On Improving Thread Migration: Safety and Performance*

Hai Jiang[1] and Vipin Chaudhary[2]

[1] Institute for Scientific Computing, Wayne State University
Detroit, MI 48202 USA
haj@cs.wayne.edu

[2] Institute for Scientific Computing, Wayne State University
and Cradle Technologies, Inc.
vipin@wayne.edu

Abstract. Application-level migration schemes have been paid more attention recently because of their great potential for heterogeneous migration. But they are facing an obstacle that few migration-unsafe features in certain programming languages prevent some programs from migrating. Most application-level migration schemes declare or assume they are dealing with "safe" programs which confuse users without explanation. This paper proposes an application-level thread migration package, *MigThread*, to identify "unsafe" features in C/C++ and migrate this kind of programs with correct results. Therefore, users need not worry if their programs are qualified for migration as they experienced before. Besides the existing characteristics such as scalability and flexibility, *MigThread* improves transparency and reliability. Complexity analysis and performance evaluation illustrate the migration efficiency.

1 Introduction

Recent improvements in commodity processors and networks have provided a chance to support high-performance parallel applications within an everyday computing infrastructure. As high-performance facilities shift from supercomputers to Networks of Workstations (NOWs), migration of computing from one node to another will be indispensable. Thread/process migration enables dynamic load distribution, fault tolerance, eased system administration, data access locality and mobile computing [1, 2, 7].

Thread migration can be achieved at kernel, user, or application level. Kernel level thread migration is a part of the operating system. Threads are moved around among processors if they are on multi-processors, such as SMPs, or among workstations by distributed operating systems. Kernel-level migration is complicated, but efficient. User-level approaches move migration functionality from the kernel into user space and typically yield simpler implementations,

* This research was supported in part by NSF IGERT grant 9987598, NSF MRI grant 9977815, NSF ITR grant 0081696, US Army Contract DAEA-32-93-D-004, Ford Motor Company Grants 96-136R and 96-628R, and Institute for Scientific Computing.

but suffer too much from reduced performance and less transparency. User-level migration is targeted for long-running threads with few OS requirements, less transparency, and a limited set of system calls. Programs will need to re-linked with certain library to enable migration feature.

Traditional application-level migration is implemented as a part of an application. It achieves simplicity by sacrificing transparency and reusability. But it has an attractive potential for heterogeneous migration feature. As internet is popular and grid computing is emerging, heterogeneous migration will be indispensable. We have proposed an application-level thread migration scheme, *MigThread*, to improve transparency and reusability for migration [3]. A big impediment to thread migration is due to "migration-unsafe" features within C/C++. If the programmer uses these "unsafe" features in their programs, the migration leads to errors. To ensure the correctness, most application-level or language level migration schemes declare they only work on "migration-safe" programs. There are two problems with this. First, the "unsafe" features are not well defined. Second, such restriction greatly reduces the domain where migration can be utilized.

In this paper, we make the following contributions:

- Determine and overcome "migration-unsafe" features in programs to widen the applicability of thread migration.
- Improve existing thread migration scheme[3].
  - Speed up source-to-source transformation at compile time.
  - Handle pointers and pointer arithmetic efficiently.
  - Support better memory segment management.
- Provide complexity comparison and analysis, and performance evaluation on real applications.

The remainder of this paper is organized as follows: Section 2 describes the performance improvements to the existing one in [3]. Section 3 identifies and overcomes some migration-unsafe features in C. In section 4, we compare the complexity of our scheme with existing implementations and show experiment results on benchmark programs. Section 5 gives an overview of related work. We wrap up with conclusions and continuing work in Section 6.

2 Optimizing thread migration using *MigThread*

In this section we present some optimization to the migration scheme in [3].

2.1 *MigThread*

MigThread is an portable and scalable application-level thread migration package. It takes thread state out of kernel or libraries, and moves it up to the language level. *MigThread* consists of two parts: preprocessor and runtime support module. At compile time, the preprocessor scans the source code and collects related thread state information into two data structures which will be integrated

