The Mojave Compiler: Providing Language Primitives for Whole-Process Migration and Speculation for Distributed Applications

Justin D. Smith [1], Cristian Ţăpuş[2], and Jason Hickey[2]

[1]University of Pennsylvania
jyasu@cis.upenn.edu

[2]California Institute of Technology
{crt,jyh}@cs.caltech.edu

Abstract

We present an approach for implementing language-level primitives for whole-process migration and speculative execution in a compiler and associated runtime environment. These primitives are exposed to the user through simple language constructs that do not require the user to manage process state explicitly. With migration and speculation we show how the user can quickly add persistent checkpoints to any large-scale distributed application that requires longevity in a faulty environment. We demonstrate the use of migration and speculation primitives for checkpointing in a canonical grid computation application, and analyze the results of this implementation.

1. Introduction

The most intuitive way to provide reliability in software is by allowing applications to take recoverable checkpoints during their lifetime. Writing checkpoints to persistent storage such as a disk array can be expensive; recovery has a similar cost. In a distributed system, the cost of recovery can be reduced by restoring only the state of processes that have failed. Our work reduces the cost of checkpoint recovery by introducing primitives that can take advantage of resident state on surviving nodes. These primitives utilize copy-on-write mechanisms to preserve and recover a recent state entirely in local memory.

In order to provide more flexibility in how a checkpoint is taken our implementation splits the checkpoint operation into two primitives: one for process migration and one for speculative execution. In the presence of reliable distributed persistent storage, process migration allows a process to create a checkpoint by migrating into persistent storage.

A speculation is a computation based on a precondition. Speculations allow a process to make progress in a computation by optimistically assuming that the precondition is satisfied before it is evaluated. If it is later discovered that the precondition is not satisfied, the process is rolled–back to the state immediately before the assumption was made, and it may potentially take a different execution path. Speculations are defined by three operations: speculate, which starts a new speculation; abort, which cancels the effects of a speculation by rolling back the program to the state just before the speculation was started; and commit, which discards a rollback point of a speculation for which the precondition was successfully verified. Speculations share many traits with traditional distributed transactions, one of the earliest and simplest abstractions for reliable concurrent programming [5]. Unlike transactions, they allow speculative processes to use communication. This relaxation of the Isolation property increses parallelism but it also requires processes that depend on the values generated by a speculative process to join that process's speculation and roll back together in case of failure.

The main contributions presented in this paper are: introducing speculative execution constructs as programming language primitives; and implementing speculative execution and process migration primitives in a compiler and runtime environment. The compiler generates process state management code automatically, removing the need for the user to implement hand-written checkpointing code.

We begin by providing a set of examples on how to use the speculation and migration primitives to write fault-tolerant and efficient distributed applications, followed by an overview of our approach and an introduc-

1-4244-0910-1/07/$20.00 ©2007 IEEE.

```
Transfer (obj1, obj2, k) {
   // We want the transfer to be atomic
   // given that read and write are atomic
   if (read (obj1, buf1, k) != k)
      return failure;
   if (read (obj2, buf2, k) != k)
      return failure;
   if (write (obj1, buf2, k) != k)
      return failure;
   if (write (obj2, buf1, k) != k) {
     // Undo first write
     while (write (obj1, buf1, k) != k) {
     // Unrecoverable error on write failure
     // Inconsistent state. Try again...
     }
     return failure;
   }
   return success;
}
```

```
Transfer (obj1, obj2, k) {
   if ((specid=speculate())>0) {
      // Enter speculation
      if (read (obj1, buf1, k) != k)
         abort(specid);
      if (read (obj2, buf2, k) != k)
         abort(specid);
      if (write (obj1, buf2, k) != k)
         abort(specid);
      if (write (obj2, buf1, k) != k)
         abort(specid);
      commit(specid); // Speculation committed
      return success;
   } else { // Speculation aborted
      return failure;
   }
}
```

Figure 1. Using speculations for fault-tolerance. Top half code is written in the traditional programming model. Bottom half code uses the speculative model, which separates the error recovery code from the transfer operation.

tion of the compiler. In Section 6 we present related projects and compare them to our approach. Details of the implementation are discussed in Section 4. Experimental results are shown in Section 5. The paper is concluded in with a discussion of future avenues of research emerging from this work.

