Thread Migration in the Presence of Pointers*

David Cronk
Computer Science Department
College of William and Mary
Williamsburg, VA 23187
cronk@cs.wm.edu

Matthew Haines
Computer Science Department
University of Wyoming
Laramie, WY 82071
haines@uwyo.edu

Piyush Mehrotra
ICASE
NASA LaRC
Hampton, VA 23681
pm@icase.edu

Abstract

Dynamic migration of lightweight threads supports both data locality and load balancing. However, migrating threads that contain pointers referencing data in both the stack and heap remains an open problem. In this paper we describe a technique by which threads with pointers referencing both stack and non-shared heap data can be migrated such that the pointers remain valid after migration. As a result, threads containing pointers can now be migrated between processors in a homogeneous distributed memory environment.

*This work was supported by the National Aeronautics and Space Administration under NASA contract No. NAS1-19480 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001.

1 Introduction.

A *thread* as a straightforward concept is a single independent sequential flow of control. A normal Unix process can generally be thought of as a single thread. Within a thread there is a single point of execution at any instant. Having multiple threads means that at any instant there are multiple points of execution, one for each thread [2].

Threads are generally classified as "heavyweight", "middleweight", or "lightweight". A thread's *weight* corresponds to the amount of context associated with the thread. A thread's context consists of a stack, program counter, machine registers, and other control information needed for its execution [16]. A typical Unix process represents a heavy weight thread. Many contemporary operating system kernels, such as Mach, allow multiple threads within a single address space, reducing the size of a thread's context. However, the context of the thread and all thread operations are controlled by the kernel, and often include more context than the application needs. These kernel level threads represent middle weight threads. Exposing all context and thread operations to the user-level allows for a minimal context, and thread operations can avoid crossing the kernel interface. These user-level threads have a much smaller context than kernel-level threads and represent lightweight threads. Due to their smaller context, lightweight threads have a much shorter context switch time (the time it takes to switch control of the processor from one thread to another) than either heavy or middle weight threads. Unless otherwise specified, the use of the term "thread" in this paper refers to lightweight threads.

Lightweight threads have become increasingly popular over the past few years. This is particularly true for threads running in a distributed memory environment [1, 10, 11]. Uses of threads in a distributed environment include, but are not limited to, providing latency tolerance by overlapping communication and computation and providing support for dynamic load balancing [14]. Dynamic load balancing involves determining the workload on each processor at runtime and transferring work from one processor to another to accommodate a load imbalance. In a multithreaded environment, this movement of work is referred to as *thread migration*, where a thread from one processor is moved, or migrated, to a remote processor.

There are at least two separate models of thread migration. The first model migrates threads by transferring the data that defines a thread's computation, but little or none of the thread state [7]. This occurs when threads are migrated before they begin execution, or at very well-defined break-points when the amount of state is minimal. One example of such a break-point is the end of a main loop of computation. Migration is accomplished by sending the data and the minimal state information to a remote processor where a new thread is created. The result is a very coarse-grained approach to load balancing, often leaving the system imbalanced for a considerable amount of time between break-points.

A second model, and the one in which we are interested, supports fine-grain load balancing by allowing a thread to migrate at arbitrary points during its execution. In this model, the current state and data of the thread to be migrated are sent to a remote processor. On the remote processor a new thread resumes execution at the point where the migrating thread was suspended. This model allows for much better load balancing because a thread can be migrated at arbitrary suspension points. However, this model requires the ability to migrate the entire state of a thread.

This paper describes the design of a novel approach for implementing dynamic thread migration in the presence of pointers to both heap and stack data. This design is intended for a homogeneous environment with a SPMD programming model. This ensures that the instructions for a thread reside in the same memory locations on each processor, and no code movement is required. We describe not only the actual design of the system, but also motivating factors that helped guide the decision making process and the pitfalls encountered along the way.

One of the most difficult problems while migrating the state of a thread is dealing with pointers in the migrant thread. This is difficult for a number of reasons. If the pointers refer to data in the thread's stack, then they will only remain correct if the stack is placed in the same memory location on the destination processor as on the source processor. Providing this assurance can be very difficult and inefficient. A similar situation exists for pointers that reference data in the heap.

