

# FACILITATING AGENT NAVIGATION USING DSM – HIGH LEVEL DESIGNS

Lei Pan, Ming Kin Lai, Javid J. Huseynov, Lubomir F. Bic, and Michael B. Dillencourt

School of Information & Computer Science

University of California, Irvine

Irvine, CA 92697-3425, USA

{*pan,mingl,javid,bic,dillenco*}@ics.uci.edu

**ABSTRACT:** By combining research results from different communities, better systems can be built. This is exemplified in our designs of two mobile agent systems on top of distributed shared memory. The benefits of our new designs include improved efficiency, ease of construction, and ease of use.

## I. INTRODUCTION

Message Passing (MP) and Distributed Shared Memory (DSM) [1], [2], [3], [4], [5] are the two most common approaches to programming on distributed memory systems. MP is scalable but difficult to use, and DSM is easy to use but unscalable. The reason behind the efficiency of MP is that it encourages the programmers to place a sub-computation on the node where the largest amount of data required by the sub-computation resides. This principle is sometimes referred to as “owner-computes” [6], but because this term has been widely used to mean slightly different things [7], we now call it the principle of **pivot-computes**. We define pivot-computes to be “the process/thread on the pivot node which owns large sized data of a sub-computation computes.” *Shared variable programming*, in which “processes/threads communicate and synchronize through shared variables,” is what makes DSM easy to use. A mobile agent approach suggested in our earlier work [8], [9], [6] combines the advantages of the two existing approaches and avoids the disadvantages.

Our mobile agent approach is facilitated by strong mobility [10], [11], [12], [13] provided by mobile agent systems. Today, most mobile agent systems are built on top of MP; that is, the daemons that provide mobile agent environments are implemented using MP (e.g., using sockets). In this paper, we present high level designs of mobile agent systems built on DSMs. Our designs have multiple benefits from utilizing research results in both DSM and mobile agent communities. In particular, the use of mobile agents exploits data accessing locality; programming on DSM systems is easy; and DSM systems provide excellent mechanisms to reuse communicated data.

We first survey DSM systems in section II, then describe our existing mobile agent system in sub-section III-A. The design of our two new systems are presented in

sub-section III-B and sub-section III-C. Related work and final remarks are provided thereafter.

## II. DISTRIBUTED SHARED MEMORY SYSTEMS

DSM systems provide a logically shared memory over physically distributed memory. They combine the programming advantages of shared memory and the cost advantages of distributed memory. This section briefly surveys some DSM systems.

### A. Classifications

DSM systems can be classified in the following dimensions: granularity of sharing, granularity of implementing an application, implementation level of the DSM, and organization of sharing memory address space.

#### A.1 Page-based vs Object-based DSM Systems

Page-based DSM systems use hardware pages as units of sharing among processes. They provide less complicated synchronization model [14]. They also provide transparency; that is, the fact that memory is distributed is hidden from the users. However, page-based DSM systems suffer from false sharing, which occurs when two processes try to access two different data items that are not shared but are physically allocated to a single page. The larger is the page size, the higher is the possibility of false sharing. Munin [15], TreadMarks [16] and Millipede [17] are examples of page-based DSM systems.

In an object-based DSM system, processes or threads on multiple machines share an abstract space filled with shared objects. The location and management of the objects is handled automatically by the runtime system. This model is in contrast to page-based DSM systems, which provide only a raw linear memory of bytes. Object-based design avoids false sharing. Examples of object-based DSM implementations are CRL [18], Adsmith [19], Midway [20], and Emerald [21].

There has been no clear indication that object-based DSM systems are better than page-based DSM systems. There have been studies suggesting that page-based DSM systems outperform object-based DSM systems in certain situations [14].

## A.2 Process-oriented vs Thread-oriented DSM Systems

DSM systems can be process-oriented or thread-oriented [22]. In process-oriented DSM systems, a parallel application is implemented as a set of operating system (OS) processes. Process-oriented DSM systems include Genesis [23] and Stardust [24].

In thread-oriented DSM systems, such as D-CVM [25], a parallel application is implemented as a set of threads, and all threads belong to a single OS process.

Both process- and thread-oriented DSM systems can be implemented at kernel level or user level.

## A.3 Kernel-level vs User-level DSM Systems

Kernel level DSM implementations take advantage of underlying OS and hardware mechanisms. The advantages of kernel-level DSM systems are simple event handling, blocking a thread without blocking an entire process, and robust scheduling. The biggest disadvantage of kernel-level DSM systems is poor portability, since the kernel is always tied to a particular OS and hardware. Examples of kernel-level DSM systems are Gobelins [22] and Genesis [23].

Most of the DSM systems are implemented at a user (software) level. User-level DSM systems provide a shared address space for shared data on top of physically distributed memory using software support [1]. They rely on user-level memory management techniques provided by the OS to detect accesses and updates to shared data at the level of pages/objects. The software DSM system then applies a memory coherence protocol to provide an illusion of shared memory. Some well-known user-level thread-oriented DSM systems are Munin [15], Millipede [26], TreadMarks [16], and DSM-PM2[27].

## A.4 Structure of Shared Address Space

There are variations as to how the address space in a DSM system is structured. A number of DSM systems, such as Amber [28] and Arias [29], do not provide separate address spaces or separate regions in an address space for non-shared data. On the other hand, some DSM systems provide such for non-shared data. Examples are IVY [30], [31], Ra (the kernel of the Clouds distributed OS [32]), Mirage [33] and CMU's Mach server [34], [35].

## B. Memory consistency models for DSM systems

DSM systems replicate data to allow concurrent access. Thus, a memory consistency model (or memory coherence protocol) is needed for shared data. There are a number of memory consistency models; among them, lazy release consistency model and multiple-writer coherence protocols improve the performance of user-level DSM systems.

A release consistency model permits a processor to delay making its changes to shared data visible to other processors until special acquire or release synchronization accesses occur. The propagation of the modifications can thus

be postponed until the next synchronization operation takes effect [36]. In the lazy release consistency model [37], the propagation of modifications are sent to other nodes only upon demand, or acquire; therefore, there is no network traffic generated until acquire access. Consequently, lazy release model has better performance compared to other memory consistency models for DSM systems. Lazy release consistency is implemented in TreadMarks.

