software, however its design is compatible with a hardware implementation for increased efficiency.

Figure 4.2 illustrates the pointer table layout. The pointer table contains entries pointing to allocated data blocks. Source-level C pointers are represented in the runtime as ($base + offset$) pairs. The base pointer always points to the beginning of a data block in the heap. Base pointers are never stored directly in the heap. Instead, the base pointer is stored as an index to an entry in the pointer table, which contains the actual address of the beginning of the data block.

The pointer table serves several purposes. First, it provides a simple mechanism for identifying and validating data pointers in aggregate blocks. When an index i for a base pointer is read from the heap, the following steps are performed:

1. i is checked against the size of the pointer table to verify if it is a valid index.

MSB	LSB
arity_tag : 32	
size = 0 : 30 (words)	0 0
index : 30 (words)	0 0
function code ⋮	

Figure 4.3: Function header format

2. The value p is read from the i^{th} entry in the pointer table.
3. p is checked to ensure it is not free (that it points into the heap). This check tests the least-significant bit of p; empty pointer table entries are always odd values, whereas valid heap pointers are always even values.

After these steps, p is always a valid pointer to the beginning of a block with index i. These steps can be performed in a small number of assembly instructions, requiring only two branch points.

The second purpose of the pointer table is to support relocation. If the heap is reorganized by garbage collection or process migration, the pointer table and registers are updated with the new locations, but the heap values themselves are preserved. This level of transparency has a cost: in addition to the execution overhead, the header of each block in the heap contains an index. In the IA32 runtime, the overhead is in excess of 12 bytes per block, including the pointer table.

4.1.4 Function pointers

Function pointers are managed through the function table *fun_table*, which serves the same purpose as the pointer table for heap data. Any function whose address is taken and stored in a function pointer is considered an *escaping* function. A function stub must be generated for each FIR function that escapes. Each function stub has a function header, and the function table contains the addresses of all the escaping-function headers. To ease some of the runtime safety checks, each function stub is formatted with a block header that indicates that the function is a data block with zero-size. This prevents a function pointer from being errantly used as a data pointer using the existing safety checks. The function header is illustrated in Figure 4.3. As with block pointers, function pointers are represented in the heap as indexes into the function table.

The function header also contains an arity tag, used to describe the types of the arguments. The arity tags are integer identifiers, computed at link time from the function signatures. The signatures themselves are generated based on the primitive architecture types, not the high-level FIR types. When a function is called, the arguments must generate an arity tag that matches the function header, or the runtime raises an exception. This check must be performed at runtime since C permits function pointers to be coerced arbitrarily.

Functions may be invoked in a tail-call directly by specifying the function name, or indirectly through a function pointer. Unlike direct function calls, indirect calls through a function pointer use a standardized calling convention. The escape stub is responsible for moving arguments from standard argument registers to the registers expected by the original function.

4.1.5 Pointer safety

The runtime operations for load, $\mathbf{runtime}(\Gamma \mid \langle \text{C} \rangle\, [i] : t)$, and store, $\mathbf{runtime}(\Gamma \mid \langle \text{C} \rangle\, [i] : t \leftarrow h)$, are guaranteed to be type-safe, even for unsafe blocks. The typing rules require any data read from an unsafe block to be type-checked by the runtime before it is used. In MCC, these checks are performed during the load operation. The location where a value is loaded may not correspond to the location where the value is used. For C programs, this will result in error-handling semantics that are inconsistent with traditional C programs, where fault handling is performed at the time that the value is used. The runtime safety check for a load operation is performed as follows.

1. The index i is compared with the bounds of block $\langle \text{C} \rangle$; an exception is raised if the index is out-of-bounds.
2. The value h at location i is retrieved, and a safety check is performed.
   - If t represents a pointer, then h should be an index into the pointer table. If h is a valid pointer table index, and the entry $ptr\_table[h]$ is a valid pointer p, the result of the load is p.
   - If t represents a function pointer, then h should be an index into the function table. If h is a value function table index, the result of the load is $fun\_table[h]$.
   - Otherwise, h does not represent a pointer, and the result of the load is h.

The safety check for a store operation is somewhat simpler. For the $\mathbf{runtime}(\Gamma \mid \langle \text{C} \rangle\, [i] : t \leftarrow h)$ operation, the runtime invariants guarantee that if t represents a pointer, then h is a valid pointer to a block in the heap; if t is of function type, then h is a valid pointer to a function header. In these two cases (after a bounds-check on the index i) the index for h is stored. If t does not represent either kind of pointer, the value h is stored directly.

4.2 Compiling FIR to assembly code

Compilation of FIR to assembly code is done through the MIR. The MIR resembles the FIR, however the type system in the MIR is much weaker and closely resembles the machine architecture types. To simplify conversion to assembly, details about polymorphic types are removed, and pointers to various blocks and high-level functions are reduced to three pointer types: safe blocks, unsafe blocks, and functions. The system migration and speculation calls are also expanded into operations that are simpler to convert to assembly code. The final assembly code is generated through the use of a special term-rewriting language known as Kupo [17]; various assembly-level optimizations including dead instruction elimination, peephole optimization, and instruction scheduling are expressed in Kupo.

Throughout the conversion from FIR to assembly code, it is important to maintain a correlation between a statement in the FIR and an instruction in the assembly code, for system migration. This is because migration recompiles the code from the FIR on the target machine, and must know where to begin execution once the code is recompiled. Currently, an exact correlation is maintained between the FIR and the assembly code for each **migrate** $[i, a_{ptr}, a_{\mathit{off}}]\ f(a_1, \ldots, a_m)$ statement.

The migration points must be explicit in the program. It is feasible to introduce conditional migration points at the beginning of each function call that will check if the process should migrate, and do so as needed. The FIR contains no explicit looping construct, so any loop must be expressed using tail-recursion, and every function must terminate. Therefore, function boundaries are encountered on a regular basis, making it reasonable to establish cooperative migration points at those locations.

To preserve the correlation between FIR instructions and assembly code, backend optimizations cannot reschedule code across function boundaries that may be the target of a migration. Currently, most advanced optimizations are performed on the FIR code where they can be formally verified using MetaPRL [12], an integrated theorem prover. It is difficult to reason about the MIR due to the weakened type system and semantics, so optimizations in the backend are limited to architecture-specific optimizations. Migration is safe because backend optimizations are limited in scope, and do not reschedule instructions across function boundaries.

4.3 Process migration

To facilitate fault-tolerant computing, MCC introduces a level of abstraction between the processes that are running in a distributed system, and the specific machines on which they are running. Each process should view the distributed cluster as a single machine and a single shared resource pool, rather than a collection of distinct nodes each with their own set of resources. Processes should have a lifetime greater than that of any particular machine in the cluster, so that the process continues running even when one or many machines it is running on fail. Processes should also be able to recover automatically from unanticipated and abrupt failures.

In a distributed system, a process will execute on a specific machine with a particular architecture. Since individual nodes in a cluster may fail at any time, a mechanism for migrating a process from one machine to another is an essential tool for fault-tolerance. Such a mechanism needs to perform three operations: a **pack** operation to capture the entire state of the process, including the program counter, register values, heap data, and code; a *transmit* operation to transmit the state of the process to a target machine; and an **unpack** operation to reconstruct the process state on the target machine and resume execution. Collectively, this sequence of operations is referred to as *process migration*.

Process migration should be architecture-independent to allow for distributed clusters of heterogeneous nodes. Also, process migration should be safe; the remote machine receiving the program should be able to verify that the program type-checks and that heap values are used in a proper manner. If the remote machine can verify that a received program is safe, then process migration is viable in environments where machines in the cluster do not trust each other entirely, such as the wide-area computing clusters on the Internet.

Since process migration requires **pack** and **unpack** operations, it is fairly straightforward to extend the mechanism to support saving the process state to a file for later execution, and to write checkpoint files while the process is running that contain snapshots of the full process state. In the event of a later failure, the process can be recovered from this file using the **unpack** operation.