There are several "solutions" that have been proposed for this problem. Some systems do not allow the use of pointers in migratable threads, while others allow pointers to become undefined following migration [15]. Both these "solutions" restrict the use of such common data structures as linked lists and trees, and are not considered practical.

Another solution is to perform memory allocations identically on all processors, reserving the memory locations for each thread and its associated data in case a thread must migrate [5, 6]. In this approach, when a thread migrates, all its associated data can be stored in the same memory location on the destination processor as on the source processor. This results in severe memory restrictions in the system. Moreover, the number of threads is limited by the memory capacity of a single processor, regardless of the total number of processors.

Some systems use a specialized type of pointer, a "global" pointer [10]. In this scheme a "pointer" is a data structure that defines both the processor on which that data resides, and a local pointer to the data that is valid within the specified processor. This requires a more complex data access mechanism in which the owner of the data must first be determined and, if it is not local, a remote data access request must be made. In addition to requiring more remote data accesses than should be necessary, this approach also requires *at least one level of indirection for each local memory access* through a global pointer. This overhead can cause dramatic performance degradation, particularly in a tight inner loop.

In this paper we explore a more general approach which allows dynamic thread migration at arbitrary suspension points, direct-access pointers for both heap and stack data, and the flexibility to relocate stack and heap data at different addresses on different processors. Our design requires keeping track of all dynamically allocated memory in such a way that all the data can be transfered to the destination processor, as well as keeping track of all pointers so that their values can be updated upon migration to reflect the new data locations.

The remainder of this paper is organized as follows: In Section 2 we discuss related work and Section 3 describes the functional requirements of the system. Section 4 introduces the actual design of the system while Section 5 discusses our reasons for making certain design decisions as well as some pitfalls which still exist. We wrap up with conclusions and proposals for future work in Section 6.

2 Related Research.

There has been a large amount of work done on thread migration. An effort at the University of Utah evolved the Mach operating system to a migrating threads model [8, 9]. However, Mach threads are kernel level threads as opposed to user level threads. In this work thread migration is used as a mechanism for RPC. Additionally, thread migration is only supported for local RPC. That is a thread can only migrate to a task (process) resident on the same node (processor).

The Amber system [6] also supports thread migration. This migration is supported by the static preallocation of thread address spaces on all machines. When a thread migrates in Amber it is doing so to access some remote object, and it will occupy the same address on the destination machine as on the source machine. As stated earlier this causes severe memory restrictions in the system.

The UPVM system [5] utilizes User Level Processes (ULPs), which are similar to threads. It supports migration using a strategy similar to the one presented in this paper, with the important distinction

that, similar to Amber, it allocates space for each ULP on every processor. In UPVM a private heap is maintained for all ULPs and all memory allocations are made from the heap associated with the calling ULP. Space for not only the ULP's stack, but also its heap, is maintained on each processor. When a ULP migrates to another processor it can be in the same memory location, and all pointers remain valid. As with Amber this will cause severe memory limitations.

Another system supporting thread migration is PM^2 [15]. In this system only a thread's context is moved, which includes the thread's stack. This means that dynamically allocated memory is not moved with the thread. This results in pointers into the heap becoming undefined following migration.

One of the most aggressive thread migration systems currently being worked on is the Ariadne system at Purdue [14]. In Ariadne a thread can be migrated for either remote data access or for improved load balance. Pointers referencing data within the thread's own stack can be updated by the explicit call to a user-level function. This allows local pointers to remain valid following migration. However, pointers referencing heap data cannot be updated because this data is not migrated with the thread. This causes these pointers to become undefined following migration and only remote data access can resolve the situation.

Emerald [3, 4, 13] is a distributed object-based language and runtime system. The primary goal of the designers is to experiment with the use of mobility in distributed programming. While the Emerald system uses some ideas similar to the ones presented in this paper, such as pointer translation, there is one main difference. The Emerald system has complete control of the language and uses compiler techniques to aid in object migration. The system described in this paper is designed as a runtime library intended to run on top of existing languages such as C. Our system also uses a different technique to keep track of and translate pointers following migration.

3 Functional Requirements.

In this section we outline the functional requirements of thread migration in the presence of pointers. In short, we address the different types of pointers that can exist in a multithreaded system and the ways in which these pointers can be manipulated.