Multiple-writer consistency models [15] address the issue of false sharing described above. They allow various processors to simultaneously modify their local copies of a shared page and merge the copies upon next synchronization. DSM software then makes a copy of the page (twin) and allows processors to send encoded modification records (diffs) to a twin, which can be merged by a receiving processor [36]. Lazy release consistency allows diff creation to be postponed until the acquire access, decreasing the number of created diffs and improving performance. Multiple-writer protocols were first implemented in Munin [15].

## C. Process/Thread migration

Although there are differences between a process and a thread as described above, we will refer to both processes and threads as threads in our discussion from now on for simplicity.

On multiprocessor systems, spreading execution of threads over several processors can exploit parallelism and thus achieve improved performance. However, two issues arise in such systems: load imbalance and remote data access.

During program execution, there may be a dense group of threads residing on a single processor while only a few threads exist on other processors. A balanced distribution of threads across processors improves system performance and provides an opportunity to achieve a better overall throughput.

Threads may access remote data and thus require unavoidable inter-processor communication. If the cost (in terms of the number of accesses and the amount of data transferred) of inter-processor data access is high, then relocating an accessing thread to the remote node can reduce inter-processor communication traffic.

Combining the advantages of load balancing and local data access, effective migration ensures better performance. Millipede, D-CVM, DSM-PM2, and TreadMarks support thread migration at the user level, Genesis supports process migration at kernel level. Gobelins provides thread migration on a Linux cluster, where threads are implemented as processes.

## III. DESIGNING MOBILE AGENT SYSTEMS

Our designs are inspired by the following observations. First, computation mobility in the form of mobile agents [38], [39], [40], or thread migration [41], is essential to

good performance as it exploits data locality. Second, shared variable programming is at a higher level and easier to use than message passing. Third, DSM systems as being used today suffer from false sharing (in page-based DSM systems), and memory coherence protocols are not being used effectively, and hence DSM systems are not scalable [42]. Fourth, a mobile agent system such as MESSENGERS suffers from a so-called “fuzzy locus” problem when it is implemented based on message passing. The objectives of our designs are more efficient mobile agent systems through better use of a DSM system, and easier constructions of mobile agent systems using a DSM system.

The navigation of a mobile agent is **explicit** if explicit navigational statements are needed in the code. MESSENGERS [43], [44], [45], [46] is such a system: it provides explicit navigation using the *hop()* statement. Mobile agents that support explicit navigation are said to possess **compile-time strong mobility**, for which the exact places in the code where navigation can happen is predefined before compile time. Agent navigation can also be done in a more transparent way in which a programmer does not directly control it. Rather, the underlying system triggers agent navigation using some type of protocol. The protocol used to drive mobile agents follows the principle of pivot-computes, and its objective is to exploit data accessing locality as much as possible. This kind of navigation is called **implicit**, and mobile agents that provide implicit navigation possess **run-time strong mobility**.

In this section, a description of our existing mobile agent system is followed by the design of two new DSM-based systems.

#### A. MESSENGERS: *explicit navigation using MP*

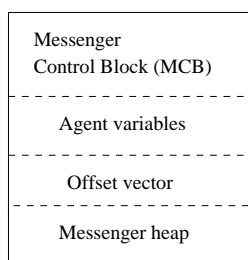


Fig. 1. Messenger structure.

The MESSENGERS system [43], [44], [45], [46], developed in School of Information & Computer Science at University of California, Irvine, is an environment for general purpose distributed computing. In MESSENGERS, applications are developed as collections of self-migrating agents, called Messengers. Like other implementations of mobile agents with *strong mobility* [10], [11], [12], [13], a Messenger can halt its execution, encapsulate the values of its

variables, move to another node, restore the state, and continue executing. In this subsection, we briefly describe the design of MESSENGERS-C [44], [47], the “compiled” version of our MESSENGERS system. This “compiled” version of MESSENGERS is faster than an older “interpreted” version named MESSENGERS-I.

Three levels of networks are used. The lowest level is the physical network (e.g., a LAN or WAN), which constitutes the underlying computational processing elements. Superimposed on the physical layer is the daemon network, where each daemon is a server process that receives, executes, and dispatches Messengers. The logical network is an application-specific computation network created on top of the daemon network. Messengers may be injected (by the program *inject* which is part of the MESSENGERS system, or by another Messenger) into any of the daemon nodes and they may start creating new logical nodes and links on the current or any other daemons. Based on application need, multiple logical nodes can be created on one physical node.

A MESSENGERS daemon executes compiled Messengers. A daemon and all Messengers running on it share one process, and the Messengers are linked to the process dynamically. Messengers are allowed to call C functions, grouped in a user library which is also dynamically linked to the process. There are four tasks for a daemon. First, to accept UNIX signals to inject Messengers. These signals are sent by the program *inject* or another Messenger. Second, to respond to requests, such as “search node” and “complete link,” from other Messengers. Third, to add incoming Messengers to a ready list. And fourth, to execute Messengers. In addition, a daemon also provides a function, the calling of which would result in the autonomous caller Messenger being sent to a destination daemon. Socket-level message passing is used by a daemon or a Messenger to communicate with remote daemons.

The structure of a Messenger is depicted in Fig. 1. A Messenger consists of a small Messenger control block (or MCB, storing such data as “pointer to next function,” library name, Messenger size), agent variables, a vector of offsets used to access memory in a Messenger heap used for dynamic arrays, and the heap itself [47].

There are two types of variables in MESSENGERS: agent variables and node variables. An agent variable is private to a particular Messenger and travels with that Messenger as it migrates through the logical network. A node variable is stationary, and is accessible by all Messengers currently at the logical node to which the variable belongs. Hence agent variables can be used to carry data between nodes, while node variables can be used for inter-agent communication.

A Messenger’s programmer tells it to migrate using the navigational statement *hop()*. A destination node’s logical address or a logical link between the source and the destination nodes can be used as the argument for the *hop()* statement. When a Messenger hops, it takes the data in its

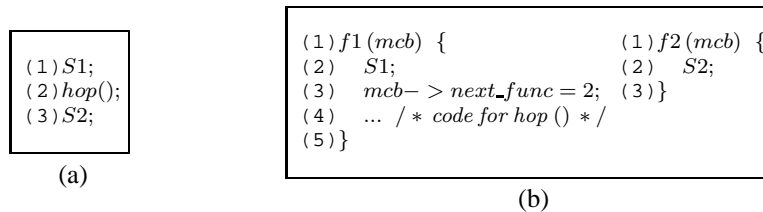


Fig. 2. Transformation of straight-line code. (a) MESSENGERS code. (b) Restructured functions.

agent variables with it to wherever it migrates.