This section provides an overview of the system migration implementation; technical details are further discussed in the MCC documentation [18].

4.3.1 Using process migration in the FIR

In the FIR, migration is expressed using the special-call mechanism. Special-calls are similar to tail-calls, taking a function name and argument list, but a special-call implies some special action is performed before the function is invoked. Since many FIR transformations are not concerned with the special action, these transformations treat a special-call exactly as a tail-call, simplifying their implementation.

Process migration is expressed in the FIR using the **migrate** $[i, a_{ptr}, a_{off}]\ f(a_1, \ldots, a_n)$ special-call. The first three arguments indicate how the migration should be performed, and are not passed as arguments to f. The integer i represents a unique label that identifies the migration call, and is used by the backend to determine where program execution resumes after a successful migration. The pointer (a_{ptr}, a_{off}) refers to a null-terminated string that determines the migration target. The string includes information on what protocol to use.

There are three protocols that may be used for process migration. The first is the `migrate` protocol, which sends the entire state of a process to another machine, and terminates the process on the original machine. If migration fails for any reason, the process will continue to execute on the original machine. While a process may indirectly observe the result of a migration by invoking external functions, the process is indifferent to the machine it is running on, and does not observe a successful migration. This encourages an abstraction between the process and the machine it is running on, and enables processes to be migrated without their specific knowledge for failure-recovery or load-balancing purposes. The fact that the process does not observe the result of migration is reflected in the operational semantics.

In order to migrate to another machine, the remote machine must run a migration server. This is a version of the compiler that will listen for incoming migration requests, recompile any inbound processes on the new machine, and reconstruct their state before executing them.

The other two protocols write the process state to a file for later execution. The `suspend` protocol writes the process state to a file, and terminates the process if it is successfully written. In contrast, the `checkpoint` protocol continues running the process even when the file is successfully written. The latter protocol is useful for taking snapshots of a process state, as a crude rollback mechanism or in anticipation of possible machine failure in the near future. A resurrection program is used to resume process execution from a state file.

4.3.2 Runtime support for migration

Process migration requires two key features from the runtime. First, to support the **pack** operation the runtime must be able to collect the entire process state. For **unpack** it must be able to restore the process state from a previous **pack** operation. Second, the runtime should accommodate migration in an architecture-independent manner.

4.3.2.1 The pack and unpack operations

The implementation of the **pack** and **unpack** operations is relatively straightforward. Since all heap data and function pointers in the heap are represented indirectly as indices, the heap data is not modified by a migration, even if the data are relocated. Also, by imposing standard byte ordering and alignment rules on heap data, the amount of translation required to migrate the heap across architectures is minimal. This is essential for unsafe languages such as C, where it is difficult or impossible to determine whether data in the heap needs to be realigned or byte-swapped. For example, an array of characters is indistinguishable from an array of 32-bit integers in languages that do not feature strong typing, defeating attempts to automatically align and byte-swap data for the native architecture.

The **pack** operation first performs garbage collection on the heap. Then it packs the live data, the pointer table, the program text, and the registers into a message that can be stored or transmitted. To migrate the register spills and hardware registers (which together cover the set of variables in the FIR program), MCC stores the set of live variables into a newly allocated block *migrate_env* on the heap, taking care to convert any real pointers into index values. The set of live variables across migration corresponds exactly to the arguments $(a_1, \ldots, a_n)$ passed to function f. By storing the variables into a block in the heap, several problems are addressed:

- All data is stored in the heap at the time of migration, with the exception of a single variable that contains the index for *migrate_env*.

- Since no data is stored in variables, no data will be stored in the hardware-specific registers. Therefore system migration does not need to construct an explicit map between register names on different architectures.

- Since all data is in the heap during migration, all data follows the standard, architecture-independent representation for MCC. Data in hardware registers may continue to have hardware-specific representations without interfering with system migration. Also, since no real pointers exist in the data, system migration does not need to construct an explicit map between pointers across different machines.

- Safety checks can be applied during the **unpack** operation on all data, including data in variables. This is performed as usual, when the data is read from the heap.

There is a performance issue related to storing all register data for a process in the heap. It is reasonable to store registers in the heap when the process is actually migrating elsewhere, since the cost of transmitting the data over the network generally outweighs any local processing. However, for the checkpoint protocol it means that all register data must be stored in the heap and then immediately loaded back into registers, with safety checks. This operation can be fairly expensive if the process intends to take frequent checkpoints. Checkpointing can be partially optimized by not attempting to unpack the registers when the process continues execution on the current machine. While the **unpack** operation and associated safety checks can be skipped in this case, the **pack** operation cannot be omitted.

On an **unpack** operation, the FIR code is type-checked, recompiled, and execution is resumed. Register values are extracted from the heap with the standard safety checks, allowing the register values to be type-checked.

4.3.2.2 The migrate operation

To implement the **migrate** operation, the source machine first transmits the following data to the server:

1. FIR code for the process
2. size of heap and pointer tables
3. index of the block containing live variables (*migrate_env*)
4. location to resume execution at (i).

The server compiles the code and links it with a special stub that initializes the heap, restores the registers and resumes execution at the location indicated by i. If this compilation is successful, then the server starts the new process using the stub, and the source machine transmits the contents of the pointer table and heap to the new process, allowing the heap to be reconstructed.

In order to achieve architecture independence, MCC never migrates the actual executable text. Instead it migrates the FIR code for the program, so the target machine can formally verify the safety of the code. The location index i in the migration call is used to correlate the runtime execution point with a corresponding execution point in the FIR.

Since all pointers are stored in the heap using indexes, migration must be careful that indexes on the destination machine *exactly* match. This is not a problem for data pointers since the pointer table is migrated as-is. The order of program globals and the function table is determined at compile time, therefore the compiler on the target machine must be aware of the original order of these symbols so the tables can be reconstructed in the proper order.

4.4 Speculative operations

Semantically, speculative execution appears atomic; that is, either all the operations in a speculation must succeed, or none of them will succeed. The FIR provides a generalization of speculation for expressing rollback of a distributed computation that is more efficient than using process migration alone in the event of a machine failure.

The primary obstacle in implementing speculation is restoration of the program state. This chapter does not include rollback of I/O operations, however the principles discussed here are being applied for the Mojave group's distributed filesystem [8]. When a speculation is aborted, the entire process state, including all variable and heap values, must be restored to the state it had on entry into the speculation. This rollback operation can be expressed with process migration by having a process write a checkpoint file each time it enters a new speculation. To abort the speculation, the previous state is restored from the checkpoint file. However, since the migration mechanism recompiles the program, and the *entire* process state must be reconstructed, this operation can be very expensive. Even taking the checkpoint is expensive, since the entire state must be written to a file, even parts of the state that have not changed since a prior checkpoint. By contrast, speculation uses a copy-on-write (COW) mechanism to keep track of modified state that must be restored if a speculation is rolled back, and speculation does not need to recompile the code.

The FIR provides three primitives for managing speculations: **entry**, which enters a new speculation level; **commit**, which marks a speculation level as completed; and **rollback**, which aborts all changes made by a level and resumes execution at the point where the level was previously entered.

4.4.1 Using speculations in the FIR

Like process migration, speculation uses the special-call mechanism in the FIR. Each **entry** operation enters a new speculation level nested within the previous level. Speculation levels are numbered from 1 to N, where 1 is the oldest speculation level entered and N is the most recent. A process that has not entered any speculation is at level 0. A level l keeps track of all changes made to the state that have occurred since l was entered. Speculation levels use copy-on-write (COW) semantics; when a block in the heap is modified, the block is cloned and the pointer table updated to point to the new copy of the block, preserving the data in the original block. On a **commit** or **rollback** operation of l, exactly one of these blocks will be discarded.