3.1 Pointers to Private Data

When discussing private data we are referring to data which is being referenced by a *single* thread. The pointers themselves can reside in either the thread's stack or within the heap, and they may reference data located in either the stack or heap. This results in four possible pointer types, as depicted in Figure 1:

1. A pointer located in the thread stack and referencing a data item in the stack.
2. A pointer located in the thread stack and referencing a data item in the heap.
3. A pointer located in the heap and referencing a data item in the thread stack.
4. A pointer located in the heap and referencing a data item in the heap.

3.2 Pointers to Shared Data

We now consider the role that pointers play in sharing data between two or more threads. Sharing data between threads can occur in two ways. The first is the use of pointers to global data. By global

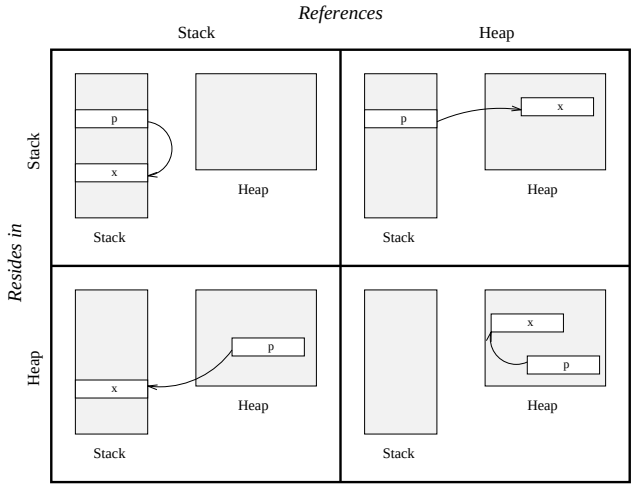

Figure 1: **Examples of the four types of pointers**

data we mean data that is declared statically by the process, outside any threads or functions. Since all threads within the process have access to this data, any thread can use a pointer to reference it. The system must ensure that all such pointers continue to reference the same data following migration, even though the values in these memory locations may not be consistent (as in the case of processor-specific data stored in global variables). A second method of sharing data between threads occurs when pointers from two or more threads reference the same location in the heap.

While the first method for sharing data can be accommodated in a relatively straightforward fashion, the second method is considerably more difficult and we plan to address this issue in future work.

3.3 Pointer Manipulation

The ways in which a pointer can be manipulated can be subdivided into three main categories: assignments, updates, and pointers as formal arguments. A general solution to the problem of migrating threads containing pointers should accommodate all three of these categories.

Assignment refers to changing the value of a pointer. This includes assigning to the pointer the address of newly-allocated memory, the value of another pointer, or the address of a variable. Since pointer assignment is such a common operation, it is very important to minimize the overhead associated with these operations. For example, traversing a linked list via an auxiliary pointer performs at least one pointer assignment for each element in the list.

Updates refer to changing the value of the data that the pointer references. It is important that all such updates survive thread migration. That is, it is not sufficient to ensure a pointer references a valid memory location following migration, but that the value referenced by a pointer following migration is identical to the value that was referenced before migration.

The integrity of pointers as *formal arguments* must also be protected. For example, consider a thread that is migrated while invoking a function call, and which has at least one pointer as a formal argument. If the function later updates the pointer, the data being referenced must be the same as it was before migration.

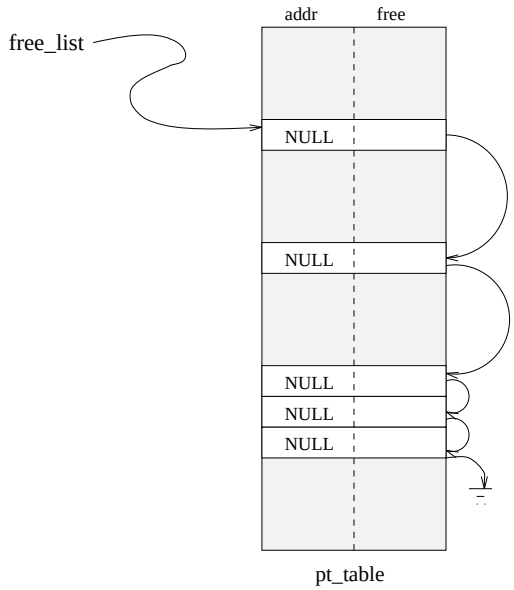

Figure 2: **Example of the `free_list` for `pt_table`**

4 Design.

This section presents the design of a new thread migration system. Since the focus of this paper is on pointer manipulation, we omit the other details of thread migration, such as selection of the thread to be migrated, selection of the destination processor, and how the actual sending takes place.

