System Migration

Version 1.0
Justin David Smith
July, 2002

Abstract

System migration is a primitive in the Mojave Compiler Collection (MCC) designed to allow programs to migrate their entire state from one machine to another, independent of architecture or language. This paper describes the basic use of system migration, including use in the source language and use of the various migration servers. This paper also discusses the implementation details for system migration in MCC.

Contents

1 Introduction

Note: The current version of this document is not thorough.

1.1 Motivation

Since the invention of the first digital computers, the processing capability of computers has increased at an exponential rate rougly following Moore's law. The form of this increase has changed over the years; the supercomputers of past decades have been replaced by distributed systems composed of many low-end machines, or *nodes*, each linked by a high-speed communication network. Computations may be performed on these distributed systems, however the computation must be broken up into smaller parts, and each part must be assigned to a specific node in the system. A large amount of code in a distributed computation is devoted to maintaining synchronization, and attempting to recover lost computation when a particular node fails. This tends to be very cumbersome for the programmer involved — in particular, if the system is to be fault-tolerant, then the programmer must spend a great deal of time determining all the possible failure modes for the system and how to recover from them.

As distributed computing becomes more ubiquitous in technology, we would like to develop a system where each node is indistinguishable from another, and where processes in the system see the machine as a single entity. Processes in such a system should be free to move from machine to machine, and link themselves with other processes to interact. This notion of *process mobility* is the central part of *ambient calculus*. This system should be integrated with the programming language, to ease the job of the programmer.

We would like any mechanism we design to support fault-tolerance, such that if a particular node in the system fails, the mechanisms will automatically recover the computation without any special effort from the programmer. We can implement *rollback* which allows a process to return to a previous "known-valid" state should a later operation fail. By implementing *atomic* operations, we can write a block of code that will attempt to perform a series of operations in the block, but will rollback to the beginning of the block (as if none of the operations were executed) should any one operation fail.

With these higher-level primitives at the language level, we allow programmers to write distributed applications with much greater ease and with little direct knowledge of the underlying architecture of the system. With an administrative monitor, we may also improve the efficiency of the system with the load-balancing and resource localization techniques described above.

1.2 Related references

Migration is also discussed in the technical report caltechCSTR 2002.007, *Process Migration and Transactions Using a Novel Intermediate Language*, by Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz and Cristian Tapus, July 2002.

2 Using System Migration

2.1 Migrate destination format and protocols

The migrate destination format is a string which describes how a process should be migrated. It resembles standard HTTP URLs, and consists of a protocol component and a name component.

mdf	::=	"`host://`" hostname ("`:`" integer)$^?$	Migrate to remote hostname, port integer
	|	"`suspend://`" filename	Save state to filename, and terminate
	|	"`checkpoint://`" filename	Save state to filename, and continue

These protocols are discussed in more detail below:

- `host://`: The host protocol is the standard form of system-to-system migration. A hostname and optional port number are specified; a migration server should be running at this location. The process will attempt to migrate its entire state to the remote machine. If the state is successfully migrated, then the process on the source machine will terminate, and the process will resume at the proper location on the destination machine. If any errors occur, then the process will continue to run on the source machine, and no process will be started on the destination machine.
- `suspend://`: The suspend protocol will save the process state to a file on disk, instead of sending it to a remote machine. The name of the file to write to is specified as part of the format. If the state is successfully written to disk, then the process will terminate; otherwise, the process will continue. A process state file can be resumed on any machine using the resurrection utility.
- `checkpoint://`: The checkpoint protocol is similar to the suspend protocol, but the process will continue executing even when the process state is successfully written to the disk. This is useful for recovering a process state when a node in the network fails; if the process has taken periodic checkpoints, then the process can be recovered from its last checkpoint.

2.2 Migration utilities

There are two migration programs which may be used by the end user — `mcc-migrated` and `mcc-resurrect`. Both of these utilities are in the `net/migrate` directory of MCC.

2.2.1 The `mcc-migrated` program

`mcc-migrated` is the migration server, and is run on any machine willing to accept migrated programs. Run `mcc-migrated -help` for a list of supported command-line options. Migration options that are shared by all programs are discussed in section 2.2.3; the only other option of interest for the server is `-port`, which allows you to run the migration server on a nonstandard port. It is usually safe to leave this set to the default.

The migration server is able to accept and run multiple processes at once; it will fork a new compiler for each process that is received. After migration, processes will inherit the standard I/O channels used by the migration server. If migration fails, then an error will be printed to the server's `stderr` channel, but other processes running on the server will not be interrupted. The migration server does not terminate under normal circumstances (it may be killed using standard UNIX signals).