The primitive for **entry** is the **speculate** $f(c, a_1, \ldots, a_n)$ special-call. c is an integer that is passed as the first argument to f. On a rollback, the value of c passed to f may be changed to indicate that the rollback occurred. This is the only way to carry state information across a rollback[1].

The primitive for **commit** is **commit** $[l]$ $f(a_1, \ldots, a_n)$. This commits data for l by folding all changes from that level into its previous level. The level l must be in the interval $\{0 \ldots N\}$, otherwise a runtime exception will occur. If $l = 0$, then the most recent level N is committed.

The primitive for **rollback** is **rollback** $[l, c]$. This reverts all changes made by in level l and *all later levels*. l must be in the interval $\{0 \ldots N\}$, otherwise a runtime exception will occur. If $l = 0$, then the most recent level N is reverted.

Rollback resumes execution at the point where level l was entered. No function or argument list is specified. The function that was associated with level l is saved as part of the checkpoint, to be called with the original atom arguments but with the new value for c. This version of the primitive is a *retry* primitive. Speculation level l is automatically re-entered after it (and all later levels) have been rolled back. In effect, the state that is captured and restored is the state immediately after level l was entered.

4.4.2 Speculation state

Speculation requires the introduction of several variables, described below and summarized in Figure 4.4. The heap layout is illustrated in Figure 4.5, which is drawn with the base of the heap at the bottom of the figure, and the limit of the heap at the top. Some parameters in the figure are specific to garbage collector, which is discussed in Chapter 5.

Speculation uses the following state variables:

- The program is currently at speculation level *spec_next*. If *spec_next* $= 0$, then there are no speculations active.

[1] For technical reasons, c must be an integer in the current implementation.

Variable	Properties	Description
$spec\_next$	$spec\_next \geq 0$	Current speculation level
$base[l]$	$l \in \{0..spec\_next\}$	Base of heap corresponding to level i
$limit[l]$	$l \in \{0..spec\_next\}$	Limit of heap corresponding to level i
$ptr\_diff[l]$	$l \in \{0..spec\_next - 1\}$	Pointer differential tables
$spec\_env[l]$	$l \in \{0..spec\_next - 1\}$	Environment (registers for a speculation)
$limit$	$limit = limit[spec\_next]$	Upper bound of the heap
$current$	$current \geq base[spec\_next]$	Current allocation point

Figure 4.4: Speculation variables

- Each speculation level $l \in \{0..spec\_next\}$ is delimited in the heap by a base pointer $base[l]$ and a limit pointer $limit[l]$. The limit pointer is determined by the base of the next-youngest generation:

$$limit[l] = \begin{cases} base[l+1] & \text{if } l < spec\_next \\ limit & \text{if } l = spec\_next \end{cases}$$

  In MCC, each generation corresponds to a speculation level. The following inequalities hold:

$$base[l-1] \leq base[l] \quad \forall\, l \in \{1..spec\_next\}$$

$$base[spec\_next] \leq limit$$

- In addition to the $base[l]$ and $limit[l]$ bounds, each level except the youngest has a pointer difference table $ptr\_diff[l]$, which is used to restore the pointer table if speculation level $l+1$ is aborted. To minimize storage requirements, the $ptr\_diff[l]$ table is stored as a set of differences with the current pointer table ptr_table. When a block is copied due to a copy-on-write fault, the original block is added to the difference table $ptr\_diff[spec\_next - 1]$.

- Each level except the youngest has an environment $spec\_env[l]$ that contains the live variables (machine registers and register spills) on entry to level $l{+}1$. The environment is constructed and accessed using the **pack** and **unpack** operations, previously introduced for system migration in Section 4.3.2.1. Since the environment contains the live variables of the program, all data that was live on entry to level $l+1$ is reachable from the block $spec\_env[l]$. The environment block $spec\_env[l]$ is always allocated as the last block within speculation level l.

- The current allocation point is in $current$. The following holds:

$$base[spec\_next] \leq current \leq limit$$

4.4.3 Speculation properties and invariants

It is useful to define a few properties for blocks in the heap:

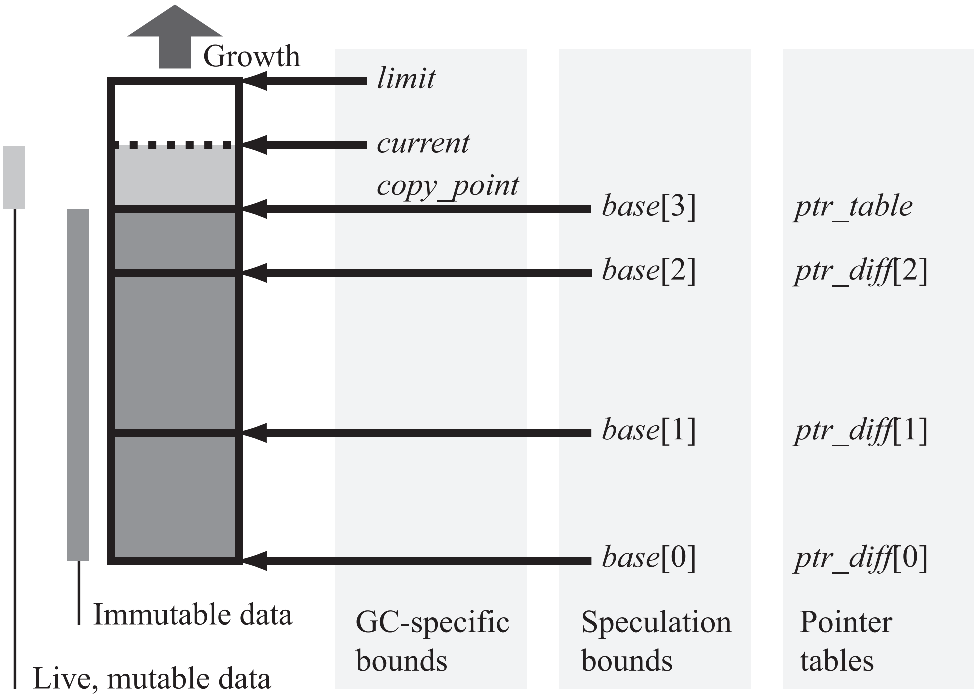


Figure 4.5: Heap data with multiple speculation levels

- A block b is *indexed* if the block is listed in *ptr_table* or *ptr_diff*. Blocks that do not appear in any pointer table are always available for collection. These non-indexed blocks are introduced by the **commit** and **rollback** operations.

- For blocks b and b' that are both indexed, block b' is a *version* of block b if $\text{indexof}(b) = \text{indexof}(b')$.

- Block b' is the *parent* of block b if b' is the largest block such that $b' < b$ and b' is a version of b. In this situation, b is the *child* of b'. This relationship occurs when a copy-on-write fault on b' creates block b.

- A block is *live* if it is reachable directly from the current live variables of a program, or if it is reachable from any indexed environment block *spec_env*$[l]$ when the pointer table is restored to the state it was in on entry to level $l + 1$. This is a definition of liveness that is typical for garbage-collected heaps, adapted to accommodate speculations. Any block in the heap that is not live, including non-indexed blocks, is considered *dead*.

With these definitions, we can define several invariants on speculations. These invariants are preserved in all three speculative operations, and also by the garbage collector.

4.4.3.1 Invariants related to the organization of the heap

The first invariant ensures that all data needed to revert to a given speculation level is contained within that speculation level or older levels.

Invariant: (Speculation Invariant) all heap data for speculation level l is between $base[0]$ and $limit[l]$. The heap data is immutable if $l < spec\_next$.

The next invariant ensures that at most one version of a block exists within a particular speculation level. The **commit** and **rollback** operations may introduce additional blocks in the speculation level that share a common index, but those blocks will not be indexed. Note that this invariant ensures that a block has at most one parent and at most one child.

Invariant: (Speculation Level Uniqueness) for a given index i and speculation level l, there is at most one indexed block b in level l with $\text{indexof}(b) = i$.