A Messenger can spawn another Messenger using the statement *inject()*. Synchronization among Messengers uses “events,” and the statements *signalEvent()* and *waitEvent()*. Since no remote data accessing is allowed, the events are local and so is synchronization. A Messenger’s execution is not preempted between any two navigational statements. A Messenger must explicitly relinquish control to other Messengers using statements such as *hop()*. We call this feature “non-preemptive scheduling.”

The MESSENGERS compiler lies at the heart of the system. MESSENGERS code, with strong mobility, is translated into stationary code communicating using message passing with sockets. This suggests that our mobile agent approach to distributed programming is at a higher level than message passing. The translated stationary code (in C) is then further compiled into machine native code for execution. Strong mobility in MESSENGERS means that computation migrates in the network in the form of a program counter. The actual mechanism for handling program counters can be seen from a simple example [44]. The basic idea is to break a MESSENGERS program into smaller functions (in C) at navigational or other context-switching statements, and use the “pointer to next function” as our artificial program counter. Fig. 2(a) shows MESSENGERS code of two statements *S1* and *S2* separated by a context-switching statement *hop()*. This MESSENGERS code is compiled by the MESSENGERS compiler into two functions, each takes a pointer to a Messenger control block (*mcb*) as its argument, shown in Fig. 2(b); *f1()* uses a “pointer to next function” to point to function *f2()* which is executed after migration. Line (4) in Fig. 2(b) represents code that calls the daemon function mentioned earlier, and by calling this function the Messenger autonomously sends itself to a destination daemon.

One subtle but important feature of the MESSENGERS system is that it allows code to either be loaded from a shared disk or, in a non-shared file system, be sent across the network at most once, irrespective of how many times the locus of computation moves across the network [48]. This is crucial to performance.

The overhead of mobile agent navigation in MESSENGERS is relatively small due to the following reasons. First, since a Messenger is compiled into smaller functions at

navigational statements, it is not necessary to save the function stack but only the “pointer to next function.” Second, the “extra” information associated with a Messenger’s status (i.e., MCB) is small. Third, no marshalling and unmarshalling of the agent variables is needed. Fourth, a Messenger runs in the same memory space of a daemon, which is why adding or removing it from the ready list takes only a few instructions to update some pointers in the list, instead of doing memory copying. The navigation of Messengers is almost as fast as sending TCP messages using a C program.

### B. Explicit navigation using DSM

In this section, we describe a high level design of an explicit-navigation mobile agent system based on a DSM.

In our earlier work [49], [50], we proposed to have shared variable programming beyond shared memory using mobile agents. *Shared variable programming* is a model of computing in which inter-thread communication and synchronization are managed through the use of variables that multiple threads share. It is easier than MP because a programmer can read from and write to remote memory with simple assignment statements. Shared variable programming is possible beyond a single-address space because the strong mobility of mobile agents enables them to migrate to remote data in order to share it. This action is dual to the data “pulling” in classical DSM systems, and both data pulling and agent migration make variable sharing possible. The actual mechanism we provide for shared variable programming in non-shared memory environment is *distributed shared variable*, or *DSV* [50], plus mobile agent migration. We assume that as the size of a problem gets larger and larger, and hence distributed computing is needed, one or more data structures need to be distributed. Examples of such data structures are “large” matrices, “large” linked lists, or “large” trees. The word “large” here implies that these data structures are too big to fit in any single memory and therefore will need to be distributed using some transformations.

If a transformation is such that (1) any item in a data structure is accessed using the same scheme (e.g., same index or same pointer arithmetic) before and after the transformation; (2) any thread can access any item in the transformed data structure directly without any help from other application-level thread, we define this transformation to

preserve **data structure integrity**. Data structure integrity represents the notion that as the problem size scales up, large data structures are each still logically an integral one even if physically they are distributed, and they are being accessed in the same way as in the original sequential program developed in a single-address space. Data structure integrity should be preserved for both indexed and pointer-based data structures.

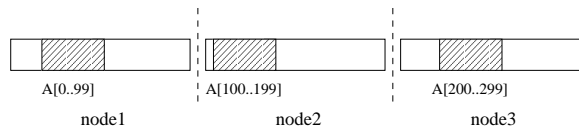


Fig. 3. Distributed shared variable.

The use of DSV helps to preserve data structure integrity. Fig. 3 shows an example of DSV. Each array on a node is a MESSENGERS node variable, and the three of them together make a DSV  $A[0..299]$  that represents a large array being distributed over three nodes. The single address of this DSV is provided through pointer shift (e.g., on node2  $A = A - 100$  and on node3  $A = A - 200$ , after memory allocation) [49], or in a more sophisticated situation, through a “global-to-local index map” [9]. A qualifier similar to *shared* in UPC [51], [52] can be provided as a user interface for the programmers to specify data distribution pattern. Transforming the large array  $A[.]$  into the DSV in this example preserves data structure integrity because the array indexing scheme is the same as if the array  $A[.]$  is small and not distributed, and a mobile agent can access any array entry autonomously by hopping to the node hosting the data. The migration of mobile agents serves as a bridge between distributed memories. It is obvious that DSM systems preserve data structure integrity. They do so by mapping local memories into a single virtual address space, and provide data pulling for the nodes that conduct remote data access through the virtual space. Our DSV with mobile agents turns DSM on its head. Instead of mapping memory to threads, we migrate threads to data. MP does not preserve data structure integrity because although it is not hard to provide a single index scheme for a physically distributed but logically single data structure (e.g., one could use the same scheme shown in Fig. 3 for arrays in MP programs), an MP process would need the help from another application-level process to access remote data. With MP system that provides one-sided communication (e.g., MPI-2 [53]), this help may be delayed but is still needed, otherwise the principle of pivot-computes will be violated sooner or later.