2.2.2 The `mcc-resurrect` program

`mcc-resurrect` is the resurrection program, and is used to recover the process state file that was written by a suspend or checkpoint operation. Run `mcc-resurrect -help` for a list of supported command-line options. Migration options that are shared by all programs are discussed in section 2.2.3; there are no options of interest that are specific to `mcc-resurrect`. `mcc-resurrect` expects one non-option argument, the name of the file to read the process state from.

The resurrection program will attempt to compile the FIR code stored in the process state; if it is successful, then it will load the process heap data and run the program. The process will inherit the standard I/O channels used by the resurrection program. Once the process terminates, the resurrection program will also terminate.

2.2.3 Options shared by migration programs

Many options are shared by all migration programs; most of the shared options control various debugging aspects of the compiler and migration. Note that while migration programs will invoke the compiler from the FIR stage into the backend, you are not allowed to specify options to control the compilation[1] — the options used to compile a migrated program must match the options used for the original program, therefore the options will be migrated along with the remaining program state.

In addition to debugging options that are imported from the compiler, several debugging options specific to migration are defined:

- `-migrate_emit_fir` emits the received FIR to a file. The location of the file emitted will be printed on the `stderr` channel.
- `-migrate_save_obj` will preserve the object file(s) and executable program generated by the compiler. By default, the object file(s) and executable program are unlinked once the migrated process terminates. Note that the object file(s) and executable program have limited use in debugging; they cannot be independently run since they do not contain the heap data.
- `-migrate_no_exec` will prevent the server from executing any received processes. This is generally not what you want, and is used only to debug the communication protocol up to the initial heap transfer.
- `-migrate_gdb` adds hooks to invoke `gdb` on the migrated process. Use this flag with caution, it is not well-documented and contains a number of developer quirks. You will not be able to run programs without user interaction if this switch is given.

2.3 Invoking system migration from an FC program

In FC, migration can be invoked using the `migrate` call. This call is defined in the standard header file `fc.h`, and has the following prototype:

```
static void migrate(char *destination) = "migrate";
```

The `destination` argument is a string which follows the migrate destination format, defined in section 2.1. When this call is executed, the process will attempt to migrate its entire state to the location specified in `destination`, and will then continue executing[2]. Currently, there is no way for the process to detect whether migration was successful from this call; future revisions will provide a status function which the process can use to determine the result of migration.

A simple example program is given below. In order to successfully use this program, you must first start a copy of the migration server using `mcc-migrated`. You should see the program migrate to the server, and return (on the server!) with code 42:

```
/* Trivial migration program */
#include "fc.h"

int main(int argc, char **argv) {
   migrate("host://localhost");
   return(42);
}
```

[1]For example, you cannot specify new optimization flags to use for the migrated program.

[2]Unless the suspend protocol is specified in `destination`; if that is the case, the process will "continue" when it is revived by the resurrection program.

2.4 Examples of using migration

The following are examples of how to use system migration.

2.4.1 Migrating from one machine to another

To migrate from machine `source.example.com` to machine `dest.example.com`, you must first start a migration server on `dest.example.com` as follows, using `mcc-migrated`. Assume MCC is installed to `/usr/local/mcc`:

```
dest.example.com# /usr/local/mcc/bin/mcc-migrated
Server listening on port 12354
```

The following FC program will migrate to `dest.example.com`:

```
/* Migrate to dest.example.com */
#include "fc.h"

int main(int argc, char **argv) {
   /* Place code to run on source.example.com here */
   migrate("host://dest.example.com");
   /* Place code to run on dest.example.com here */
   return(0);
}
```

Assuming this program is saved to `migrate.c` on the machine `source.example.com`, run the following commands:

```
source.example.com# /usr/local/mcc/bin/mcc -O3 -o migrate migrate.c
source.example.com# ./migrate
```

At this point, you should see the program migrate from `source.example.com` to the machine `dest.example.com`. Any code after the migration code will be run on the destination machine, and any output after migration will be sent to the console the migration server was run from.