4.4.3.2 Invariants related to pointer difference tables

The next two invariants ensure that the difference tables remain consistent. When a block is listed in the difference table $ptr\_diff[l]$, the block should belong to speculation level l or an older level, and there should be a child residing in speculation level $l + 1$ that was created by a copy-on-write fault in level $l + 1$. It is possible no child exists in the case where the child was not live during garbage collection, in which case *ptr_table* is automatically reverted to the original block.

Invariant: (Difference Table Invariant 1) for every block $b \in ptr\_diff[l]$, $b < limit[l]$.

Invariant: (Difference Table Invariant 2) for every block $b \in ptr\_diff[l]$, either there exists a child block b' in speculation level $l + 1$, or $ptr\_table[\text{indexof}(b)] = b$.

4.4.3.3 Liveness invariant

If a block is live entering speculation l, then it must be preserved as long as l exists in the event that l is rolled back. Since it will be included in $spec\_env[l - 1]$, it will remain live as long as level l exists.

Invariant: (Speculation Liveness) if a block b is live on entry to speculation l, then it must remain live until level l is either committed or rolled back.

4.4.4 Implementation of speculations

Speculations are implemented in close cooperation with the garbage collector. Speculation data for each level is maintained in contiguous sections of the heap, and is ordered from oldest to youngest level to simplify the test for immutability. The garbage collector must maintain the invariants related to speculations.

4.4.4.1 The entry operation

On speculation **entry**, a new generation is set up in the heap by creating a new level $l = spec\_next+1$. The live variables of the program are packed into a new indexed block $spec\_env[l-1]$, allowing the live variables to be recovered if the speculation is later aborted. Then, *current* is advanced to point past the end of $spec\_env[l-1]$. The heap is partitioned, with $base[l]$ pointing at *current*, the end of the allocated heap. All live data before *current*, including $spec\_env[l-1]$, becomes immutable. An empty difference table $ptr\_diff[l-1]$ is created; each copy-on-write fault in level l will add a pointer to this table. Finally, the *spec_next* variable is incremented.

Speculation Invariant is preserved since all data in the program is before $base[i]$. Speculation Level Uniqueness is also preserved, since $spec\_env[l-1]$ is allocated a new index and no other blocks are introduced. Difference Table Invariant 1 and Difference Table Invariant 2 both trivially hold since the new difference table is empty. Speculation Liveness holds because no blocks become dead during this operation.

4.4.4.2 Copy-on-write faults

Within a speculation, the only operations requiring special support are the assignment operations. If a block is to be mutated, and the block belongs to a previous speculation level (its address is below $base[spec\_next]$), the block is copied into the youngest generation (level *spec_next*), the current pointer table *ptr_table* is updated with the new location, and the previous pointer difference table $ptr\_diff[spec\_next-1]$ is updated with the block's original location. The original data remains unmodified. The garbage collector uses the *ptr_diff* tables as "root" pointers, to ensure that the original block remains live.

Note that a copy-on-write fault does not affect the liveness of the original block. Since the original block was live on entry to the current level, it is referred to by $spec\_env[spec\_next-1]$, therefore by definition it is still live, and Speculation Liveness is preserved. The pointer table always refers to the most recent live version of a block, therefore there was no indexed version of the block in the current speculation level prior to the copy-on-write fault, and Speculation Level Uniqueness holds. The invariants Difference Table Invariant 1 and Difference Table Invariant 2 remain true by construction. Speculation Liveness holds because no blocks become dead during this operation.

4.4.4.3 The commit operation

When a speculation level l is committed, blocks b which belong to level $l-1$ can be discarded if a newer block with the same index is live in level l. There are two cases to consider for **commit** operations:

- When $l = 1$, *ptr_diff* $[l-1]$ is discarded — the original copies of blocks which faulted at level 1 is reclaimed automatically during the next garbage collection.

- When $l > 1$, the difference table *ptr_diff* $[l-1]$ must be consolidated with the next-oldest difference table *ptr_diff* $[l-2]$ to preserve the pointer table history for blocks which had a copy-on-write fault in level l, but whose previous version is not in level $l-1$. For each block $b \in$ *ptr_diff* $[l-1]$, b is added to *ptr_diff* $[l-2]$ iff $b <$ *base* $[l-1]$. This allows the garbage collector to collect blocks that were preserved for a potential rollback of level l, but preserves the pointer table history for earlier levels.

By coalescing the difference tables, we maintain the invariants Difference Table Invariant 1 and Difference Table Invariant 2. Any block b in level $l-1$ that has a child in level l will no longer be indexed, because *ptr_table* will refer to a descendant of b, and *ptr_diff* $[l-1]$ (the only other table which could refer to b) will be deleted; therefore Speculation Level Uniqueness also holds. Only blocks that were in level $l-1$ may become dead during this operation, therefore Speculation Liveness holds.

Once the difference tables are coalesced, the limit of the previous level is adjusted, *limit* $[l-1] :=$ *limit* $[l]$, preserving Speculation Invariant. The base and limit pointers for level l are deleted, and all younger levels are shifted down. The environment block *spec_env* $[l-1]$ is no longer indexed. At the end of the operation, *spec_next* is decremented.

4.4.4.4 The rollback operation

The **rollback** of speculation level l assumes that all levels after l have already been rolled back, and *spec_next* $= l$. The pointer table is restored from the current pointer table and the *ptr_diff* $[l-1]$ table as follows: for each $b \in$ *ptr_diff* $[l-1]$, set *ptr_table* $[\text{indexof}(b)] := b$. The difference table *ptr_diff* $[l-1]$ is then discarded. Difference Table Invariant 1 and Difference Table Invariant 2 are preserved since *ptr_diff* $[l-1]$ is deleted and no other difference table is affected by this rollback.

The limit of the previous level is adjusted, *limit* $[l-1] :=$ *limit* $[l]$, preserving Speculation Invariant. The base and limit pointers for level l are deleted, and the environment block *spec_env* $[l-1]$ is no longer indexed. No block allocated in level l will be indexed after **rollback**, so Speculation Level Uniqueness is preserved. Furthermore, only blocks that were in level l may become dead during this operation and there are no younger speculation levels, therefore Speculation Liveness trivially holds. At the end of the operation, *spec_next* is decremented.

Chapter 5

Garbage collection

The garbage collector implements generational, mark-sweep, compacting collection. It incorporates two phases: a minor collection phase that is fast and eliminates blocks with short live ranges, and a major collection phase that sweeps and compacts the entire heap. Use of a compacting collector is possible through the use of the pointer table, and is beneficial since it preserves temporal data locality. Two blocks that are allocated near each other temporally are more likely to be used together than two blocks that were allocated far apart from each other. By preserving temporal locality, we increase the likelihood that frequently-accessed data will be close together in memory, thereby improving the cache performance over breadth-first copying collectors.

The garbage collector maintains a number of heap invariants that are required for efficient implementation of speculations. This section describes the necessary heap invariants and presents an outline of the garbage collection algorithm, and discusses how the invariants are maintained by the algorithm.

5.1 Heap and pointer table properties

For speculations to function efficiently, the following invariants are imposed on the heap:

- The heap is a single contiguous span of memory. The heap is partitioned into $\mathit{spec\_next} + 1$ contiguous segments, each representing one speculation level l with bounds $\mathit{base}[l]$, $\mathit{limit}[l]$.

- The segments are ordered from the oldest speculation level to the newest. The interval $[\mathit{base}[0], \mathit{limit}[0])$ corresponds to data at level 0 (data allocated outside of any speculation), $[\mathit{base}[1], \mathit{limit}[1])$ corresponds to data allocated after the first **speculate** $f(c, a_1, \ldots, a_m)$ call, and so forth.

- New data can only be allocated in the youngest speculation level, within the interval $[\mathit{base}[\mathit{spec\_next}], \mathit{limit}[\mathit{spec\_next}])$. Copy-on-write faults on immutable blocks also generate new blocks within this interval.