The design revolves around two basic concepts. The **first** of these concepts is the use of a private heap for each thread. This private heap is allocated from the processor's heap as a contiguous block of memory at thread creation time. All dynamic memory allocation performed by the thread occurs within the private heap associated with the thread. This facilitates the calculation of offsets for the purpose of updating pointers following migration, and allows us to migrate all of the thread's heap data in a single message. The **second** concept is keeping track of all valid pointers which allows all of them to be updated following migration.

This design for thread migration requires the addition of a few new data structures and user-level functions which we present below.

4.1 Auxiliary Data Structures

To ensure that heap variables reside in a contiguous block of memory that can be easily moved, each thread uses a private heap. This thread heap is allocated from the process heap space, where the size can be specified by the user at thread creation time. An efficient memory allocation scheme is used within this heap, the details of which are irrelevant to this paper.

It is also necessary to keep track of all the pointers being used by the thread, and this is done using a table, the *pt_table*, whose entries consist of two fields: `addr` and `free`. The `addr` field is used to hold the absolute address of the pointer itself, and the `free` field is used to facilitate the creation of a free list of table entries. The table itself (depicted in Figure 2) is statically allocated at the front of the thread heap during thread creation. Again, the size of the table can be controlled by the user at the time of thread creation.

```
int pt_register(q)
   void *q;
{
   int i = find_free_entry();
   pt_table[i].addr = q;
   return i;
}
```

(a)

```
int pt_release (q)
   void *q;
{
   int i = find_index(q);
   pt_table[i].addr = NULL;
   add_to_free_list (i);
   return i;
}
```

(b)

Figure 3: **Code for (a) registering and (b) releasing pointers**

Finally, the thread control block (*tcb*) of the thread to be migrated needs to be supplemented to contain an additional four fields: pointer to the base of the thread's private heap, the size of the thread's heap, the size of the thread's `pt_table`, and the head of the free list. Most user-level thread packages, including the pthreads standard [12], provide a mechanism for adding information to the context of a thread via `setspecific/getspecific` function calls, so adding these four values should not be difficult.

4.2 Auxiliary User-level Functions

Our thread migration system relies on the fact that the runtime system knows about all active pointers within a thread. Therefore, a mechanism is needed that allows all active pointers to be registered. Without compiler support we assume that the user will insert the pointer registration calls, though it would not be difficult to modify a compiler to insert the calls automatically.

To register a pointer, a free entry in the `pt_table` must be found and the `addr` field of this entry set to the address of the pointer being registered. Code for this function is depicted in Figure 3(a), where the function `find_free_entry` locates and removes the next element from the free list. The system requires that users register all pointers before they are used, including pointers that are formal parameters to functions and pointers in dynamically allocated data structures.

In conjunction with a registration function, a function is needed to *release* pointers when they are no longer being used or when they go out of scope. This prevents the `pt_table` table from overflowing and prevents stack memory from becoming corrupted following thread migration, as will be explained later. When a user releases a pointer, the pointer must be found in the `pt_table` and added to the free list. Code for the release function is depicted in Figure 3(b).

While adding an element to the free list is straightforward, finding the element from its address requires a search. A simple linear search will cause on average $n/2$ comparisons per release where n is either the size of the table or the highest index of a valid pointer, whichever is smaller. However, searching and hashing techniques can be employed to reduce this overhead to near constant time.

Finally, thread-specific `malloc` and `free` routines for managing memory in the thread's heap are also needed. Code for the former function is given in Figure 4.

4.3 The Migration System

The thread migration system is divided into two components, one for the source processor to prepare a thread for migration and another for the destination processor to prepare for receiving a migrating thread.