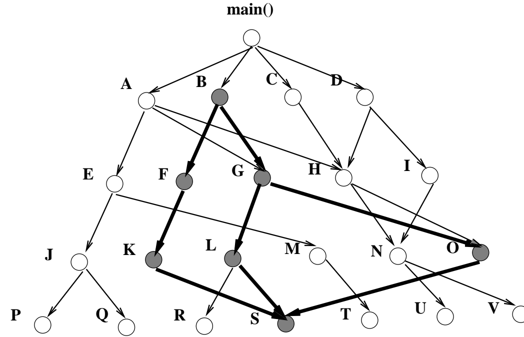

Fig. 1. Function call graph to reduce compile time overhead

into the thread state at runtime. Local variables, function arguments, and Program Counters (PC) are all parts of thread state. Dynamically allocated memory is also supported. Thus data in heap are migrated with the thread state and restored remotely. Since destination nodes might use different address spaces, pointers referencing stack or heap might be invalid after migration. *MigThread* detects and marks pointers at language level so that at runtime it just accesses predefined data structures to update most of them precisely. Adaptation points where thread migration can take place are detected, labelled, and pointed by **switch** statement [3].

At runtime, *MigThread* maintains a thread control area (TCA) which holds a record for each thread containing references to a stack, a control block for memory segments in the heap and a pointer translation table. During migration, thread state is constructed, transferred, and restored. After updating pointers, *MigThread* resumes computation at the right place. Since the physical thread state is transformed into a logical form, *MigThread* has great potential to be used in heterogeneous environments without relying on any type of thread libraries or operating systems. More design and implementation details are in [3].

2.2 Reducing compile time overhead

To reduce the overhead, *MigThread* only transforms the related functions invoked by the migrating threads. It creates a call graph starting out of **main()** to detect the thread starting function and the migration function. These two and all other functions between them should be transformed into the migration-enabled code. For example, in Fig. 1, **B** and **S** are the thread starting and migration functions respectively. There are three paths between them: **BFKS**, **BGLS** and **BGOS**. The program execution has to take one of them based on the runtime situation. Therefore, at compile-time, *MigThread* uses breadth first search to identify and transform related functions on these possible paths. Since only a fraction of the entire program is transformed, compile time overhead is greatly reduced (see Fig. 6 in Section 4).

2.3 Generalizing pointer handling

The scheme in [3] identifies pointer variables at language level and collects them into a data structure *sr_ptr*. If some structure type variables in *sr_var* contain pointer fields, they need to be referenced by new pointer variables in the other data structure *sr_ptr*. On the destination node, it scans the memory area of *sr_ptr* to translate most pointers. This strategy is more efficient than reporting pointers one-by-one as in Porch[6] and SNOW[7]. *MigThread* extends this model further to handle more complicated cases in dynamically allocated memory. If some pointer variables in *sr_ptr* contains pointer type subfields, the preprocessor just reports their offsets in base units and the runtime support module will detect other dynamic pointer fields by pointer arithmetic. *MigThread* does not trace pointers if programs are "migration-safe". No matter how pointers are manipulated, only the current values of variables and pointers hold the correct thread state. This "ignore-strategy" makes *MigThread* efficient. *MigThread* traces pointers only when "unsafe" features are involved as in Section 3.

2.4 Memory management optimization

Unlike the linked-list structure in [3], *MigThread* maintains a red-black tree of memory segment records, traces all dynamically allocated memory in local or shared heap, and provides the information for pointer updating. Each segment record consists of the address, size, and type of the referenced memory block, with an extra linked list of offsets for inner pointer subfields. In user applications, when **malloc()** and **free()** are invoked to allocate and deallocate memory space, the preprocessor inserts **STR_mig_reg()** and **STR_mig_unreg()** accordingly to let *MigThread* create and delete memory segment records at runtime. Since memory blocks are maintained in order, the insertion, deletion or searching of one node in the red-black tree takes $O(logN)$ time. Again, the dynamically allocated memory management is moved up to the application level [3].

3 Handling Migration-unsafe Features

Application level migration schemes rely on programming style to ensure the correctness of the resumed computation after migration. Most migration schemes declare that they only work on "safe" programs to avoid those "unsafe" features in C and obtain the correct thread state. *MigThread* can detect and handles some such "unsafe" features, including pointer casting, pointers in unions, library calls, and state-carrying instructions.