Variable	Properties	Description
$base[minor]$	$base[minor] \geq base[spec\_next]$	Base of minor heap (within youngest level)
$limit[minor]$	$limit[minor] \leq limit$	Limit of minor heap
$minor\_roots$	none	Blocks that contain pointers into *minor*
$copy\_point$	$copy\_point = base[spec\_next]$	Marks end of immutable segment of heap
gc_level	$gc\_level \in \{0..spec\_next\}$	Indicates which immutable levels were collected

Figure 5.1: GC variables

- Data in level l that is dead on entry to level $l + 1$ may be removed by the garbage collector at any time, but data that is live on entry to $l + 1$ must remain live until level $l + 1$ is either committed or rolled back, to preserve SPECULATION LIVENESS.
- All live data in speculation level l', $l' < spec\_next$, is immutable.
- Garbage collection does not reorder the live blocks in the heap.

Several properties are also imposed on the pointer table. In general, the pointer table refers to the most recent live version of a block at the end of each garbage collection. If a block b with index i is live, then:

1. $ptr\_table[i] \neq$ **empty**
2. $ptr\_table[i] = b$ iff b has no live child block.
3. $ptr\_table[i] > b$ iff b has a live child block.

Maintaining the pointer table correctly requires special attention in the case where block b is live entering a new speculation and subsequently generates a new block b' through a copy-on-write fault. If b' subsequently becomes dead during the course of the current speculation, then the pointer table must be reverted to point to b.

5.1.1 Garbage collector state

In addition to the speculation state summarized in Figure 4.4 and illustrated in Figure 4.5, the garbage collector maintains several additional variables, listed in Figure 5.1 and described below.

The youngest speculation level $spec\_next$ contains a minor heap, with a base pointer $base[minor]$ and a limit pointer $limit[minor]$. All blocks outside the minor heap that may contain pointers into the minor heap must be listed in the set of root blocks $minor\_roots$. This allows the minor heap to be collected in isolation, reducing the overall expense of the garbage collector. The minor heap must be a subinterval of $[base[spec\_next], limit)$. It is permissible for $limit[minor]$ to be less than the heap limit $limit$. We usually want the minor heap to remain small, to improve cache performance in

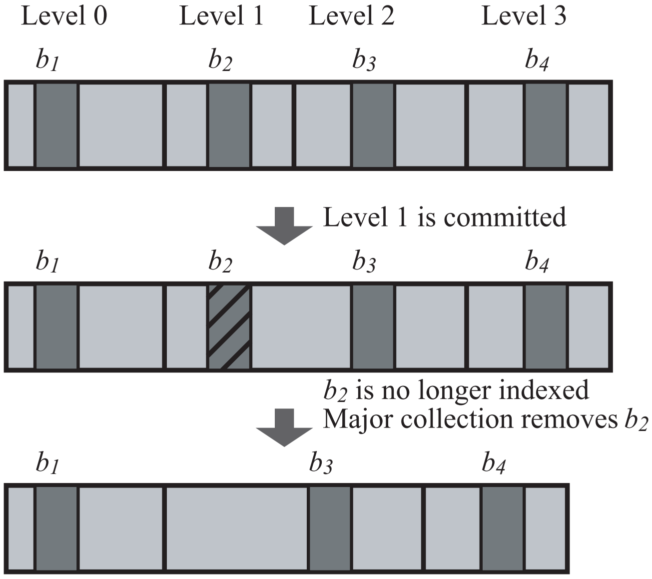


Figure 5.2: GC properties in presence of speculative operations

the system. *minor_roots* is maintained by the same mechanism that copy-on-write faults use, and is maintained conservatively.

copy_point is used to indicate the location of the end of the immutable heap segment. All speculation levels except the youngest level are immutable. Since the levels are ordered in the heap, a copy-on-write test for block b can compare against *copy_point* to determine if the block is immutable.

When the process enters speculation level $l + 1$, any data in l that is live on entry remains live throughout level $l + 1$. As a result, the garbage collector only needs to collect level l once to identify the data that was dead on entry to level $l + 1$. The *gc_level* variable is used to track which speculation levels have already been collected and can be disregarded in subsequent major collections. This optimization improves performance for processes that enter a large number of speculations and programs that have long-life speculations. *gc_level* always refers to the first level that has not been fully collected.

5.1.2 Garbage collection and speculations

Typically, garbage collection on multiple versions of a block marks at most one version dead. Consider a sequence of blocks $b_1 < b_2 < \cdots < b_m$, shown in Figure 5.2, that are in the heap at a garbage collection such that b_i is a parent of b_{i+1}. Such a sequence is generated by copy-on-write faults on $b_1, \ldots, b_{m-1}$. By Speculation Level Uniqueness and Speculation Liveness, the first $m - 1$ blocks must remain live on a garbage collection. As a result, b_i *must* be live on garbage collection for $i < m$, and only b_m may be marked dead. The pointer table entry *ptr_table*[indexof(b_i)] for these

— *Perform GC. The live registers are in registers (a mutable set).*
function GC(*registers*, *request_bytes*, *request_pointers*):
  if *request_bytes* $\ll$ sizeof(*minor*) **then**
    Minor-GC(*registers*)
  if *request_bytes* $\geq$ free(*minor*) $\vee$ *request_pointers* $\geq$ free(*ptr_table*) **then**
    Major-GC(*registers*, *request_bytes*, *request_pointers*)

Figure 5.3: GC main function

blocks can be reclaimed only if $m = 1$.

Note that blocks which become non-indexed during a **commit** or **rollback** operation are never listed in any pointer table, and are always collected by the garbage collector. The **commit** and **rollback** operations allow multiple blocks $b_1, \ldots, b_m$ within a speculation level to have the same index, however at most one of the blocks may appear in any pointer table, and therefore at most one of the blocks can be live. After a major garbage collection, for every block b, no block b' exists which is in the same speculation level and has the same index. This stronger property implies Speculation Level Uniqueness.

Speculations are responsible for maintaining the garbage collector state in a manner such that the properties stated above remain valid. On **entry**, the minor heap is reset, *base*[*minor*] := *current*, and the minor roots in *minor_roots* are cleared. Also, the immutable heap limit is updated, *copy_point* := *current*.

On **commit** and **rollback** of level l, level $l-1$ may contain dead data, so *gc_level* must be rolled back such that *gc_level* $\leq l-1$. *copy_point* also needs to be adjusted so that it reflects the base of the current speculation level, *base*[*spec_next*]. Neither operation needs to modify the minor heap or the minor roots, since both are conservatively maintained and will still satisfy the properties.

5.2 Garbage collector main algorithm

The main GC function is shown in Figure 5.3. The function is passed the number of bytes required for allocation in *request_bytes*, and the number of new pointer table entries required in *request_pointers*. The set of live variables (both hardware registers and register spills) which may contain pointers is passed in as the *registers* list. The main function determines whether a minor collection is sufficient based on a simple heuristic, and performs a major collection if necessary.