Source processor:

```
void* thr_malloc (size)
    int size;
{
   thread_t *t = thread_self();
   void* heap = find_heap(t);
    void* p = find_block (size, heap);
    /* This will include memory managment
          within the thread's private heap */
    return p;
 }
```

Figure 4: **Code for thread specific malloc**

1. Send a message to the destination processor indicating that a thread is about to be migrated. Included in this message is the sizes of both the stack and heap of the migrant thread.

2. Send the thread's private heap, `pt_table`, and stack to the destination processor.

3. Free resources associated with the thread (stack, heap, etc).

Destination processor:

1. Receive a message that specifies the memory requirements (for both stack and heap) of an incoming thread.

2. Allocate space for the heap and stack of the thread.

3. Receive the thread's heap and stack directly into the newly allocated heap space, thus avoiding a user space buffer copy.

4. Update the `pt_table` and the value of all valid pointers.

5. Put the new thread on the ready queue.

We expand on step 4 below.

To update the `pt_table`, each valid entry in the `pt_table` (`addr` != NULL) is examined. By knowing both the base of the old stack and heap (available from the thread control block) as well as the base of the new stack and heap, the `addr` field of the `pt_table` entry can be used to calculate the new address of the pointer. This new address is then assigned to the `addr` field of the table.

The values stored in the pointers themselves also need to be updated, since the location of the data that they reference has now changed. Since the new address of the pointer has been computed, the pointer can be accessed directly and its current value, i.e., the value on the old processor, can be examined. The following scenarios are possible:

- If the pointer is NULL, then it is not referencing anything and should be left alone.

- If the pointer references global data, then nothing is done since it is assumed that the global data is located at the same addresses on all processors.

- If the pointer references data in the stack or heap, then the same address computation that was done for the pointer addresses is applied, and the value of the pointer updated.

An example of this calculation is shown next. Let a thread migrate from processor A to processor B, with the following pertinent values:

Processor A	Processor B
• heap base: 1000	• heap base: 1500
• stack base: 200	• stack base: 550
• pointer address: 226	• pointer address: 576
• pointer value: 1013	• pointer value: 1513

The pointer has an offset of 26 (226-200) from the base of the stack. This translates to an address of 576 (550 + 26) on processor B. The system now accesses memory location 576 on processor B and retrieves the value 1013. This is an offset of 13 (1013-1000) from the old base of the heap, and translates to a new value of 1513 (1500 + 13) on processor B. The system now updates memory location 576 with the value 1513, and the pointer in question now references the correct data.

5 Discussion.

This section begins with a discussion of the rationale for the design decisions which were made, including things that can go wrong if some of the steps from the system are omitted. This is followed by a discussion of some of the potential pitfalls which still exist in the design.

5.1 Rationale

One of the fundamental problems in dealing with pointers during thread migration is that data can potentially reside in different memory locations following migration. Thus it is necessary to update pointer values following migration to ensure the pointers are still referencing the correct data items. The only way to do this is to keep track of all pointers in use so they can be updated. If all pointers were established via `malloc`, then a special `malloc` function could be created to register all pointers in use. However, many pointers are created without ever calling `malloc`, e.g. data being passed into functions as reference parameters, and all these pointers must also be updated following migration. Therefore another method of detecting the definition of a pointer is needed. One way would be to use compiler support, where the compiler uses a combination of its symbol table and use-def chains to locate the definition of all live pointers. These pointers could then be recorded in the `pt_table` as previously discussed. However, modifying compilers to provide this new capability is typically not a practical option. In the absence of compiler support the user is required to inform the runtime system when a pointer is in use.

Given that all active pointers are to be registered, we considered whether it is necessary for these pointers to be released once they were no longer in use. Other than releasing pointers to free up space in the `pt_table`, is it necessary? The answer is yes.