3.1 Pointer casting

Pointer manipulations can cause problems with migration. Pointer casting is one of them. It does not mean the cast between different pointer types, but the cast to/from integral types, such as integer, long, or double. The problem is

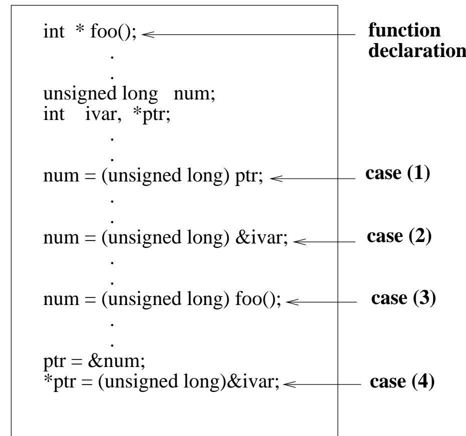

Fig. 2. Four cases of hiding pointers in integral type variables

that pointers might hide in integral type variables. Application level migration schemes identifies pointer values (or memory addresses) by pointer names or even types if they are in dynamically allocated memory segments. If pointers are cast into integral type variables, migration schemes might miss updating them when address space changes during migration. So the central issue is to detect those integral variables containing pointer values (or memory addresses) so that they could be updated during state restoration. Casting could be direct or indirect. There are four ways to hide pointers in integral type variables (shown in Fig. 2):

1. Cast pointers directly or indirectly. In Fig. 2, case (1) only shows the direct cast. If *num* is assigned to another integral type variable, indirect cast happens and it also can cause problems.
2. Memory addresses are cast into integral type variables directly.
3. Functions' returning values are cast in.
4. Integral variables are referenced indirectly by pointers or pointer arithmetic and their values are changed by all the above three cases.

To avoid dangerous pointer casting, *MigThread* investigates pointer operations at compile time. The preprocessor creates a pointer-group by collecting pointers, functions with pointer type return values, and integral variables that have already been cast in pointer values. When the left-hand side of an assignment is an integral type variable, the preprocessor checks the right-hand side to see if pointer casting happens. If members of pointer-group exist without changing their types, the left-hand side variable should also be put into pointer-group for future detection and reported to the runtime support module for possible pointer update during migration. The preprocessor ignores all other cases.

The preprocessor is insufficient for indirect access and pointer arithmetic as case (4) in Fig. 2. The preprocessor inserts primitive **STR_check_ptr(mem1, mem2)** to request the runtime support module to check if *mem*1 is actually

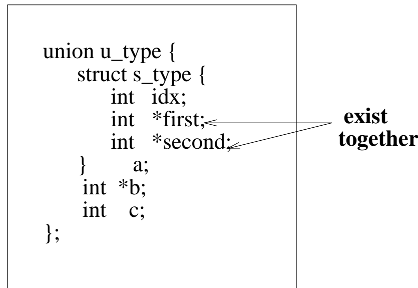

Fig. 3. Pointers in Union

an integral variable's address (not on pointer trees) and $mem2$ is an address (pointer type). If so, $mem1$ will be registered as a pointer which could also be deregistered later. Here $mem1$ is the left-hand side of assignment and $mem2$ is one member of right-hand side components. If there are multiple components on the right-hand side, this primitive will be called multiple times. Frequently using pointer arithmetic on the left-hand side can definitely cause heavy burden on tracing and sacrifice performance. This is a rare case since normally pointer arithmetic is applied more on the right-hand side. Thus, computation is not affected dramatically. During the migration, registered pointers will be updated no matter if their original types are pointer ones or not. *MigThread*'s preprocessor and runtime support module work together to find out memory addresses hidden in integral variables and update them for migration safety.

3.2 Pointers in Union

Union is another construct where pointers can evade updating. In the example of Fig. 3, using member a means two pointers are meaningful; member b indicates one; and member c requires no update. Migration schemes have to identify dynamic situations on the fly. Application-level migration schemes have advantages over kernel- and user-level ones. When a **union** variable is declared, the compiler automatically allocates enough storage to hold the largest member of the **union**. In the program, once the preprocessor detects a certain member of the **union** variable is in use, it inserts primitive **STR_union_upd()** to inform the runtime support module which member and its corresponding pointer fields are in activation. The records for previous members' pointer subfields become invalid because of the ownership changing of the **union** variable. We use linked list to maintain these inner pointers and get them updated after migration.