2. Speculations through examples

The first example we consider is the traditional database example of money transfer between two accounts, represented as a transfer of data between two objects. We want to implement an atomic function that swaps the first k bytes of two account objects, obj1 and obj2, using read/write operations that may fail. Figure 1 presents a traditional implementation of such a transfer function together with a speculative version. One of the more important differences is that the speculative version *separates* the error recovery code from the implementation of the transfer operation. In contrast, in the traditional version the error recovery code is written in-line, which can obscure the code, and also makes error recovery dependent on the execution path.

In this example, we have included explicit calls to the speculation operations speculate, abort, and commit. However, an even simpler implementation can be obtained by treating the speculation like an exception mechanism, where the read/write operations raise an exception on failure. In this case, the speculative assumption is that the transfer is successful; if an error occurs, the speculation is rolled-back.

Along the same lines, we can use speculative execution to prevent certain types of software bugs from crashing applications. For example, applications that suffer from unchecked buffer overflow issues could be instrumented using speculative execution. The instrumentation would make it such that if a buffer overflow occurs the program is rolled back to where the memory allocation occurred and a different path of execution (potentially allocating more memory and retrying) could be taken, thus preventing the application from crashing. This would require minimal support from the operating system. A similar approach, using only simple checkpoints was suggested in the Rx [10] system. The advantage of using speculative execution as opposed to traditional checkpointing is that users can instrument their own code and provide alternate execution paths based on how the precondition of their speculation is invalidated.

The second example (Figure 2) presents the benefits of using the primitives for speculations and process migration through a scientific computing application. The computation runs for very long periods of time and is prone to node failures.

The single processor implementation creates a matrix and computes the value of each grid point based on the values of its neighbors at the previous time step. Parallelizing the application and making it run in a distributed environment is of particular interest to the problem we address in this work.

The decomposition of the computation grid and the distribution of the work for a sample example is presented in Figure 2. The computation domain of each node overlaps with the computation domain of its neighbors, allowing it to compute local information with only limited boundary information from its neighbors. The update of the border data is done using a customized message passing interface.

The application has been implemented in the C language with primitives for speculation and process migration, and it has been compiled using our compiler,

```
specid=speculate();
for(step = 1; step <= timesteps; step++) {
   /* Get boundary values from neighbors. */
   /* May have to rollback due to failure */
   err=get_borders(u,rows,cols,myid,step);
   if(err == MSG_ROLL)
       abort(specid);
   /* Perform the computation. */
   do_computation(step, u, rows, cols);
   /* Save a checkpoint if it's time. */
   if((step % checkpoint_interval) == 0) {
       /* Save the current speculation */
       commit(specid);
       /* Save checkpoint to file */
       migrate(checkpoint_name);
       /* Start a new speculation */
       specid=speculate();
   }
}
```

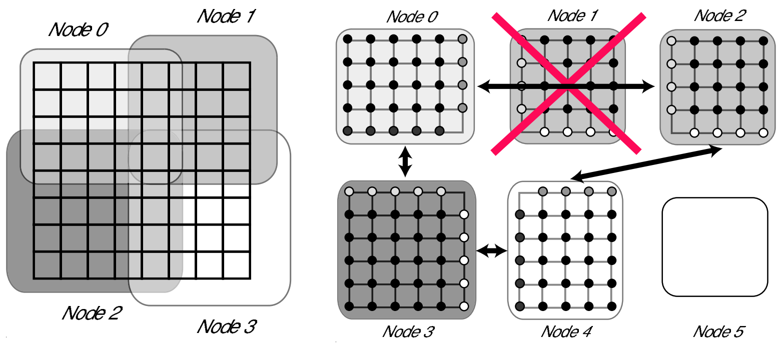

Figure 2. Simplified speculative main loop, the 2D domain decomposition, and the work distribution and process migration

the MCC. The main computing loop of the application is presented in Figure 2. The program starts a speculation by calling the *speculate* function. A speculation is started at the beginning of the computation and after each checkpoint. Checkpoints are taken regularly, after a fixed number of iterations. Depending on the failure frequency, this parameter of the application can be adjusted to balance the overhead of speculations against the expected cost of fault recovery. At each time step, each computing cell retrieves the boundary information from its neighbors. If any of its neighbors fails, the local computation is rolled back to the previous speculation, and the border information for that timestep is requested again from the neighbors. In this particular example there is a guarantee that the computation will not rollback more than one speculation, due to the commit operation present before each checkpoint is taken. In the general case however, any uncommitted speculation can be rolled back, independent of its age, and commits for speculations can occur out of order.