5.3 Mark operation

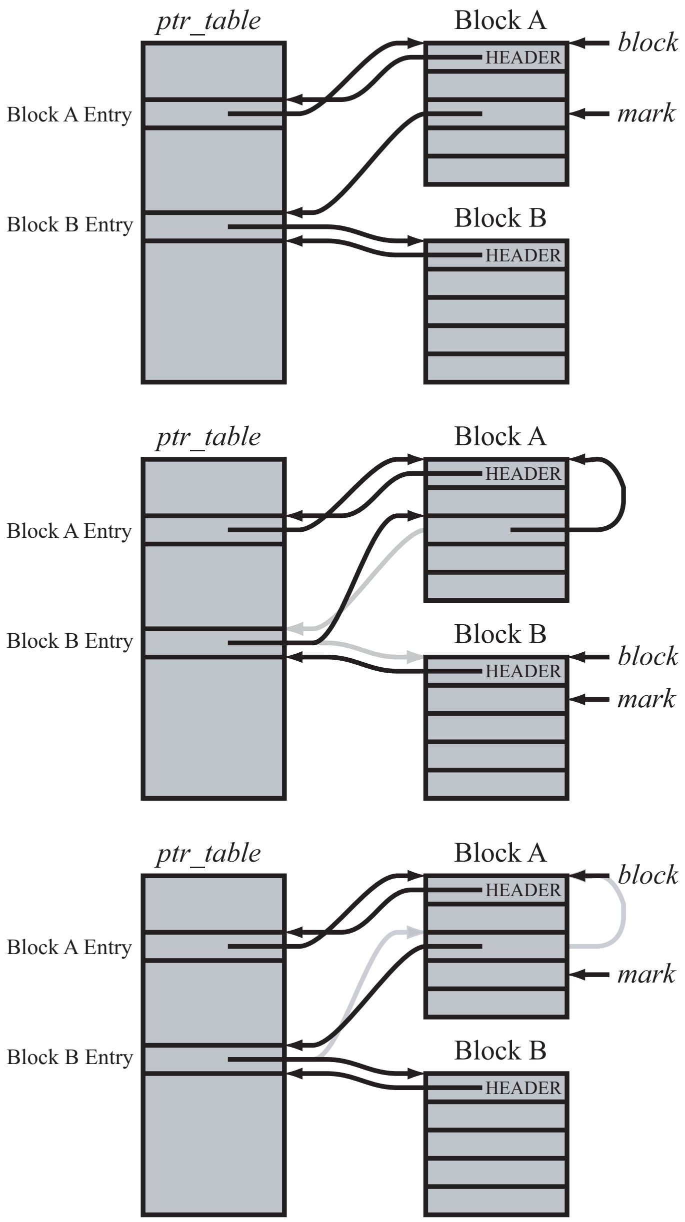
Both major and minor collection include a mark phase that marks live blocks within a particular interval *interval*, which may be a speculation level or the minor heap. A traditional marking algo-

```
function Mark-Contents(block, interval):
    size := sizeof(block), mark := 0, level := 0

label scanner:
    while mark < size do     — Iterate through fields in block.
        index := block[mark]
        if index is a valid index then
            field := ptr_table[index]
            if field ≠ empty ∧ field even ∧ field not marked ∧ field ∈ interval then
                mark field     — Discovered a new block; start marking field instead.
                ptr_table[index] := addressof(block[mark]) + 1
                block[mark] := block
                block := field
                size := sizeof(field), mark := 0, level := level + 1
                goto scanner
        mark := mark + 1

    if level > 0 then     — Backtrack a level.
        index := indexof(block)
        field := ptr_table[index] − 1
        ptr_table[index] := block
        block := field[0]
        mark := (field − block)/sizeof(field)
        block[mark] := index
        size := sizeof(block), mark := mark + 1, level := level − 1
        goto scanner
    return
```

Figure 5.4: Mark operation in garbage collector

rithm iterates over each field in a block b: for each unmarked block b' referenced, it marks b' and adds it to a queue for later processing. This queue requires storage that is linear in the number of blocks in the heap. MCC's marking algorithm uses a pointer reversal-scheme to eliminate the need for additional storage during traversal. It exploits the redundant encoding in the block headers and the pointer table to deliver constant storage requirements during the mark phase.

The algorithm described here is presented in the function Mark-Contents in Figure 5.4. It is not a recursive algorithm; instead, it implements a state machine that traverses the pointers in live blocks, modifying the pointer table and heap to indicate a return path. The mark operations are illustrated in Figure 5.5. The marking algorithm begins with a root block *block*. It traverses the fields in *block* until it finds a field that contains a valid pointer index pointing to an unmarked block. For each such field *block*[*mark*], the algorithm loads a pointer to the new block into *field*. Then, it modifies the pointer table entry for *field* to point back at *block*[*mark*], the field within *block*. *block*[*mark*] itself is modified to point to the head of the original block, *block*. All of these modifications are reversible as long as we know *field*. The mark phase continues, iterating over the fields of *field*.



STEP 1:
Initially, the mark algorithm examines a field *block*[*mark*] in block A.

STEP 2:
The mark algorithm begins scanning block B. B's pointer table points at the original field in block A, and the original field in block A points at A's header.

STEP 3:
The mark algorithm completes scanning of block B, and follows the reverse pointers to return to block A and continue. B's pointer table entry and the original field in block A are restored.

Figure 5.5: Illustration of pointer inversion in mark phase

When all fields in *field* have been checked, the algorithm backtracks to the original *block*. Using the index embedded in the header for *field*, the algorithm looks up its pointer table entry, which was previously modified to point to *block*[*mark*]. It then traverses the pointer in *block*[*mark*] to recover the original *block*, and then the value in *block*[*mark*] is restored to *field*. The mark phase is then able to continue iterating over the original fields in *block*.

To ensure the algorithm makes progress, it immediately marks each new block that it discovers, and it sets the least significant bit on the modified pointer table entry to prevent traversal of the now-invalid pointer table entry. All valid entries in the pointer table are even (blocks are word-

```
— Perform a minor collection; the live registers are in registers.
function MINOR-GC(registers):
    MARK-MINOR(registers)            — mark phase
    NORMALIZE(registers)             — convert pointers to index values
    POINTER-TABLE-MINOR              — revert pointer table to state before minor heap
    SWEEP-MINOR                      — sweep the minor heap
    SHIFT-MINOR                      — shift minor heap if it is full
    DENORMALIZE(registers)           — convert index values back to pointers
```

Figure 5.6: GC minor collection

aligned), so by setting the least significant bit in the pointer table entry, we ensure that references back to the current block will be ignored.

Whenever a block is marked in the GC functions, MARK-CONTENTS is called to mark all blocks reachable from it.

5.4 Minor collection

A minor collection collects only within the minor heap. During a minor collection, the live data in the minor heap is compacted, and a portion of the oldest data is moved to the major heap. The minor collector uses a mark-sweep algorithm for identifying live blocks, and updates entries in the pointer table as blocks are compacted. Any block located outside of the minor heap that contains pointers into the minor heap is added to the list of root blocks *minor_roots*. Minor collection never resizes the heap or pointer table areas; if the minor collection is unable to reclaim sufficient space, then the major collector is run.

The algorithm for minor collection is given in Figure 5.6. Minor collection involves the following actions:

1. During MARK-MINOR, the minor collector uses *minor_roots* and *registers* to mark live blocks in the minor heap. This function also removes blocks from *minor_roots* that do not contain pointers into *minor*.

   Invariant: (MARK MINOR INVARIANT) During any minor collection, only blocks $b \in [\mathit{base}[\mathit{minor}], \mathit{limit}[\mathit{minor}])$ are marked.

2. In NORMALIZE, the collector replaces all real pointers in registers with index values, so the blocks may be compacted without maintaining a pointer relocation map. This utilizes the pre-existing indirection to defer updating pointers in registers until all blocks have been relocated. Under this scheme, only the pointer table is updated when a block is relocated; at the end of collection, the register values are converted back to pointers by looking up the updated entry in the pointer table.

```
— Mark all reachable blocks in the minor heap.
function Mark-Minor(registers):
    mark all b in minor heap reachable from minor_roots ∪ registers
    remove all b from minor_roots that do not contain pointers into minor

— For COW-faulted minor blocks, update pointer table to refer to prior immutable version.
function Pointer-Table-Minor:
    if spec_next > 0 then
        for all b ∈ ptr_diff[spec_next − 1] do
            if ptr_table[indexof(b)] ≥ base[minor] then
                ptr_table[indexof(b)] := b
```

Figure 5.7: GC minor collection, mark phase

```
— Compact the specified interval in the heap.
function Sweep-Area(b′, interval):
    for all blocks b ∈ interval do
        if b marked then
            unmark b
            move block b to location b′
            ptr_table[indexof(b)] := b′     — Supersedes any prior version
            update any references to b in ptr_diff
            b′ := b′ + sizeof(b)
        else if ptr_table[indexof(b)] = b then
            ptr_table[indexof(b)] := empty     — No prior version of b
    return b′

— Compact the minor heap.
function Sweep-Minor:
    current := Sweep-Area(base[minor], minor heap)
```

Figure 5.8: GC minor collection, sweep phase

3. In Pointer-Table-Minor, entries in the pointer table referring to minor blocks are reverted to the parent of the block, if a parent exists. This code is responsible for reverting the pointer table entries in the event that the youngest version of the block is dead. This function *only* alters pointer table entries corresponding to blocks in the minor heap. The pseudo-code is given in Figure 5.7.