3.3 Library calls

Library calls bring difficulties to all migration schemes since it is hard to figure out what is going on inside the library code. Application-level migration schemes work on the source code. Without the source code of libraries, problems can

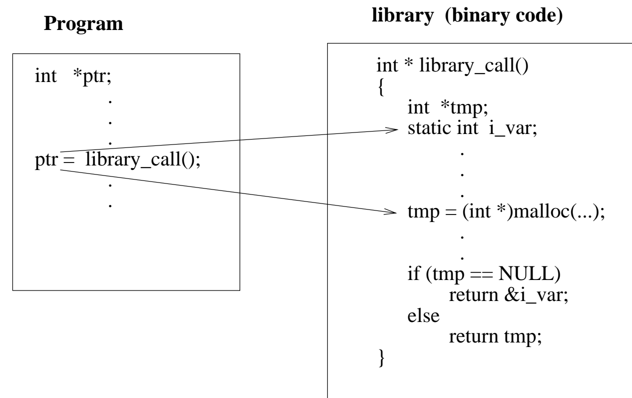

Fig. 4. Pointers dangling after library calls

occur when pointers are involved. For *MigThread*, the major concerns are static local variables and dynamically allocated memory. In the example of Fig. 4, the pointer *ptr* might be pointing to an address of static local variable *i_var* for which compilers creates permanent storage or a dynamically allocated memory block. Both of them are invisible to *MigThread*. Pointers pointing to these unregistered locations are also called “dangling pointers”, as those pointing to de-allocated memory blocks. This phenomena indicates that *MigThread* is unable to catch all memory allocations because of the “blackbox” effect. The current version of *MigThread* can inform users of the possible danger of “memory leakage” so that programmers can register these memory blocks by hands if they know the library calls well. This is one workaround solution whereas all other migration schemes have to face the same problem. For **malloc()** wrappers, another option is for users to specify the syntax so that the preprocessor can know how to insert proper primitives for memory management.

3.4 State-carrying instructions

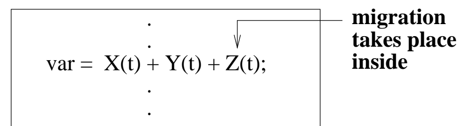

Fig. 5. Migration happens inside of single complex statement

Since *MigThread* works at language level, the adaptation points can only be inserted at this level. Thus, there is at least one C language statement between two adaptation points. It seems that the migration can only happen between

Table 1. Complexity comparison in data collecting

System	Collect Variables	Collect Pointers	Collect Memory Blocks	Save Variables	Save Pointers	Allocate Memory Blocks
Porch	$O(N_{var})$	$O(N_{ptr})$	$O(N_{mem})$	$O(N_{var})$	$O(N_{ptr})$	$O(N_{mem} * logN_{mem})$
SNOW	$O(N_{var})$	$O(N_{ptr})$	$O(N_{mem} * logN_{mem})$	$O(N_{var})$	$O(N_{ptr})$	$O(N_{mem})$
MigThread	1	1	$O(N_{mem})$	0	0	$O(N_{mem} * logN_{mem})$

Table 2. Complexity comparison in data restoration

System	Restore Variables	Restore Pointers	Update Pointers	Re-allocate Memory Blocks	Delete Memory Blocks
Porch	$O(N_{var})$	$O(N_{ptr})$	$O(N_{ptr} * logN_{mem})$	$O(N_{mem})$	$O(N_{mem} * logN_{mem})$
SNOW	$O(N_{var})$	$O(N_{ptr})$	$O(N_{ptr} * N_{mem})$	$O(N_{mem})$	$O(N_{mem}^2)$
MigThread	1	1	$O(N_{ptr} * logN_{mem})$	$O(N_{mem})$	$O(N_{mem} * logN_{mem})$

statements. But functions break this rule and enable migration to take place within single statement as the example in Fig. 5. The right-hand side is a summation of three functions' results. Suppose thread migration takes place inside the third function $Z(t)$. We assume that the functions are executed in order. Before migration, the compiler saves the results of the first two functions in some temporary storages which are useful only if they are under *MigThread*'s control. To achieve this, the statement should be broken down and temporary variables are introduced to save temporary results. The advantage of this is that we increase the adaptation points. But this brings minor changes to the program's structure.

To avoid modification of programs, the first two functions could be rerun to retrieve their return values. This means they should be "re-entrant" and deliver the same result with the same inputs. *MigThread* has to detect "state-carrying" functions and make them "stateless" and label the position right before each **return** statement. During the re-running of the first two functions after migration, their **switch** statements dispatch computation directly to their last escape points. Therefore, no actual computation goes through function bodies and functions become stateless.