Consider the scenario as depicted in Figure 5. Some thread invokes a function `foo`, which causes a stack frame for `foo` to be added to the thread's stack. Lets further assume that `foo` makes use of a pointer `ptr` that resides on the thread stack, within `foo`'s stack frame. `ptr` is registered, which creates an entry in the `pt_table`. Function `foo` then returns without releasing `ptr`, which means there is still

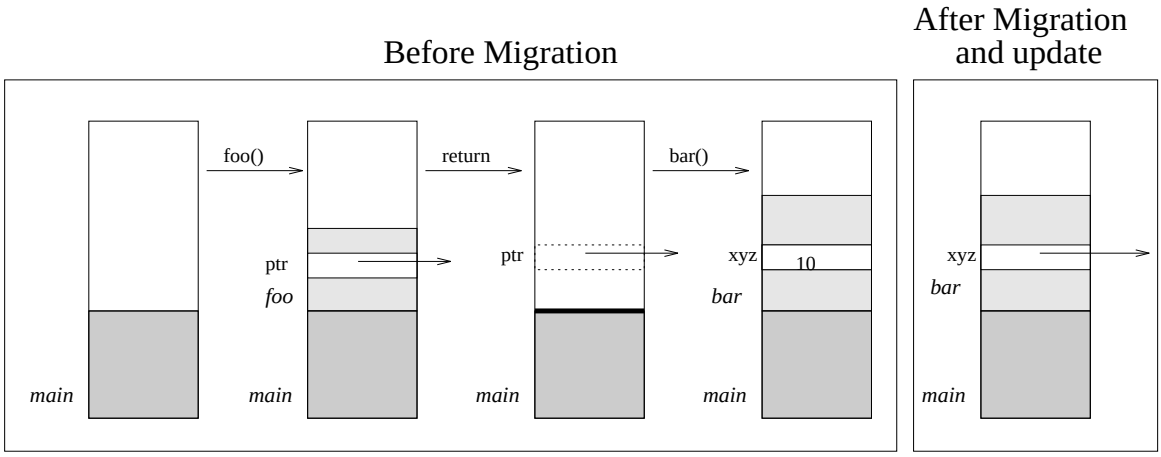

Figure 5: **Example of what can happen if a pointer is not released prior to going out of scope**

an entry in the `pt_table` for `ptr`. The thread then makes a call to the function `bar`, causing the stack frame for `bar` to at least partially overlap the memory of the former stack frame for `foo`. Assume that an integer variable, `xyz`, is now located in the same memory location where `ptr` used to be. Now suppose the thread is migrated. Following the migration all the pointers must be updated. Since the pointer `ptr` was not released, the system will try to update its value. However, this is now the value of `xyz` and if this value is updated the value of `xyz` will be erroneously modified.

Now that it has been shown that it is necessary to release all pointers, the method for doing so is discussed. To release a pointer a search is performed to locate the pointer in the `pt_table`, set the `addr` field to NULL, and insert the entry onto the `free_list`. The `addr` field is set to NULL so that when the table entries are processed after migration there is a quick method for determining which entries are valid without having to chain through the free list.

This brings up the question of using the `free_list` in the first place, as it adds another field to the `pt_table`, thus increasing both memory and communication requirements. It was decided, however, that the advantages far outweigh the disadvantages. The main advantage of the free list is the ability to find a free entry from the table in constant time, rather than needing to search through the table each time a new pointer is registered. The size of the `pt_table`, even with the extra field for a free list index would be much smaller than the size of the thread stack and heap. This means that the additional memory usage associated with an extra field should be minimal, relative to total memory used.

Storing the `pt_table` at the front of the heap allows the shipping of the `pt_table` and heap as one contiguous block of memory, lowering communication costs. It is also possible to send all the information needed to complete the migration in a single message. However, to avoid copying data it was decided to send an initial message to the destination processor informing it that a thread is being migrated to its location, followed by the stack and heap itself[1]. This allows the destination processor to pre-allocate the stack and heap. It can then receive the stack and the heap directly into this pre-allocated memory without an intermediate system buffer copy.

[1] We present it this way for the sake of simplicity. In an actual implementation the stack and heap would be allocated in a contiguous block of memory, allowing a single send-receive pair to transport both the stack and the heap.

5.2 Pitfalls

One must consider the drawbacks of using a system such as this. A primary drawback of this system is that, without compiler support, it requires the user to register and release all pointers. This complicates programming slightly, though the addition of registration and release calls is fairly mechanical. However, if the registration and releasing of pointers is not performed properly, incorrect results may be obtained.