2.4.2 Migrating to a nonstandard port on a machine

To migrate from machine `source.example.com` to a nonstandard port (e.g. port 9999) on machine `dest.example.com`, you must first start a migration server on `dest.example.com` using `mcc-migrated`. Assume MCC is installed to `/usr/local/mcc`:

```
dest.example.com# /usr/local/mcc/bin/mcc-migrated -port 9999
Server listening on port 9999
```

The following FC program will migrate to `dest.example.com`:

```
/* Migrate to port 9999 on dest.example.com */
#include "fc.h"

int main(int argc, char **argv) {
   /* Place code to run on source.example.com here */
   migrate("host://dest.example.com:9999");
   /* Place code to run on dest.example.com here */
   return(0);
}
```

Assuming this program is saved to `migrate.c` on the machine `source.example.com`, run the following commands:

```
source.example.com# /usr/local/mcc/bin/mcc -O3 -o migrate migrate.c
source.example.com# ./migrate
```

At this point, you should see the program migrate from `source.example.com` to port 9999 on `dest.example.com`. Any code after the migration code will be run on the destination machine, and any output after migration will be sent to the console the migration server was run from.

2.4.3 Suspending a process for later execution using `mcc-resurrect`

Processes can use the migration mechanism to save a checkpoint for later execution. To do this, processes should use the suspend protocol and the `mcc-resurrect` program. A sample program which suspends itself for later execution is given below:

```
/* Suspend process for later execution */
#include "fc.h"

int main(int argc, char **argv) {
   /* Place code to run before suspending here */
   migrate("suspend://migrate.state");
   /* Place code to run after suspending here */
   return(0);
}
```

Assuming MCC is installed to `/usr/local/mcc` and the source file is named `migrate.c`, run the following commands:

```
example.com# /usr/local/mcc/bin/mcc -O3 -o migrate migrate.c
example.com# ./migrate
```

At this point, any commands before the suspend call will be run, and then the file `migrate.state` will be written to the current directory[3]. The process may be revived on any machine and at any later time using this file. To revive the process, run:

```
example.com# /usr/local/mcc/bin/mcc-resurrect migrate.state
```

At this point, any remaining code in the file will be run, and the process will terminate normally. Note that you can take a process checkpoint using a similar procedure.

3 Implementation of System Migration

3.1 FIR implementation

Like all useful things in MCC, system migration requires a primitive in the FIR. System migration uses the SpecialCall mechanism in the FIR, which is designed to simplify manipulation of the FIR. SpecialCall takes an operation to perform, and a function application; once a special operation is performed, the function is called with its arguments. The syntax and semantics of a SpecialCall resemble TailCall, therefore most program manipulations can treat SpecialCall in the same way they treat TailCall. This helps to minimize the overhead of system migration in the FIR.

[3]An absolute path for the state file can be specified in the suspend protocol by using 3 slashes instead of 2; e.g. `suspend:///full/path/to/migrate.state`.

The exact form for system migration is given in the following ML type definitions:

type *exp*	::=	...	FIRexpressions
	|	SpecialCall of $debug\_label * tailop$	Special tail calls
and *tailop*	::=	...	Operations for special calls
	|	TailSysMigrate of $int * atom * atom * atom * atom$ list	System migration

The *tailop* TailSysMigrate $(id, dst\_base, dst\_off, f, args)$ will migrate to the location specified by the string in $dst\_base + dst\_off$, then call $f(args)$ after migration[4]. The *id* argument is an integer that is unique over all TailSysMigrate calls present in the FIR program, used to determine where execution should resume after a successful migration.

Note that *args* covers all live variables of the program, therefore all data that must be migrated is either contained in the heap or in variables indicated in *args*. This is useful in the implementation of system migration, since we do not have to perform a live variable analysis to identify variables which must be migrated.

3.1.1 Generation of initial special call

The FIR performs no special manipulation of special calls; special calls are generally constructed by the frontend during the IR-to-FIR transformation. In the FC frontend, the migration primitive is treated as an external function call with a well-known name until the IR-to-FIR transformation, where it is rewritten into a special call.

3.1.2 Marshal program data structure

Migration will marshal the FIR program, therefore at the end of the FIR transformations a marshallable version of the FIR must be constructed. A standard FIR program uses the `Fir.prog` data structure, but marshallable programs use the `Fir.mprog` data structure. The `Fir.mprog` data structure contains the full FIR program, and it also contains two lists of ordered symbols which correspond to `Mir.prog_fun_order` and `Mir.prog_global_order` in the MIR program (described in section 3.2).

Once constructed, the `Fir.mprog` data structure must never be modified. The `Fir.mprog` data structure is generated by the Fir_mprogmodule.