4 Complexity Analysis and Performance Evaluation

Besides *MigThread*, there are two other application level migration systems, Porch [6] and SNOW [7] which collect and restore variables one-by-one explicitly at each adaptation point in time $O(N)$. This makes it hard for users to insert adaptation points by themselves. Our *MigThread* only registers variables once in time $O(1)$ and at adaptation points the programs only check for condition variables. Therefore, *MigThread* is much faster dealing with thread state.

For memory blocks, Porch and *MigThread* have similar complexity because they both maintain memory information in red-black trees. SNOW uses a memory space representation graph, which is quick to create a memory node, but

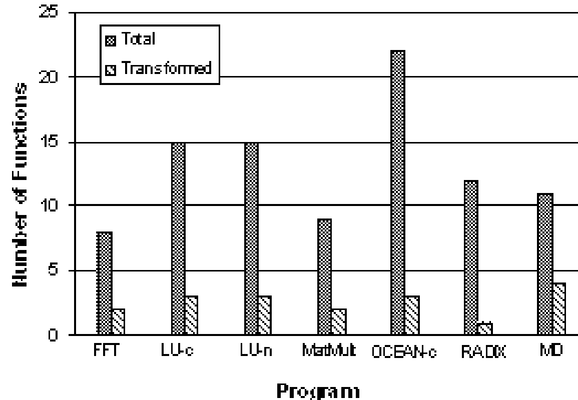

Fig. 6. Functions transformed by Preprocessor

extremely slow for other operations because searching for a particular node in a randomly generated graph is time-consuming. Also, SNOW traces all pointers and slows down much for pointer-intensive programs. *MigThread* virtually only cares about results ("ignore process") and therefore less dependent on type of programs. We summarize the complexity of these three systems, and list the results in Table 1 and 2. The N_{var}, N_{ptr} and N_{mem} represent numbers of variables, pointers and dynamically allocated memory blocks. From these, we can see that *MigThread* is very efficient.

The migration platform is a software Distributed Shared Memory (DSM) System [5] over SMPs (SUN UltraEnterprise 3000s) connected by fast Ethernet. Each SMP contains four 330Mhz UltraSparc processors. The parallelized programs are running on two SMP machines with one thread on each. The communication layer is UDP/IP. Since the inserted primitives do not cause any noticeable slowdown when no migration happens, we only focus on the migration cost and compare it with pure execution time on two SMP nodes. We use several applications from the SPLASH-2 application suite, matrix multiplication, and Molecular Dynamics (MD) simulation to evaluate the thread migration cost.

MigThread's preprocessor scans and transforms C programs automatically. The function call graph eliminates unnecessary functions so that preprocessor only transforms a fraction of functions to reduce the compile-time cost. Only about 10-20% functions require to be transformed (see Fig. 6). This one-time transform procedure takes about 1-8 seconds for our benchmark programs.

The runtime overheads are shown in Table 3. As mentioned before, no noticeable overhead is seen when no migration happens. For most applications, the thread states range from 100 to 184 bytes, and their migration time is around 2.4 ms. Even though the thread state of OCEAN-c is increased to 432 bytes, its migration time does not change. Only thread states of RADIX and MD are big enough to make difference. Since shared data are in DSM's global shared regions which do not need to be migrated with threads, thread state sizes are invariant

Table 3. Migration Overhead in real applications

Program	Input Size	State Size (bytes)	Transform Time (sec)	Execution Time (ms)	Migration Time (ms)	Migr./Exec. Rate (%)
FFT	64 Points	160	5.87	85	2.42	2.85
	1024 Points	160	5.87	112	2.46	2.20
LU-c	16 x 16	184	4.19	77	2.35	3.05
	512 x 512	184	4.19	7,699	2.41	0.03
LU-n	16 x 16	176	4.17	346	2.34	0.68
	128 x 128	176	4.17	596	2.37	0.40
MatMult	16 x 16	100	1.34	371	2.32	0.63
	128 x 128	100	1.34	703	2.47	0.35
OCEAN-c	18 x 18	432	7.98	2,884	2.45	0.08
	258 x 258	432	7.98	14496	2.40	0.02
RADIX	64 keys	32,984	2.86	688	5.12	0.74
	1024 keys	32,984	2.86	694	5.14	0.74
MD	5,286 Atoms	7,040,532	2.45	38,067	83.65	0.22

to problem sizes in Table 3. Compared to programs' execution time, migration cost is so small (mostly less than 1% and at most 3%) for benchmark programs.