4. In Sweep-Area, the heap within *interval* is compacted. The pseudo-code is given in Figure 5.8. In this case, it is only used to compact the minor heap. For minor blocks that remain live, the pointer table entry is restored to the youngest version of the block (undoing the effect of Pointer-Table-Minor). For minor blocks that are dead, the pointer table entry is freed iff the pointer table points at the dead version of the block. If the pointer table refers to a parent of the block, then the pointer table is left unaltered.

5. In Shift-Minor, the minor heap is adjusted within the current speculation level if it is still

```
— Perform a major collection; the live registers are in registers.
function MAJOR-GC(registers, request_bytes, request_pointers):
    CLEAR-MINOR-ROOTS                         — clear the minor roots
    MARK-MAJOR(registers)                     — mark phase
    NORMALIZE(registers)                      — convert pointers to index values
    SWEEP-MAJOR                               — sweep the entire heap
    EXPAND-HEAP(request_bytes,
                request_pointers)             — expand the heap if necessary
    SETUP-NEW-MINOR                           — setup the new minor heap
    DENORMALIZE(registers)                    — convert index values back to pointers
```

Figure 5.9: GC major collection

nearly full. Often, $limit[minor] < limit$ to trigger minor collection more frequently. The minor heap is represented as a sliding window at the end of the heap. When it is overfull, data in the minor heap is moved into the major heap, and the minor heap slides toward the end of the heap. A major collection may be triggered once the minor heap reaches *limit*. This function uses a simple size heuristic to determine when to shift the minor heap.

6. Finally, DENORMALIZE replaces the saved index values in registers with real pointers, reversing the effect of NORMALIZE.

5.5 Major collection

A major collection is run whenever a minor collection is unable to reclaim sufficient space. The major collector also uses a mark-sweep algorithm and compacts live blocks in the heap. The major collector will cull all dead blocks in the heap, and will also resize the heap and pointer table if necessary to accommodate the request in $request\_bytes, request\_pointers$.

Since all data in speculation levels $\{0..gc\_level - 1\}$ must be live, it is sufficient for the major collector to collect the speculation levels $\{gc\_level..spec\_next\}$. After this collection, all data in levels $\{gc\_level..spec\_next - 1\}$ is live and immutable, therefore major collection can advance the *gc_level* pointer to *spec_next*.

The algorithm for major collection is given in Figure 5.9. Major collection involves the following actions:

1. CLEAR-MINOR-ROOTS clears the root marker on all blocks in the heap. Major collection will always reconstruct the minor heap as a new area of memory, so the existing root set must be cleared.

2. MARK-MAJOR marks live blocks in the heap. The pseudo-code for the mark phase is given in Figure 5.10. The mark phase marks live blocks in the youngest speculation level *spec_next* first. Then, it proceeds to mark blocks in speculation levels $\{gc\_level, spec\_next - 1\}$, from

```
— Revert entries in the pointer table to the indicated level.
function REVERT-POINTER-TABLE(table):
    for i := 1 to sizeof(table) do
        b := table[i]
        ptr_table[indexof(b)] := b

— Mark all reachable blocks in the heap above gc_level.
function MARK-MAJOR(registers):
    mark all b in youngest speculation level reachable from registers
    for l := spec_next − 1 down to gc_level do
        REVERT-POINTER-TABLE(ptr_diff[l])
        — Pointer table does not refer to speculation levels younger than l.
        mark all b in speculation level l reachable from registers
        — Next, make sure we mark blocks that are only reachable from older generations.
        mark all b reachable from ptr_diff[l], ..., ptr_diff[l − 1]
        mark spec_env[l] and all blocks reachable from it    — Environment is always live
    — The next step ensures pointer table refers to blocks in the uncollected region.
    if gc_level > 0 then
        REVERT-POINTER-TABLE(ptr_diff[gc_level − 1])
```

Figure 5.10: GC major collection, mark phase

youngest (largest level number) to oldest (smallest level number). While marking level l, it is important that each pointer table entry refers to a version that was current when the process transitioned from level l to level $l+1$; MARK-MAJOR maintains the following invariant:

Invariant: (MARK POINTER TABLE INVARIANT) when marking speculation level l, for all blocks $b \in [base[0], limit[l])$, $ptr\_table[b] \leq limit[l]$.

Note that MARK-MAJOR marks blocks in the entire heap, not just within the uncollected region above $base[gc\_level]$. This is necessary because it is possible the only surviving references to a block in the uncollected region are from blocks in the collected region. As a result, the sweep phase needs to scan all of the heap, even though compaction only occurs above $base[gc\_level]$. Note that at most one version of a block in the uncollected region is marked.

At the end of MARK-MAJOR, for every block b, its pointer table entry refers to the youngest version of b within the uncollected speculation levels $\{0..gc\_level - 1\}$, if such a version exists. Otherwise, the pointer table entry refers to the oldest version of the block. The effect on the pointer table is similar to POINTER-TABLE-MINOR in minor collection — if all versions of a block are dead in the collected region, then the pointer table is left referring to the most recent version in the uncollected region.

3. NORMALIZE replaces real pointers in registers with index values.

4. SWEEP-MAJOR compacts the heap, starting at speculation level gc_level and continuing to the youngest speculation level. The pseudo-code for the sweep phase is given in Figure 5.11.

```
— Sweep the heap above (and including) gc_level.
function SWEEP-MAJOR:
    — Clear the mark bits from previously-collected levels.
    unmark all blocks b ∈ [base[0], limit[gc_level − 1])
    — Sweep all uncollected levels.
    b′ := base[gc_level]
    for l := gc_level to spec_next do
        new_base := b′
        b′ := SWEEP-AREA(b′, level l heap)
        base[i] := new_base
    current := b′
    copy_point := base[spec_next]
    base[minor] := current
    gc_level := spec_next
```

Figure 5.11: GC major collection, sweep phase

The heap is compacted from *gc_level* up to the youngest generation, in order. As with minor collection, the pointer table is updated for every live block encountered, resulting in a pointer table that always refers to the youngest version of a block.

If an unmarked block is encountered and the pointer table entry refers to that version of the block, then the pointer table entry is released. Note that there can only be one version of a block b within a given speculation level. Also, for a version to exist in speculation level l (dead or live), there must be a version of the block in speculation level l', $l' < l$ that is live. As a result, only the most recent version of a block can be dead at any given time. This means that the mechanism to free pointer table entries does not interfere with the mechanism that ensures the pointer table refers to the most recent version of a block.

Note that SWEEP-MAJOR is also responsible for clearing mark bits in the speculation levels that were already collected.

5. EXPAND-HEAP resizes the heap and pointer table areas if necessary, if there is still insufficient space to fulfill the request after the major collection.

6. SETUP-NEW-MINOR sets up a new minor heap starting at *current*.

7. DENORMALIZE replaces index values in registers with real pointers.

Chapter 6

Conclusion

6.1 MCC benchmarks

System benchmarks are shown in Figure 6.1 for version 0.5.0 of the Mojave compiler, which was released in May 2002, about a year after the Mojave project started. The performance numbers measure total real execution time on an unloaded 700MHz Intel Pentium III. The Mojave system is freely available at `mojave.caltech.edu` under the GNU General Public License.