When a failure is observed at a given node the computation has to be revived on a different node in the system. To allow migration, each node of the computation grid executes an MCC migration daemon. If automatic resurrection of failed computation processes is expected, the application needs to implement specific daemons to resurrect failed processes. The checkpoints are formatted as executable files and the resurrection of processes is done by executing the saved checkpoint. A run of this application on the cluster in the presence of a fault is also illustrated in Figure 2. If computing node 1 fails the computation thread is resurrected on a remote node from the last checkpoint. All the other processes rollback their last speculation to bring the computation to a consistent state. The existence of a reliable and distributed storage medium is needed for a real fault-tolerant implementation. For the purpose of this example an NFS mount point visible across the entire cluster provided the required functionality.

The code presented in Figure 2 shows the clear distinction between the checkpointing and speculative code and the rest of the algorithm and it can easily be used as a template for a large variety of scientific computing applications. The minimal annotation required through the use of specific language primitives is computation and architecture independent.

3. Overview of MCC

We implement whole-process migration and speculative execution as part of the Mojave Compiler Collection (MCC). We built MCC as a multi-language compiler that compiles C, Pascal, ML, and Java. MCC provides an active test bed for research in several areas of distributed systems.

We chose to expose migration and speculation to the user as language primitives. Since the compiler determines how process state is organized, we can use it to automatically generate code to manage the process state during migration or speculative execution, requiring minimal knowledge from the user.

The compiler is in an ideal position to enforce *safety* in a program, by introducing runtime safety checks. The compiler can ensure the process will not attempt to access illegal areas of memory or use values with inappropriate types. To support this, MCC compiles all source languages to a semi-functional intermediate representation (FIR) [7, 13]. FIR is a type-safe intermediate language where variables are immutable, but heap values can be modified. Function calls in the source language are converted to tail-calls using continuation passing style. Loops are expressed with recursive functions.

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. Our primary runtime implementation is a native-code runtime for the Intel IA32 architecture. An additional runtime environment is available that simulates RISC architectures. Object code generation is performed by elaborating the FIR code to machine-specific assembly code, introducing runtime safety checks as necessary.

MCC is a suitable platform for developing whole-process migration primitives in an efficient manner. Also, MCC's safety properties make it ideal for use in distributed applications that are deployed over untrusted networks such as the Internet. MCC's heap design allows for easy support of speculative execution models.

4. Implementation of migration and speculations

Most support for process migration and speculations is provided by the MCC runtime environment. The runtime manages several tasks, including garbage collection, process migration, speculation, and runtime type-checking for heap operations. Process migration and speculation are tightly integrated with the garbage collector.

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. The interaction with the garbage collector is beyond the scope of this paper, but is discussed elsewhere [13, Chapter 5].

4.1. Process state in the runtime

To support process migration and speculation, the runtime provides a standardized, architecture-independent representation of the entire program state. Each memory structure, or *block*, is stored in a *heap*. Each block has a header, and stores its data in an architecture-independent format.

Data for FIR variables are stored in *registers* in a machine-dependent representation for efficiency. During migration, register state that is live across a migration point is copied into the heap first so a standard, architecture-independent representation of the data may be migrated.

Data blocks in the heap are tracked by a *pointer table*. 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. With speculation, it is possible that there will be valid blocks in the heap whose pointer table entry refers to a different block; these special blocks are tracked by a *checkpoint record* in the event that a speculation is rolled back. A *function table* contains pointers to all valid higher-order functions.

The program state includes code in a *text area*, containing 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.

4.1.1 Pointer table

The pointer table's main purpose is to allow for relocation (enabling migration and speculation) and to provide safety for C memory. The pointer table is implemented in software, however its design is compatible with a hardware implementation for increased efficiency.

Figure 3 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 provides a simple mechanism for identifying and validating data pointers in aggregate blocks. When an index i for a base pointer is read from

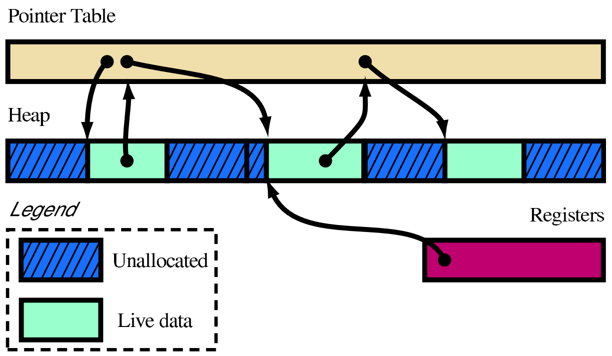

Figure 3. Pointer table representation

the heap, i is checked against the size of the pointer table T to verify if it is a valid index, then T_i is read from the i^{th} entry in the pointer table and checked to ensure it is not a free entry in the pointer table. These steps can be performed in a small number of assembly instructions, and ensure T_i is a valid block pointer.