3.2 MIR implementation

In the MIR, system migration is translated from the SpecialCall form into a native MIR primitive called SysMigrate. SysMigrate resembles its SpecialCall form, however the function arguments are replaced with a single environment pointer to simplify the migration process. SysMigrate has the following type:

type *exp*	::=	...	MIR expressions
	|	SysMigrate of $int * atom * atom * var * atom$	System migration

The statement SysMigrate $(id, dst\_base, dst\_off, f, env)$ will migrate to the location specified in $dst\_base + dst\_off$, then call $f(env)$. *env* is a pointer to an aggregate memory block containing all live variables of the program (the *args* from TailSysMigrate). At this point, certain implementation details of system migration are exposed in the language, notably the use of a single atom to contain the live variables. Note that *env* is the only live variable in program across SysMigrate.

The translation from TailSysMigrate $(\ldots, f, args)$ primitive to SysMigrate $(\ldots, f', ienv)$ involves the following steps:

[4]Following standard convention for FC string pointers, the location pointer is stored in two variables, a base pointer in *dst_base* and an integer offset in *dst_off*.

- Generate a stub function f' which accepts a single argument, a pointer index to an aggregate block *ienv*. Convert the index into a real aggregate pointer *env*, with standard safety checks. Then, load all corresponding to *vars* from the *env* pointer into temporary variables. All standard safety checks are applied to the load operations, including bounds checking of the *env* pointer. The stub f' terminates in a call to f with the temporary variables as arguments.

- Generate MIR code to allocate the aggregate pointer *env*, and store all variables in *vars* into the *env* pointer. The normal bounds checking on *env* can be omitted here, since the allocation is guaranteed to be sufficiently large to hold all variables. The *env* pointer is converted into an pointer index *ienv*. The new MIR code terminates with the statement $\mathsf{SysMigrate}\,(\ldots, f', \mathit{ienv})$.

By storing all variables into an aggregate block in the heap, we simplify several problems in system migration:

- All data is stored in the heap at the time of migration, with the exception of the *ienv* aggregate index. This value is migrated explicitly as part of the context.

- 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 a map between register names across different architectures.

- Since all data is stored in the heap, all data has a standard, architecture-independent representation. Data in hardware registers may continue to have hardware-dependent representation without interfering with system migration. No real pointers exist in the data, therefore system migration does not need to construct a map between pointers across different machines.

- All safety checks are enabled in the stub function f', therefore the data in variables will be validated for accuracy. This would be difficult to perform if we attempt to migrate registers containing real pointers[5].

There are a few caveats to storing data in the heap. Notably, all registers for a process must be stored in the heap on each migration. While this is reasonable when the process is actually migrating elsewhere, 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 may be fairly expensive if the process intends to take frequent checkpoints.

Also, since all pointers are stored in the heap using indexes, we 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; however, the order of program globals and the function closure table must be preserved across migration[6]. Unfortunately, the sort order of symbols is not preserved when a FIR program is marshalled to another machine.

To resolve the issue of symbol ordering, the MIR maintains two fields to keep track of symbol ordering: `Mir.prog_global_order` is an ordered list of all program globals, and `Mir.prog_fun_order` is an ordered list of all functions which may escape. When a program migrates to another machine, these ordered lists are preserved so the global and function closure tables will be emitted in the same order. Note that the stub functions escape by definition, therefore they must be inserted into the `Mir.prog_fun_order` list in a well-defined order. Typically, a stub function f' is inserted into the `Mir.prog_fun_order` list immediately after the entry for the function f that it calls.

[5]It is for this reason that we migrate the *env* pointer as an aggregate index as well. It would be difficult to verify *env* is a real pointer on the destination machine, and we would also have to worry about translating mapping pointers between two different machines.

[6]The program globals are stored in the context, and referred to by index. The function closure table is the table of all valid function pointers; each function which *escapes* is allocated an entry in the function pointer table. If this table is reordered, then all function pointers in the heap will become invalid.

3.2.1 Generation of MIR primitive

Due to the complexity of the FIR-to-MIR transformation, the MIR preserves the FIR SpecialCall primitive until an early stage in the MIR transformation. The Mir_special module is responsible for rewriting all SpecialCall forms into the appropriate MIR primitives, such as SysMigrate. Notably, Mir_special generates the stub function and packages all arguments into the environment block.

3.3 Backend/runtime implementation

The implementation for the Intel IA32 backend is discussed here. Other backends should use a similar mechanism for implementing system migration.

3.3.1 The `Frame_type.migrate` data structure

Any program compiled by the IA32 backend is either a *normal* program or a *migrated* program. Normal programs are programs compiled by the user using the standard compilation process; these programs begin execution at the `main` function, like any standard program. Migrated programs are programs being compiled by `mcc-migrated` or `mcc-resurrect`; these programs use a nonstandard entry point that loads the process state from a network connection or state file, and resume execution where the original process left off.