We have already discussed the problem of not releasing pointers. Another problem arises when a user accidentally registers the same pointer twice, which causes a similar situation. In this case there will be two entries for the same pointer in the `pt_table`. This causes the pointer to be updated twice following a migration. When the first update occurs, the new value will be correct. When the second entry for the pointer is reached in the table, the value will be updated again, which will cause it to reference the wrong memory location. If the values of the pointers were recorded prior to migration and this stored value was used for the update then this problem would be averted. However, this would cause considerable processing overhead on the source processor prior to migration. To ensure correct execution of the migration system, each pointer must be registered exactly once prior to being used (defined), and must be released after it is no longer being used or prior to going out of scope. Failure to follow these rules can result in corrupted data.

A more problematic pitfall involves the migration of a thread during its entry into a function. For instance, let a function be called where a pointer is passed as a parameter. Upon function invocation, this parameter will reside on the stack. If the thread migrates before the call to `pt_register` has been invoked, the pointer will not be registered and its value will not be updated following migration. This will, of course, cause erroneous behavior. This demonstrates the need for *critical sections* during which a thread must not be allowed to migrate.

Another drawback is that the amount of heap space for a thread is now limited. While the heap size in most threads systems is limited only by the processor's memory, this system requires the user to limit a thread's heap size at the time of thread creation. Also, due to internal fragmentation of the private heaps there is less memory available to the system as a whole. We are currently looking at a design which would allow *heap chunks* to be allocated and chained together when space on the thread's heap is exhausted.

6 Conclusions.

Dynamic thread migration is an emerging technology that supports both data locality and dynamic load balancing. However, despite its usefulness, supporting pointers during thread migration has received little attention.

This paper introduces the design of a novel approach which provides support for thread migration in the presence of pointers. We have outlined the steps required to ensure that all pointers are updated properly following a migration. This includes pointers that are located in either the stack or heap and which reference data in either the stack or heap. The system was designed with efficient implementation in mind.

Future work will improve both the efficiency and functionality of our design, and compare it with existing systems that offer dynamic thread migration. We also plan to devise a better way of maintaining our pointer table to allow for a more efficient searching method. The use of a hash tables for this purpose is currently being investigated. It is not clear at this time if this will improve efficiency due to the increased overhead associated with adding pointers to the table and the fact that deletions from hash tables are less efficient in their own right.

We also hope to improve our design to allow for better sharing of data. The system should allow threads to share both stack and heap data and for this sharing to survive migration.

References

[1] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pages 213–226, San Diego, CA, September 1993. USENIX.

[2] A.D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Equiptment Corporation, January 1989.

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, Portland, OR, October 1986.

[4] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987.

[5] J. Casas, R. Konuru, S. Otto, R. Prouty, and J. Walpole. Adaptive load migration systems for PVM. In *Proceedings of Supercomputing '94*, pages 390–399, Washington D.C., November 1994. ACM/IEEE.

[6] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield. The Amber system: Parallel programming on a network of multipocessors. In *ACM Symposium on Operating System Principles*, December 1989.

[7] N.P. Chrisochoides. Multithreaded model for dynamic load balancing parallel adaptive PDE computations. Technical Report CTC95TR221, Cornell University, October 1995.

[8] B. Ford, M. Hibler, and J. Lepreau. Notes on thread models in Mach 3.0. Technical Report UUCS-93-012, Department of Computer Science, University of Utah, April 1993.

[9] B. Ford and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the Winter 1994 USENIX Conference*, January 1994.

[10] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An interoperability layer for parallel and distributed computer systems. Technical Report Version 1.3, Argonne National Labs, December 1993.

[11] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing '94*, pages 350–359, Washington D.C., November 1994. ACM/IEEE.

[12] IEEE. *Threads Extension for Portable Operating Systems (Draft 7)*, February 1992.

[13] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[14] E. Mascarenhas and V. Rego. Ariadne: Architecture of a portable thread system supporting mobile processes. Technical Report CSD-TR-95-017, Purdue University, March 1995.

[15] R. Namyst and J.F. Mehaut. PM^2: Parallel Multithreaded Machine. In *Proceedings of ParCo '95*, Gent, Belgium, September 1995.

[16] J. Pinakis. Remote thread execution. In *Proceedings of the 16th Australian Computer Science Conference*, pages 489–500, Brisbane, Australia, February 1993.