Table I gives a taxonomy of variables in MESSENGERS. In addition to the primitive node and agent variables, shared variable programming in distributed memory requires the use of DSV. Since DSVs are constructed using node vari-

ables that are local to nodes, in order to access an item in a DSV, a mobile agent is required to migrate to the node variable containing the item. In other words, no remote data accessing is provided. Note that in Table I, a distributed and publicly shared DSV is not globally accessible. On the other hand, a non-shared privately owned agent variable is globally accessible by its owner agent. This is in contrast to the classical use of DSM in which all data put on DSM is both shared and globally accessible.

TABLE I  
A TAXONOMY OF VARIABLES

	non-distributed	distributed
non-shared	(none)	agent variable
shared	node variable	DSV

DSV with mobile agents exploits data locality. To explain how this is done, we introduce the concept of **distributed code building block**, or DBLOCK. A DBLOCK is defined to be any fragment of code (e.g., a loop, an if-then-else statement, or a sequence of straight-line code) that accesses data that has already been allocated to two or more distinct machines. A DBLOCK cannot execute exclusively on a single machine. The essence of distributed programming is nothing else but resolving DBLOCKS. In DBLOCK resolution, an efficient way is to find the pivot of the DBLOCK (i.e., the node that owns the largest sized data used by the DBLOCK), move the other distributed data involved to the pivot, and compute on the pivot node. Note that the pivot can change during the execution of a DBLOCK. One example of this is a loop that sequentially browses through the distributed 1-D array  $A[.]$  shown in Fig. 3. The computation result may need to be assigned back to a remote node. This DBLOCK resolution follows the principle of pivot-computes. With MP, a DBLOCK is broken down into smaller code blocks assigned to different nodes communicating with each other using messages. These smaller code blocks are no longer DBLOCKS because they compute using only local data and locally buffered messages. The problem with the MP way of DBLOCK resolution is code restructuring. With DSM, a whole program is treated as one DBLOCK, and the way to resolve it is to provide transparent remote data accessing. This approach causes more data to be moved because computations of the DBLOCKS may not happen on their pivot nodes. With DSV and mobile agents, we resolve a DBLOCK by putting its large data in a DSV and its small data in agent variables. A mobile agent carrying small data migrates to the pivot node and computes there. In this way, the principle of pivot-computes is followed, and the code structure is preserved.

The fact that we proposed shared variable programming beyond shared memory does not mean that DSM systems cannot be used to help us. In fact, DSM systems can be

used to serve as communication media for the globally accessible data, or in other words, the agent variables. This is why we are designing our mobile agent systems on top of DSM systems. Fig. 4 depicts the functional components of the system and their dependencies. A DSM is, as usual, built on top of a physical network of machines. Our mobile agent system, with a compiler and a daemon system as its major components, is built on the DSM. And the agent programmers program explicit agent navigation using our system. The architecture of a DSM-based explicit-navigation mobile agent system is sketched in Fig. 5.

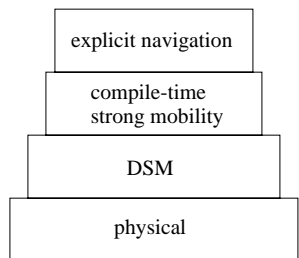


Fig. 4. Functional dependencies of an explicit-navigation mobile agent system.

Conventional wisdom puts the large distributed data in the “shared” memory space in DSM, and this data is at the same time globally accessible. This causes several problems. First, using DSM in this fashion causes inadvertent movement of large data by a programmer, resulting in violation of the principle of pivot-computes. This is the major reason why classical DSM systems are not scalable. One way to correct this, is to provide the programmers with data distribution and thread migration, so that they can control the programs more consciously to follow the principle of pivot-computes. This is basically provided by our design in this subsection. Another possibility is a more automatic solution, which will be the topic of the next subsection. Second, because of the larger-than-necessary data movement, data replication is used in order to reuse the data moved and amortize the cost of communication over multiple repeated uses. But replication leads to the problem of coherence, and memory coherence protocols are then needed in order to guarantee correctness. Now the new problem is that the traffic caused by consistency requirements is by itself a big overhead in some situations. And the protocols are complicated, and they do not always work for data accessing patterns that are dynamic. Third, false sharing can happen because it is impossible to prevent two threads from accessing two separate data items that happen to reside on the same memory page.

In contrast to the classical use of DSM systems, our design puts the large data that we do not want to see moved into the “local” memory space of the computing nodes. The node variables holding the data form one DSV. The array

and the linked list in Fig. 5 are two examples. Here we assume that we are building our mobile agent system based on a DSM system (such as IVY) that provides separate memory space on each node for non-shared data. It is important to realize that “shared” or public data does not have to be “globally accessible.” “Shared” means multiple threads have the right to access the data, while “globally accessible” says the data is accessible from anywhere. These are two different properties. The classical use of DSM provides both properties at the same time to the distributed data, which opens door to false sharing. Our design of DSV makes it false-sharing free, simply because all data in a DSV is not moved at all. Data moving is a result of global accessibility.

On the other hand, “non-shared” or privately owned data may need to be made “globally accessible.” This is true for agent variables. Agent variables are put on DSM so that they can be available when their owner agents executing DBLOCKS compute on the corresponding pivots.  $x$ ,  $y$ , and  $z$  in Fig. 5 are examples of such agent variables. There are several advantages of using DSM rather than message passing to handle mobile agent migration. First, it prevents false sharing, because agent variables are not shared among different agents, and we can easily make different groups of agent variables belonging to different agents not to share same memory pages by starting a different group from a new page. In this way, we will at most waste a page for each mobile agent. Second, agent migration is generally faster using DSM. An agent can start executing after migrating to a new node as soon as its MCB is shipped via the DSM. This is not possible if we treat the entire Messenger as a message like in MESSENGERS-C. An auxiliary thread can be started to read through the agent’s agent variables causing the DSM to send these variables to the local node. We can utilize the CPU better in this way if the agent is computing heavily using mostly local data stored in DSVs, because the CPU is switching between computation and communication threads without idling. Of course, if the agent hits a read/write miss of some agent variable, it’ll have to wait until the auxiliary thread brings it in. But no expensive context switching is done in this case. Using DSM to pull agent variables on demand saves communication cost in the situation when only part of the agent variables are accessed during a visit. This is very hard to do in MP based MESSENGERS-C. A Messenger, upon finishing its computation, can either stop the auxiliary thread, or hop away leaving the auxiliary thread running to pull the unused agent variables for its next visit. Third, DSM memory consistency protocols can help to reuse the agent variables when the owner agent visits the same node multiple times, because coherence traffic only contains diffs but not the whole agent variable set. Fourth, using DSM can dramatically simplify daemon programming because now we are using shared variable programming rather than message passing (using sockets) as in MESSENGERS-C. Basically,