Migrated programs are compiled with additional information that tells the runtime where to load the heap data from and where to resume execution. This information is specified in the `Frame_type.migrate` data structure. This data structure is not passed to the backend for normal programs.

The `Frame_type.migrate` data structure contains several significant fields, described below:

- `Frame_type.mig_key` is the string representation of the migrate key, or *id*, that determines where program execution should resume at.
- `Frame_type.mig_channel`, `Frame_type.mig_bidir` and `Frame_type.mig_initofs` describe the channel the runtime should use to read heap data from. This may contain a file descriptor for a UNIX file or a network connection. If the channel is bidirectional, the runtime can reply to the sender to verify data integrity.
- `Frame_type.mig_registers` is the index for the block containing the program registers, or *env*.
- `Frame_type.mig_ptr_size` and `Frame_type.mig_heap_size` indicate the sizes of the pointer table and heap, so the runtime can allocate sufficient space before receiving the heap data.
- `Frame_type.mig_debug` contains miscellaneous debugging flags for the migrate runtime.

When the IA32 backend is given the `Frame_type.migrate` data structure, then the compiled program will automatically start in the migrate runtime, instead of invoking the program's `main` function. This data structure is also written out as part of the code segment of a migrated program.

3.3.2 Transforming SysMigrate to assembly code

The MIR primitive SysMigrate (*id*, *dst_base*, *dst_off*, *f*, *env*) is translated into assembly code in the IA32 backend by the X86_mir module. Most of system migration is implemented in the runtime, so most of the assembly code deals with calling the migrate runtime and generating the "key", or the label that the migrate runtime will jump to on a successful migration. The key is derived from *id*, and is the same key that will be stored in `Frame_type.mig_key` when the migrated program is compiled on the destination machine.

The pseudocode for the assembly code generated is below:

`migrate_begin:`

1. Push *dst_off*, *dst_base*, *env*, *id* onto the stack.
2. Push a pointer to the section containing binary FIR onto the stack.
3. Call the function "`__migrate`" in the runtime.
   If migration is successful, this call will not terminate.
   If migration fails or checkpoint protocol used, execution resumes.
4. Clear the stack arguments from call to runtime.
5. Jump to `migrate_continue`.

`migrate_key:`

1. Clear stack arguments from entry.
2. Jump to `migrate_continue`.

`migrate_continue:`

1. Call $f(env)$.

Normal programs enter at `migrate_begin`. A migrated program enters to the runtime; once the heap is reconstructed, the program execution resumes at `migrate_key`, which is a label derived from the *id* value.

3.3.3 Runtime support for migration

The runtime support for migration consists of two parts: a *client* which is used by a program which wants to migrate, and a *server* which is called by a migration server after a migrated program has been compiled. The migration runtime is implemented in `arch/x86/runtime/x86_migrate.c`.

The migration client runtime is responsible for sending the binary FIR code and the heap data to the destination machine, or to a state file. The following actions are performed by the migration client[7]:

1. Send binary FIR representation.
2. Send migration key, *id*.
3. Send register index, *env*.
4. Send information on the pointer table and heap sizes*.
5. Wait for destination to compile program.
6. Send pointer table contents and heap contents.
7. Send global data blocks.
8. Verify data with server*.

The migration server runtime has a simpler task. By the time the runtime is invoked, the program has already been compiled, therefore it only needs to reconstruct the heap data. Also, the migration key *id* and register index *env* are given as part of the `Frame_type.migrate` data structure.

1. Allocate initial pointer table and heap.
2. Receive and reconstruct pointer table contents and heap contents.
3. Receive and reconstruct global data blocks.
4. Verify data with client*.

3.3.4 Migration server and resurrection support

The core of `mcc-migrated` and `mcc-resurrect` is implemented in `net/migrate/migrate.ml`. This code is responsible for parsing the first part of a migration state (either from a state file or network connection) which contains the binary FIR and `Frame_type.migrate` data structure. The migration server and resurrection programs perform the following steps:

[7] The steps marked with * are omitted when the process state is being written to a file; these steps only make sense when there is communication with a migration server.

1. Read the binary FIR from the channel.
2. Read and construct parametres for the `Frame_type.migrate` data structure.
3. Compile the binary FIR to native code, using `Frame_type.migrate`.
4. Run the program. Because the program is compiled with an initialised `Frame_type.migrate`, it should automatically enter the migrate server runtime.

Any migrated process will automatically inherit the console of `mcc-migrated` or `mcc-resurrect`.

4 Safety of Migrated Programs

Safety is not currently discussed in this paper, but will be added in a later revision of this paper.

Index