The pointer table also supports 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.

4.2. 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. The mechanism for migrating a process from one machine to another 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. The same mechanism is used to generate checkpoint files while the process is running.

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.

4.2.1 Using process migration in the FIR

Process migration is expressed in the FIR as a pseudo-instruction: **migrate** $[i, a_{ptr}, a_{off}]\ f(a_1, \ldots, a_n)$. 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. (a_{ptr}, a_{off}) is a pointer (pointer to head of a data block, and offset within that block) that refers to a string describing the migration target. The string includes information on what protocol to use to transfer state to the target. $f(a_1, \ldots, a_n)$ is a continuation function (with arguments) to call once the program has migrated.

There are three protocols that may be used for process migration: `migrate`, `suspend`, and `checkpoint`. The `migrate` protocol sends the entire state of a process to another machine for immediate execution, 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.

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.

4.2.2 Runtime support for migration

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.

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. All heap data follows the standard, architecture-independent MCC representation, including *migrate_env*; 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.

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

To implement the **migrate** operation, the source machine first transmits the following data to the server: FIR code for the process, size of heap and pointer tables, index of the block containing live variables (*migrate_env*), and 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 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 to preserve order in the pointer and function tables.

4.3. Speculative operations

Semantically, speculative execution appears atomic; that is, either all the operations in a speculation succeed, or none of them 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.

When a speculation is aborted, the entire process state, including all variable and heap values, is 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. 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 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: **speculate**, 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.

4.3.1 Using speculations in the FIR

Each **speculate** 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 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 **speculate** operation is represented in the FIR using the following primitive: **speculate** $f(c, a_1, \ldots, a_n)$. The function f is called within a speculative context. f does not return (the FIR is expressed in a continuation-passing style), and all live data must be passed as arguments. c is an integer that is passed as the first argument to f. On rollback, the value of c passed to f may be changed to indicate that the rollback occurred. This is currently the only way to carry state information across a rollback.

The primitive for the **commit** operation is **commit** $[l]$ $f(a_1, \ldots, a_n)$. This commits data for level $l \in \{1 \ldots N\}$ by folding all changes from that level into its previous level. Once the speculation is committed, the function $f(a_1, \ldots, a_n)$ is invoked.

The primitive for the **rollback** operation is **rollback** $[l, c]$. This reverts all changes made by in level $l \in \{1 \ldots N\}$ and *all later levels*. Rollback resumes execution at the point where level l was entered. The function that was called when level l was entered is saved as part of the checkpoint and is called with the original arguments but with the new value for c. This version of the primitive is a *retry* primitive; level l is automatically re-entered after it has been rolled back. In effect, the state that is captured and restored is the state immediately after level l was entered.

5. Speculations and process migration in action

Due to space constraints we only present a brief summary of the experimental results. A detailed discussion can be found in a technical report [13]. The test bed is a cluster composed of nodes with dual 700MHz processors and connected via a 100Mbps network.

We observed a migration time of 4 seconds for a process with a 1MB heap in an untrusted environment that required re-compilation of the FIR at the destination. Of this 10% represented the actual network transfer and the rest was due to re-compilation. For the same process, the binary migration time was under 1 second, of which 30% represented the data transfer from source to destination. This overhead included the time for establishing a TCP connection and the actual data transfer.

The cost of speculative execution was analyzed as a function of the mutation percentile of the heap during the life of the speculation. While the entry time was independent of it, and ran at about $40\mu sec$ for a process with heap of size 200KB, the abort time of a speculation in case of a 10% mutation was $120\mu sec$ and went up to $135\mu sec$ in the case of 100% mutation. The commit times were $81\mu sec$ for a mutation of 10% and $87\mu sec$ for a mutation of 100%. By comparison, the context switch time on the cluster used for data collection was about $300\mu sec$ if only 2 processes with heap sizes of 200KB ran in parallel.

In conclusion, the overhead from using speculative execution and process migration is small compared to having to re-start the application from scratch in the presence of certain types of failures. This is especially true in the case of long-running applications where migration is infrequent.