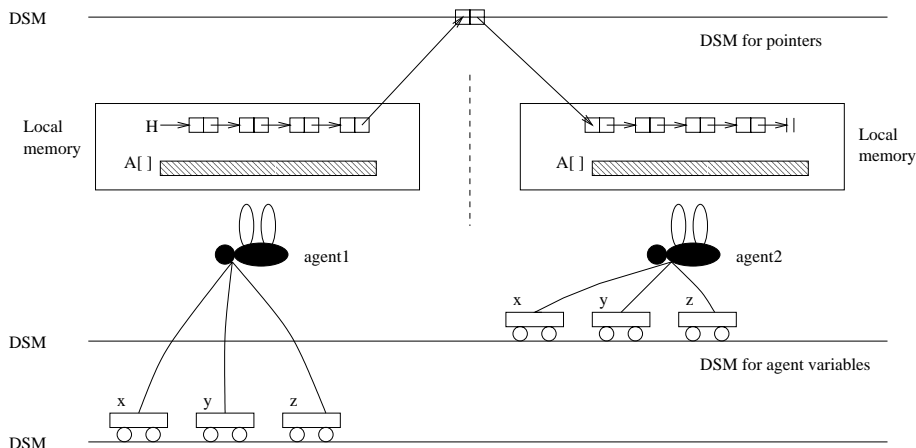


Fig. 5. DSM-based mobile agent system that supports explicit navigation.

a MESSENGERS daemon sets up the connection to other daemons and provides functions to send the Messengers to other daemons. These communication and synchronization can be all done through shared variables put on DSM.

One may question why we call an agent variable “distributed” in Table I. In MESSENGERS-C, an agent variable cannot exist in multiple locations at the same time. So it is “distributed” only in the sense that it resides in distributed locations at different time. In our new design, if a page-based DSM system is used, an agent variable (e.g., an array) can really be spatially distributed at a certain time because the DSM system may only migrate the pages containing a portion of the agent variable (e.g., the first half of the array) that is accessed by the agent after it migrates to a new node. In this situation, the array is indeed distributed on two different nodes.

We should point out that applications with all agents smaller than a page are not good fits for our new design because a page is the basic unit for communication. Possible solutions are vectorizing communication to increase the size of Messengers, or adjusting page size in the underlying network. Fortunately, many numerical problems require putting in agent variables vectors or sub-matrices (in MBs) that are much larger than a page (typically in KBs, but adjustable).

In distributed pointer-based data structures, local pointers are most efficient in accessing local data elements. But occasionally when the machine boundary is hit, we need a “global pointer” to help us jump across. DSM can help us bridge the gap, as depicted in Fig. 5, simply because variables put on DSM can be seen from both nodes.

### C. Implicit navigation using DSM

The use of the explicit-navigation mobile agents requires a programmer to manually drive the agents to follow the principle of pivot-computes for the DBLOCKS. Because of

the use of DSVs which do not provide remote data accessing, in resolving a DBLOCK, a programmer has to be exactly right in driving the locus of computation to the right nodes, or data accessing is going to fail and so is the program. If we put all the data back to the DSM without using DSVs, it relieves the programmer of the burden but ignores the principle. An interesting question is whether we can have a run-time system that automatically drives the Messengers to follow the principle of pivot-computes.

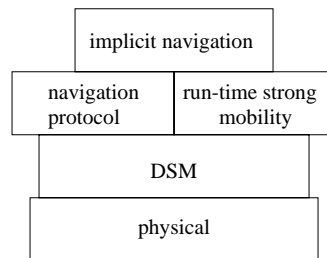


Fig. 6. Functional dependencies of an implicit-navigation mobile agent system.

Fig. 6 depicts a functional design of an implicit-navigation agent system. An important building block is a navigation protocol, which decides when and to which destination a Messenger migrates. Such protocols exist in earlier work [41], and they follow the principle of owner-computes, which for some applications is equivalent to pivot-computes but for others is not. A protocol that follows pivot-computes can also be developed; the basic idea can be as simple as comparing local versus remote page accesses. We will not discuss the protocols in detail here, but once we build our system, we will test out both types of protocols.

The next important building block is a mechanism to support run-time strong mobility. That is, a Messenger should



Fig. 7. Mechanism for run-time migration. (a) MESSENGERS code. (b) Translated functions.

be able to migrate at run-time anywhere in the program without any user hint. In a homogeneous environment, this can be done by moving a function’s activation record [41]. Our design uses the activation record and our “next function pointer” in a hierarchical way. An example of our run-time migration mechanism is depicted in Fig. 7. A MESSENGERS source code is sub-divided into blocks at dashed lines in Fig. 7(a). The decision of where to put those dashed lines can be made by a programmer (who knows where the next pivot node is) by inserting explicit *hop()* statements, or by the MESSENGERS compiler using natural language blocks (e.g., a loop). This source code is compiled by the MESSENGERS compiler into smaller functions each holding a *next\_func* pointer. When the underlying protocol decides to migrate inside the *for* loop of *f2()* to a new pivot node, the activation record of the thread and the *next\_func* pointer of *f2()*, which has a value of 3, are moved to the destination as a compound program counter.