The chosen programs are popular, but all array-based. Fortunately, *MigThread* does not slow down particularly for pointer-intensive applications because pointers are not traced all the time. Definitely more memory blocks incur bigger overhead, which is inevitable.

5 Related Work

The major concern in thread migration is that the address space could be totally different on various machines and internal self-referential pointers may no longer be valid. There are three approaches to handle the pointer issue. The first approach is to use language and compiler support to identify and update pointers[4, 8], such as in Emerald[9] and Arachne[8]. But they rely on new languages and compilers. The second approach requires scanning the stacks at runtime to detect and translate the possible pointers dynamically, as in Ariadne[10]. Since some pointers in stack are probably misidentified, the resumed execution can be incorrect. The third approach is popular, such as in Millipede[11] and necessitates the partitioning of address spaces and reservation of unique virtual addresses for the stack of each thread so that the update of internal pointers becomes unnecessary. This faces severe scalability and portability problem [11, 4].

Application-level implementation achieves the heterogeneity feature. The Tui system [2] is a heterogeneous process migration package which modifies a compiler (ACK) to provide runtime information via debugging code and relies on Unix ptrace to obtain the state of processes. SNOW [7] is another heterogeneous process migration scheme which only work on "migration-safe" programs.

Its memory representation model implies a pointer-sensitive design which slows down the migration dramatically. The Porch system [6] reports pointers individually to create state as in SNOW. This might cause flexibility and efficiency problems for complex applications. The thread migration approach in [3] is similar to *MigThread* but has limitations. *MigThread* can handle pointer arithmetic, memory management, and "migration-unsafe" features efficiently.

6 Conclusion and Future Work

MigThread is shown to be generic in its scope. It handles four major "migration-unsafe" features in C/C++. Under *MigThread*, more programs become migratable and programmers do not need to worry if they are coding in "migration-safe" style. Thread state is constructed efficiently at runtime. More adaptation points can be inserted into programs to improve sensitivity of dynamic environment without sacrificing performance. As an application-level approach, *MigThread* places no restriction on thread types and operating systems. Experiments on real applications indicate that the overhead of *MigThread* is minimal.

We are currently porting *MigThread* to multiple platforms to exploit its heterogeneity potential. More work is being conducted on transferring process state and communication state for a complete thread migration package.

References

[1] Milojicic, D., Douglis, F., Paindaveine, Y., Wheeler, R. and Zhou, S.: Process Migration, ACM Computing Surveys (2000)

[2] Smith, P. and Hutchinson, N.: Heterogeneous process migration: the TUI system, Tech rep 96-04, University of British Columbia (1996)

[3] Jiang H. and Chaudhary, V.: Compile/Run-time Support for Thread Migration, Proc. of 16th Int. Parallel and Distributed Processing Symposium (2002)

[4] Thitikamol, K. and Keleher, P.: Thread Migration and Communication Minimization in DSM Systems, Proc. of the IEEE (1999)

[5] Roy, S. and Chaudhary, V.: Design Issues for a High-Performance Distributed Shared Memory on Symmetrical Multiprocessor Clusters, Cluster Computing: The Journal of Networks, Software Tools and Applications No. 2 (1999)

[6] Strumpen, V.: Compiler Technology for Portable Checkpoints, submitted for publication (http://theory.lcs. mit.edu/ strumpen/porch.ps.gz) (1998)

[7] Chanchio, K. and Sun, X.H.: Data Collection and Restoration for Heterogeneous Process Migration, Proc. of Int. Conf. on Distributed Computing Systems (2001)

[8] Dimitrov, B. and Rego, V.: Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms, IEEE Transactions on Parallel and Distributed Systems 9(5), 1998.

[9] Jul, E., Levy, H., Hutchinson, N. and Blad, A.: Fine-Grained Mobility in the Emerald System, ACM Transactions on Computer Systems, Vol. 6, No. 1, 1998.

[10] Mascarenhas, E., and Rego, V: Ariadne: Architecture of a Portable Threads system supporting Mobile Processes, CSD-TR 95-017, Purdue University (1995)

[11] Itzkovitz, A., Schuster, A., and Wolfovich, L.: Thread Migration and its Applications in Distributed Shared Memory Systems, Journal of Systems and Software, vol. 42, no. 1, 1998.