6. Related Work

Whole-process migration has been widely studied [2, 15]. The JoCaml system [1] provides process mobility for OCaml programs based on the join calculus [3].

DEMOS/MP [9] is a message-based operating system that provides process migration by leaving a stub of the process on the *source* node to forward all communication to the *destination*, or current location, of the process. DEMOS/MP does not address the issue of node failure. Kerrighed OS [4] is an operating system designed to provide a *single system image* of a distributed environment, which supports process migration between its nodes. However, using a custom operating system in a production environment and the lack of portability to multiple architectures are some of the main concerns of these approaches. The Mojave compiler takes a formal approach to process migration by using a functional intermediate representation (FIR) of programs based on a formal operational semantics [7]. Furthermore, due to the use of the FIR, which is a concept similar to Java bytecode, the Mojave compiler provides process migration in heterogeneous environments. Unlike the Java bytecode, the FIR is a higher level representation of programs that could be used to verify the correctness of the programs through the use of theorem provers.

Ramkumar and Strumpen discuss the idea of portable checkpoints in heterogeneous environments [11] by using a source-to-source compiler. They developed a C to *fault tolerant C* compiler. Their approach is limited by the use of ambiguous type information when generating checkpointing code and by requiring the use of specific memory allocation routines provided by their system.

While speculations are similar to the concept of lookahead-rollback introduced by the TimeWarp [8] mechanism, we extend the concept by enabling speculations throuh programming language extensions.

Speculations share many traits with traditional distributed transactions, one of the earliest and simplest abstractions for reliable concurrent programming [5]. 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. An important difference between speculations and transactions is that speculations do not provide isolation, allowing processes to communicate and collaborate while inside an atomic operation.

As part of the Venari project, Haines et.al. [6] implement a transaction mechanism as part of Standard ML, utilizing a mutation log produced by a generational garbage collector to implement undoability. The speculatives in MCC provide a programming paradigm that permits optimistic distributed computation.

Recent related work includes the AtomCaml [12] project, which is an extension to Objective Caml that provides a synchronization primitive for atomic (transactional) execution of code to replace locks. In our approach we allow speculative programs to use communication to interact with other programs while executing inside a speculations.

7. Conclusion and Future Work

We presented a new approach for checkpoints in distributed applications using programming language primitives for process migration and speculative execution. Applications using these constructs are more reliable and recover more easily from failures.

We are exploring ways to enhance this work to support speculative I/O with MojaveFS, a distributed speculative filesystem. This will allow users to use normal file I/O operations and socket communication using standard calls inside a speculation. While we have not discussed the issue of migrating network connections, we believe our work can be easily integrated with a system like Migratory-TCP [14].

Migration and speculation primitives allow for a number of interesting programming concepts, such as dynamic transparent load balancing and mobile agents. Our work can be applied to a wide variety of applications in applied sciences that would benefit from automatic checkpointing code and code mobility.

References

[1] S. Conchon and F. L. Fessant. Jocaml: mobile agents for Objective-Caml. In *ASA/MA'99 Joint Agents Symposium*, October 1999.

[2] G. Di Marzo Serugendo, M. Muhugusa, and C. Tschudin. A survey of theories for mobile agents. *World Wide Web Journal, special issue*, 1998.

[3] C. Fournet, G. Gonthier, J.-J. L. a nd Luc Maranget, and D. R¡E9¿my. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of PoPL'96*, January 1996.

[4] P. Gallard and C. Morin. Dynamic streams for efficient communications between migrating processes in a cluster. In *Euro-Par 2003: Parallel Processing*, volume 2790.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1994.

[6] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, November 1994. Short Communication.

[7] J. Hickey, J. D. Smith, B. Aydemir, N. Gray, A. Granicz, and C. Ţăpuş. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR 2002.007, California Institute of Technology, Computer Science, July 2002.

[8] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3), 1985.

[9] M. L. Powell and B. P. Miller. Process migration in demos/mp. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*.

[10] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, New York, NY, USA, 2005. ACM Press.

[11] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous archtitectures. In *FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*.

[12] M. F. Ringenburg and D. Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM Press.

[13] J. D. Smith. Fault tolerance using whole-process migration and speculative execution. Master's thesis, California Institute of Technology, Department of Computer Science, 2003.

[14] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: Connection migration for service continuity in the internet. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*.

[15] G. Vigna, editor. *Mobile Agents and Security*. Springer, 1999. LNCS 1419.