There are at least two benefits from this system. First, this allows a better decision combining both the programmer’s knowledge and the run-time system’s capability. For example, a human knows better about the next pivot node of a coarse-level task, while the run-time system is good at determining dynamic situations. Second, the run-time strong mobility based on a DSM can be used to resolve the problem of the so-called “fuzzy locus” existing in explicit-navigation systems such as MESSENGERS-C. The problem of fuzzy locus happens when the locus of computation moves across node boundaries. Within one or a couple of statements, the program accesses data on both sides of the boundary. A simple example of fuzzy locus is shown in Fig. 8. This code is a loop over a 1-D array  $A[\cdot]$  depicted in Fig. 3. Without remote data accessing, a MESSENGERS programmer needs to handle the situation when  $A[i]$  and  $A[i - 1]$  are on neighboring nodes by breaking line (2) and inserting an explicit *hop()* statement. This may result in inefficient code because for most of the loop  $A[i]$  and  $A[i - 1]$  reside on the same node, and may be too tedious to do if the statement is more complicated and more *hop()* statements are required. The problem of fuzzy locus can happen with

```

(1) for i = 2 : n
(2)   A[i] = A[i - 1] + 1
(3) end

```

Fig. 8. Pseudocode that can cause the phenomenon of fuzzy locus.

pointer-based data structures for similar reason. With implicit navigation, no manual code change is needed, and the program runs correctly on either side of the boundary. The efficiency of the program is the job of the navigation protocol. When there is no obvious pivot node (i.e., data accessed on either side is about the same size), the protocol may want to avoid aggressive migration. And as an obvious pivot emerges, the protocol would need to detect and react to it quickly.

We should point out that in this design, all data will be put on DSM, and is both shared and globally accessible by all Messengers, and hence false sharing is still a possibility.

#### IV. RELATED WORK

Our design focuses on how to use a DSM system to implement a mobile agent system to explore data locality. In this section we describe how our work relates to some other work in the DSM area.

There is a small group of DSM systems, such as Cohesion [54], Millipede[55], [26], MigThread-outfitted Strings [56], and JIAJIA [57], and DSM-included distributed operating systems, like Gobelins, that use process/thread migration mainly to improve load balancing. There also exist DSM systems, e.g. D-CVM, and those, e.g. Prelude [58] and MCRL [59], that make use of thread migration and computation migration respectively to exploit data locality.

Prelude was one of the pioneers in using computation migration, which can be viewed as a generalization of thread migration, on a DSM system, to exploit data locality. Computation migration is the partial migration of active threads. While Prelude used a compiler to implement computation migration, the Olden system [60] implemented it using a run time system. MCRL is a multithreaded DSM system

and Hsieh [41] developed it to implement dynamic computation migration, which is a combination of data migration and computation migration. The decision to migrate computation or data is made at run time. The Prelude and MCRL projects were mainly focused on data locality rather than load balancing. The former relies on a compiler but the latter provides only an interface for a compiler-based implementation but not a compiler itself. Hsieh pointed out that data migration with replication, employed by most DSM systems, hurts the performance of write operations involving data shared by different processes, as this data must be kept coherent. He observed that computation migration reduces the demand for network bandwidth and eliminates coherence traffic. Hsieh implemented computation migration by migrating single activation records of a thread, though he argued that it should be possible to migrate partial or multiple activation records of a thread. The Prelude, Olden and MCRL projects use program annotations, such as a “move” statement, to indicate (potential) points of computation migration. Hsieh proposed two heuristics, STATIC and REPEAT, to guide the choice between computation migration and data migration: STATIC always migrates computation for non-local writes, and migrates data for non-local reads; REPEAT always migrates computation for non-local writes, and dynamically chooses between computation migration and data migration for non-local reads: specifically, data migration is chosen when two repeated reads for the same data occurs; otherwise, computation migration is used.

In summary, among the vast number of DSM systems, there are only a handful that make use of thread migration, and only a couple such as MCRL that employ thread migration to exploit data locality. Our investigation aims to encourage further research in this promising area.

#### V. FINAL REMARKS

We have described the design of two mobile agent systems based on the use of DSM systems. In the discussion we implicitly assumed in several places that we are using a page-based DSM.

In the explicit-navigation system, two performance benefits can be obtained: (1) avoiding inadvertent movement of large data in conventional use of DSM; (2) improving reuse of communicated data using data replication and memory coherence protocols. The first benefit is a result of following the principle of pivot-computes, and it can be obtained in handcrafted message passing programs as well. However a message passing system such as MPI does not support automatic message reuse, i.e., the second benefit. The fact that MP programs generally have better scalability than classical DSM programs suggests that the first benefit is magnitudes more important than the second. It seems that the classical use of DSM focuses more on maximizing parallelism rather than exploiting data locality which makes them lose the first benefit and hence lose the “battle” with

the MP approach. In our design, we enjoy both benefits, the second of which, even though smaller compared to the first one, is the one that MP systems do not provide.

Our design of implicit-navigation system aims at providing automatic or semi-automatic distributed computing. Although it is generally true that programs manually coded with application knowledge taken into consideration perform better, automatic systems are easier to use and hence have their own places in the community. It would be very interesting to build both and compare the usefulness using a suite of applications. This will be our future work.

More than a decade of efforts have been put into the research of DSM systems, and the output is not ideal in the sense that DSM systems are still not as scalable and efficient as MP ones [42]. But the building blocks of DSM systems, especially the memory coherence protocols, represent the fruits of excellent research and deserve to be built on. The problem now is how to use and reuse them properly, and this paper is part of our attempt.

#### ACKNOWLEDGEMENTS

The authors wish to thank Koji Noguchi for helpful discussions about the MESSENGERS system.

#### REFERENCES

- [1] B. Nitzberg and V. Lo, “Distributed shared memory: A survey of issues and algorithm,” *IEEE Computer*, vol. 24, no. 8, pp. 52–60, Aug. 1991.
- [2] S. Venkatesh and M. Kumar, “A survey of distributed shared memory systems,” *Computer Science & Informatics*, vol. 24, no. 3, pp. 1–14, 1994.
- [3] M. T. J. Protić and V. Milutinović, “A survey of distributed shared memory systems,” in *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, T. N. Mudge and B. D. Shriver, Eds. Los Alamitos, Calif.: IEEE Computer Society Press, Jan. 1995, pp. 74–84.
- [4] W. Shi, W. Hu, and Z. Tang, “Shared virtual memory: A survey,” Center of High Performance Computing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, Tech. Rep. 980005, Apr. 1998.
- [5] A. Judge, P. A. Nixon, V. J. Cahill, B. Tangney, and S. Weber, “Overview of distributed shared memory,” Department of Computer Science, Trinity College, Dublin, Ireland, Tech. Rep. TCD-CS-1998-24, Oct. 1998.
- [6] L. Pan, L. F. Bic, M. B. Dillencourt, and M. K. Lai, “Mobile agents – the right vehicle for distributed sequential computing,” in *Proceedings, 9th International Conference on High Performance Computing - HiPC 2002*, ser. Lecture Notes in Computer Science, S. Sahni, V. K. Prasanna, and U. Shukla, Eds., vol. 2552. Berlin, Germany: Springer-Verlag, Dec. 2002, pp. 575–584.
- [7] A. Rogers and K. Pingali, “Compiling for distributed memory architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 3, pp. 281–298, Mar. 1994.
- [8] L. Pan, L. F. Bic, and M. B. Dillencourt, “Distributed sequential computing using mobile code: Moving computation to data,” in *Proceedings of the 2001 International Conference on Parallel Processing (ICPP 2001)*, L. M. Ni and M. Valero, Eds. Los Alamitos, Calif.: IEEE Computer Society, Sept. 2001, pp. 77–84.
- [9] L. Pan, L. F. Bic, M. B. Dillencourt, J. J. Huseynov, and M. K. Lai, “Distributed parallel computing using navigational programming: Orchestrating computations around data,” in *Proceedings, 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*. Calgary, AB, Canada: ACTA Press, Nov. 2002, pp. 458–463.