The Mojave system is currently under development, and benchmark performance varies widely. Performance numbers are given for several compilers. The `gcc` column uses the GNU compiler collection, version 2.96; `gcc2` uses the `-O2` optimization. The `mcc2` columns list performance numbers for the Mojave compiler. For comparison purposes (only), the `mcc2u` column lists performance without runtime safety checks. In the current state of development, the `mcc2` compiler performs only minimal optimization, including dead-code elimination, function inlining, and assembly peephole optimization. Advanced FIR optimizations are fairly easy to implement, and the `mcc6u` column lists performance numbers using an optimizer under development that implements alias analysis and partial redundancy elimination. Naml benchmarks are similar, and include numbers for the INRIA OCaml compiler [16], version 3.04.

The specific benchmarks include the following. The `fib` program computes the n^{th} Fibonacci number (using the naive exponential-time algorithm). This benchmark is highly recursive, and the performance numbers reflect the use of continuation-passing style. The `mcc` programs allocate an exponential number of closures *on the heap*, and much of the time is spent in garbage collection.

The `mandel` benchmark computes a Mandelbrot set. This is a special case where `mcc` C compiler, using the standard optimizations, happens to perform significantly better than `gcc -O2` (performance numbers for `gcc -O3` are shown in parentheses). In contrast, the performance for Naml reflects the use of minimal optimization. The program is implemented with fixed-point numbers, and each arithmetic operation is a function call. The `ocamlopt` compiler inlines the function calls, while `mcc2` and `ocamlc` do not.

C benchmarks (time in seconds)					
Name	gcc	gcc2	mcc2	mcc2u	mcc6u
fib 35	1.0	0.78	4.6	4.6	4.32
mandel	54.7	42.1 (5.5)	7.2	7.3	6.0
msort1	3.83	1.15	5.92	3.01	
msort4	5.4	1.15	8.22	4.13	
imat1	37.1	6.27	27.9	17.3	7.6
fmat1	8.9	2.98	10.2	8.33	4.86
migrate			1.77		
regex			2.87		

Naml benchmarks (time in seconds)				
Name	ocamlc	ocamlopt	mcc2	mcc2u
fib 35	3.89	0.61	8.33	7.81
mandel	545	8.1	183	160

Figure 6.1: Mojave benchmarks

```
function match(pattern, buffer):
    pattern_index := 0, buffer_index := 0
    if atomic_entry(0) ≠ 0 then
        print_string("Pattern did not match")
        return false
    while pattern[pattern_index] ≠ nil do
        if pattern[pattern_index] = "*" then
            if buffer[buffer_index] = nil then
                increment pattern_index
            else if atomic_entry(0) = 0 then
                increment buffer_index
            else
                atomic_commit()
                increment pattern_index
        else if pattern[pattern_index] = buffer[buffer_index] then
            increment pattern_index and buffer_index
        else
            atomic_rollback(1)
    if buffer[buffer_index] ≠ nil then
        atomic_rollback(1)
    print_string("Pattern matched")
    return true
```

Figure 6.2: Unix-style pattern matching using speculations

The `msort` benchmarks implement a bubble-sort algorithm, `imat1` performs integer matrix multiplication, and `fmat1` tests floating-point matrix multiplication.

The `migrate` benchmark measures the minimal process migration time. The program consists of a single migration call. Nearly all of the time is spent in recompilation on the target machine.

The `regex` algorithm is a naive, imperative implementation of a Unix-style regular-expression matcher, using speculations to perform backtracking. C-style code for the algorithm is shown in

Figure 6.2. The time listed is for determining that the pattern `*h*e*l*l*o*w*o*r*l*d*` occurs in the text of the introduction to this paper. The benchmark enters 945341 speculations with a maximum speculation nesting depth of 6833.

6.2 Future work

Currently, process checkpointing and migration do not extend to process I/O or any other machine-specific resource that may be accessed through standard libc. This requires support from the operating system. To partially accommodate I/O operations, we are developing the MojaveFS distributed filesystem. MojaveFS provides a distributed filesystem whose namespace and resource handles are consistent across all nodes in a distributed system. MojaveFS also incorporates primitives for managing speculation information associated with a file in the system. The speculative information manages both metadata information, such as file creation and deletion, `open` and `close` operations, and file data that is modified by `write` calls. By standardizing the directory namespace and resource handles, we remove several dependencies a process might have on a particular machine. Any I/O to files under MojaveFS will be subject to speculative rollback; that is, $\mathcal{C}$ will reflect the state of all files in MojaveFS.

We may also accommodate other I/O operations with operating system assistance, including I/O to specific devices that may be connected to a particular node. This will allow processes with device-specific I/O to migrate to a remote location, but the device-specific I/O may not be subject to speculative rollback. In particular, I/O to sequential-mode devices such as printers, and I/O to the console, cannot be subject to speculative rollback because the device is unable to undo the operation after it is performed. The operating system may provide several policies for controlling I/O in this case, including a policy that postpones all `write` operations until all speculations that were pending on the `write` have been committed. It is feasible in certain circumstances to allow the write to proceed even if a speculation is pending, such as for debugging output sent to a console or logging device.

The current version of the MCC compiler has several performance issues that are introduced by the conversion of inherently imperative languages into a functional form which relies on continuation-passing-style, followed by a conversion back to an architecture designed for imperative execution. We need to introduce more optimizations to improve the performance of MCC, and make it a more feasible compiler for real-world applications.

Currently, we have not completed work on the high-level language primitives. MCC currently exposes the migration operation and the three speculation operations directly to the source languages. We are considering several higher-level primitives which are more natural for the expression of distributed algorithms. Most of this work focuses on using guarded statement constructs to encapsulate fault tolerance.

Bibliography

[1] *IA-32 Intel Architecture Software Developer's Manual*. Intel Corporation, 1997.

[2] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 365–377, 2000.

[3] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, pages 198–229, 2001. Special Issue on Coordination.

[4] Sylvain Conchon and Fabrice Le Fessant. Jocaml: mobile agents for Objective-Caml. In *ASA/MA'99 Joint Agents Symposium*, October 1999.

[5] Cristian Ţăpuş, Justin D. Smith, and Jason Hickey. Kernel level speculative DSM. 2003. Submitted to the *Distributed Shared Memory* workshop, awaiting notification.

[6] G. Di Marzo Serugendo, M. Muhugusa, and C. Tschudin. A survey of theories for mobile agents. *World Wide Web Journal, special issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents*, 1998.

[7] Cdric Fournet, Georges Gonthier, Jean-Jacques Lvy, Luc Maranget, and Didier Rmy. The reflexive chemical abstract machine and the join-calculus. In *The 23rd Annual ACM SIGPLAN–SIGACT Symposium on Principles Of Programming Languages (PoPL'96)*, January 1996.

[8] Jason Frantz, Cristian Ţăpuş, Justin D. Smith, and Jason Hickey. MojaveFS: A transactional distributed file system. 2003. Unpublished.

[9] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1994.

[10] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, November 1994. Short Communication.

[11] Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz, and Cristian Tapus. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR 2002.007, California Institute of Technology, Computer Science, July 2002.

[12] Jason J. Hickey. *The MetaPRL Logical Programming Environment.* PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, 2001.

[13] An-Chow Lai and Babak Falsafi. Memory sharing predictor: the key to a speculative coherent dsm. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 172–183. IEEE Computer Society Press, 1999.

[14] Shubhendu S. Mukherjee and Mark D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 179–190. IEEE Press, 1998.

[15] J. Oplinger and M.S. Lam. Enhancing software reliability using speculative threads. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems.*

[16] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object–oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 40–53, 1997.

[17] Justin David Smith. Kupo language specification 1.3.1. (available as part of the MCC distribution), August 2002.

[18] Justin David Smith. System migration 1.0. (available as part of the MCC distribution), July 2002.

[19] Giovanni Vigna, editor. *Mobile Agents and Security.* Springer, 1999. LNCS 1419.

Index