- [10] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, May 1998.
- [11] D. Milojevic, "Trend wars: Mobile agent applications," *IEEE Concurrency*, vol. 7(3), pp. 80–90, July–Sept. 1999.
- [12] L. Bettini and R. De Nicola, "Translating strong mobility into weak mobility," in *Proceedings, 5th International Conference on Mobile Agents, MA 2001*, ser. Lecture Notes in Computer Science, G. P. Picco, Ed., vol. 2240. Berlin, Germany: Springer-Verlag, Dec. 2001, pp. 182–197.
- [13] D. Kotz, R. Gray, and D. Rus, "Future directions for mobile agent research," *IEEE Distributed Systems Online*, vol. 3, no. 8, 2002. [Online]. Available: <http://dsonline.computer.org/0208/f/kot/print.htm>
- [14] B. Buck and P. Keleher, "Locality and performance of page- and object-based DSMs," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Computing*. Los Alamitos, Calif.: IEEE Computer Society, Mar.–Apr. 1998, pp. 687–693.
- [15] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. New York, N.Y.: ACM Press, Oct. 1991, pp. 152–164.
- [16] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "Tread-Marks: Distributed shared memory on standard workstations and operating systems," in *Proceedings of the Winter 1994 USENIX Conference*. Berkeley, Calif.: USENIX Association, Jan. 1994, pp. 115–131.
- [17] A. Itzkovitz, A. Schuster, and L. Shalev, "Thread migration and its applications in distributed shared memory systems," *Journal of Systems and Software*, vol. 42, no. 1, pp. 71–87, July 1998.
- [18] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach, "CRL: High-performance all-software distributed shared memory," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. New York, N.Y.: ACM Press, 1995, pp. 213–226.
- [19] W.-Y. Liang, C.-T. King, and F. Lai, "Adsmith: An object-based distributed shared memory system for networks of workstations," *IEEE Transactions on Information and Systems*, vol. E80-D, no. 9, pp. 899–908, Sept. 1997.
- [20] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The Midway distributed shared memory system," in *Digest of Papers, COMPCON Spring '93. (Proceedings of the 38th Annual IEEE Computer Society International Computer Conference)*, Feb. 1993, pp. 528–537.
- [21] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109–133, Feb. 1988.
- [22] G. Vallée, C. Morin, J.-Y. Berthou, I. D. Malen, and R. Lottiaux, "Process migration based on Gobelins distributed shared memory," in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid CCGRID 2002*. Alamitos, Calif.: IEEE Computer Society, May 2002, pp. 301–306.
- [23] A. M. Goscinski, M. J. Hobbs, and J. Silcock, "Genesis: The operating system managing parallelism and providing single system image on cluster," School of Computing and Mathematics, Deakin University, Geelong, Victoria, Australia, Tech. Rep. TR C00/03, Feb. 2000.
- [24] G. Cabillic and I. Puaut, "Stardust: An environment for parallel programming on networks of heterogeneous workstations," *Journal of Parallel and Distributed Computing*, vol. 40, no. 1, pp. 65–80, Jan. 1997.
- [25] K. Thitikamol and P. Keleher, "Thread migration and communication minimization in DSM systems," *Proceedings of the IEEE, Special Issue on Distributed Shared Memory Systems*, vol. 87, no. 3, pp. 487–497, Mar. 1999.
- [26] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster, "MILLIPEDE: Easy parallel programming in available distributed environments," *Software – Practice and Experience*, vol. 27, no. 8, pp. 929–965, Aug. 1997.
- [27] G. Antoniu and L. Bouge, "DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols," in *Proceedings, 6th International Workshop of High Level Parallel Programming Models and Supportive Environments (HIPS 2001)*, ser. Lecture Notes in Computer Science, F. Mueller, Ed., vol. 2026. Berlin, Germany: Springer-Verlag, Apr. 2001, pp. 55–70.
- [28] 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 multiprocessors," in *Proceedings of the 12th ACM Symposium on Operating System Principles*. New York, N.Y.: ACM Press, Dec. 1989, pp. 147–158.
- [29] E. Pérez-Cortés, P. Dechamboux, and J. Han, "Support for synchronization and consistency in Arias," in *Proceedings, Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*. Los Alamitos, Calif.: IEEE Computer Society, May 1995, pp. 113–118.
- [30] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," in *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*. New York, N.Y.: ACM Press, Nov. 1986, pp. 229–239.
- [31] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 321–359, Nov. 1989.
- [32] J. M. Bernabeu-Auban, P. W. Hutto, Y. M. Khalidi, M. Ahamad, W. F. Appelbe, P. Dasgupta, and R. J. LeBlanc, "Clouds - a distributed, object-based operating system architecture and kernel implementation," in *Proceedings of the Autumn 1988 EUUG Conference*. Europe UNIX System User Group, Oct. 1988, pp. 25–37.
- [33] B. D. Fleisch and G. J. Popek, "Mirage: A coherent distributed shared memory design," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*. New York, N.Y.: ACM Press, Dec. 1989, pp. 211–223.
- [34] A. Forin, J. Barrera, M. Young, and R. Rashid, "Design, implementation, and performance evaluation of a distributed shared memory server for Mach," School of Computer Science, Carnegie Mellon University, Pittsburg, Pa., Tech. Rep. CMU-CS-88-185, Aug. 1988.
- [35] A. Forin, J. Barrera, and R. Sanzi, "The shared memory server," in *Proceedings of the Winter 1989 USENIX Conference*. Berkeley, Calif.: USENIX Association, Jan.–Feb. 1989, pp. 229–243.
- [36] P. Keleher and C.-W. Tseng, "Improving the compiler/software DSM interface: Preliminary experiences," in *Proceedings of the First SUIF Compiler Workshop*, Jan. 1996. [Online]. Available: <http://www-suif.stanford.edu/suifconf/suifconf1/papers/paper14.ps>
- [37] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*. New York, N.Y.: ACM Press, May 1992, pp. 13–21.
- [38] D. Chess, C. Harrison, and A. Kershenbaum, "Mobile agents: Are they a good idea?" in *Selected Presentations and Invited Papers, Second International Workshop on Mobile Object Systems, MOS '96*, ser. Lecture Notes in Computer Science, J. Vitek and C. Tschudin, Eds., vol. 1222. Berlin, Germany: Springer-Verlag, July 1997, pp. 25–47.
- [39] D. B. Lange and M. Oshima, "Seven good reasons for mobile agents," *Communications of the ACM*, vol. 42, no. 3, pp. 88–89, Mar. 1999.
- [40] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus, "Mobile agents: Motivations and state-of-the-art systems," Dartmouth College, Hanover, N.H., Tech. Rep. TR2000-365, Apr. 2000.
- [41] W. C.-Y. Hsieh, "Dynamic computation migration in distributed shared memory systems," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, Mass., 1995.
- [42] A. S. Tanenbaum and M. Van Steen, *Distributed Systems Principles and Paradigms*. Upper Saddle River, N.J.: Prentice Hall, 2002.
- [43] L. F. Bic, M. Fukuda, and M. B. Dillencourt, "Distributed computing using autonomous objects," *IEEE Computer*, vol. 29, no. 8, pp. 55–61, Aug. 1996.
- [44] C. Wicke, L. F. Bic, M. B. Dillencourt, and M. Fukuda, "Automatic state capture of self-migrating computations in MESSENGERS," in *Proceedings, Second International Conference on Mobile Agents, MA '98*, ser. Lecture Notes in Computer Science, K. Rothermel and F. Hohl, Eds., vol. 1477. Berlin, Germany: Springer-Verlag, Sept. 1998, pp. 68–79.
- [45] M. Fukuda, L. F. Bic, and M. B. Dillencourt, "Messages versus messengers in distributed programming," *Journal of Parallel and Distributed Computing*, vol. 57, pp. 188–211, 1999.
- [46] E. Gendelman, *MESSENGERS User's Manual (version 2.1)*, 2001.
- [47] C. Wicke, "Implementation of an autonomous agents system," Master's thesis, Dept. of Information and Computer Science, University of California, Irvine, Irvine, CA, Sept. 1998.
- [48] E. Gendelman, L. F. Bic, and M. B. Dillencourt, "Fast file access for fast agents," in *Proceedings, 5th International Conference on Mobile*

- Agents, MA 2001*, ser. Lecture Notes in Computer Science, G. P. Picco, Ed., vol. 2240. Berlin, Germany: Springer-Verlag, Dec. 2001, pp. 88–102.
- [49] L. Pan, L. F. Bie, and M. B. Dillencourt, “Shared variable programming beyond shared memory: Bridging distributed memory with mobile agents,” in *Proceedings of the 6th Biennial World Conference on Integrated Design and Process Technology (IDPT-2002)*, H. Ehrig, B. Kramer, and A. Ertas, Eds. Grandview, Texas: Society for Design & Process Science, June 2002.
- [50] L. Pan, L. F. Bie, M. B. Dillencourt, and M. K. Lai, “Navigational programming,” 2003, in preparation.
- [51] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, “Introduction to UPC and language specification,” IDA Center for Computing Sciences, Bowie, Md., Tech. Rep. CCS-TR-99-157, May 1999.
- [52] T. A. El-Ghazawi and S. Chauvin, *Getting Started with UPC*, High Performance Computing Laboratory, George Washington University, Washington DC, June 2001. [Online]. Available: [ftp://ftp.seas.gwu.edu/pub/upc/downloads/quick\\_start.pdf](ftp://ftp.seas.gwu.edu/pub/upc/downloads/quick_start.pdf)
- [53] “MPI-2: Extensions to the message-passing interface,” The MPI Forum, July 1997.
- [54] A.-C. Lai, C.-K. Shieh, J.-C. Ueng, Y.-T. Kok, and L.-Y. Kung, “Load balancing in software distributed shared memory systems,” in *Proceedings, IEEE International Performance, Computing, and Communications Conference (IPCCC 1997)*. Piscataway, N.J.: IEEE, Feb. 1997, pp. 152–158.
- [55] A. Itzkovitz, A. Schuster, and L. Wolfovich, “Thread migration and its applications in distributed shared memory systems,” Computer Science Dept., Technion - Israel Institute of Technology, Haifa, Israel, Tech. Rep. LPCR9603, July 1996.
- [56] H. Jiang and V. Chaudary, “MigThread: Thread migration in DSM systems,” in *Proceedings, International Conference on Parallel Processing Workshop on Compile/Runtime Techniques for Parallel Computing*. Alamos, Calif.: IEEE Computer Society, Aug. 2002, pp. 581–588.
- [57] W. Shi, W. Hu, Z. Tang, and M. R. Eskicioglu, “Dynamic task migration in home-based software DSM systems,” Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, Tech. Rep. 990004, 1999.
- [58] W. E. Weihl, E. A. Brewer, A. Colbrook, C. Dellarocas, W. C.-Y. Hsieh, A. D. Joseph, C. A. Waldspurger, and P. Wang, “PRELUDE: A system for portable parallel software,” Massachusetts Institute of Technology, Cambridge, Mass., Tech. Rep. MIT/LCS/TR-519, Oct. 1991.
- [59] K. L. Johnson, “High-performance all-software distributed shared memory,” Massachusetts Institute of Technology, Cambridge, Mass., Tech. Rep. MIT/LCS/TR-674, Dec. 1995.
- [60] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, “Supporting dynamic data structures on distributed-memory machines,” *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 233–263, Mar. 1995.