Integrating Page Replacement in a Distributed Shared Virtual Memory

Yvon Kermarrec

Télécom Bretagne
Département Informatique
Technopole de Brest - Iroise
F 29 285 Brest Cedex - France
yvon@enstb.enst-bretagne.fr

Laurent Pautet

Télécom Paris
Département Informatique
46, Rue Barrault
F 75 013 Paris - France
pautet@inf.enst.fr

Abstract

This paper presents a new algorithm for distributed shared virtual memory dedicated to diskless embedded systems. In this context, we adapt an existing algorithm in order to include a page replacement mechanism. We also propose a memory partition to optimize memory space use. In conclusion, our algorithm has a complexity comparable with the initial one.

1 Introduction

A distributed system consists of multiple autonomous processors that do not share memory, but cooperate by sending messages over a communication network. If we want the programmer to benefit from a shared address space and use the shared memory programming paradigm [2], we have to design a distributed shared virtual memory (DSVM) by implementing a software layer based upon: the local memory of each machine and a set of adequate algorithms and protocols which transmits data between machines and which guarantees memory consistency.

In this paper, we show the advantages and the issues raised in implementing a DSVM on diskless embedded systems. In this particular context, we propose an improved DSVM algorithm which integrates page replacement. We indicate elements of proof and the upper bound on the number of messages needed to satisfy a given number of concurrent page faults. Finally, we conclude with research in progress.

2 Solutions and issues

Shared memory based communication presents interesting features for embedded systems in a distributed environment [1]. But, we are aware of the limits of the shared memory communication scheme. The major drawback of such a model is the cost required to ensure memory consistency [5] [8]. Expressing synchronization is not as easy as in the message-based paradigm.

Message-based communication is not very flexible and may introduce deadlock situations. In order to cope with such a situation, application programmers have to mix deadlock detection and resolution in their code thus making the application more complex and harder to design. In this context, a shared memory may be considered as an alternative communication medium in many situations. Activities may communicate provided they access the same address space directly. In a message-based communication scheme, the data must be flattened and then all the values have to be transferred. In contrast to the message-passing model, shared information can be passed by reference rather than by value. A write (resp. read) operation on a shared variable may have an immediate effect provided the data (resp. a copy) is on the node whereas communication through a network always induces delays. The upper bound on the number of messages needed to get data is predictable and the eventual delay of transfer is known as soon as a real-time network is used. Therefore, when passing a large data structure, an embedded or real time system may prefer the DSVM model to the message-based communication scheme. The DSVM ensures performance and preserves predictability. Moreover, the design of a fault tolerant system will be easier with such a mem-

1063-6927/94 $03.00 © 1994 IEEE

ory since activity migration or global state may be stored in it. The above benefits show that a DSVM may be an interesting model for diskless embedded applications as long as developers are aware that the performance degradation depends on the number of processors and primarily on the degree of shared data updating.

2.1 Various algorithms

A distributed virtual address space is usually partitioned into pages and many algorithms maintain a strong memory coherence. A page server is present on each node of the system and the servers all together maintain the consistency of the virtual memory. When a user activity wants to access pages, it contacts its local server and two situations may occur: the page is available or not. If the page is unavailable, the server makes a read/write page fault in order to get the page from the others.

In a first approach, a server devoted to a page centralizes write and read operations. It receives these requests, executes them and sends acknowledgements or page copies. Naturally, the page server (or the page owner) may be overloaded by too many write requests especially when the page locality is not adequate. Thus, another strategy allows the page to migrate to the last client which becomes the new page owner. Read request bottleneck has a more flexible answer since the uniqueness of the page in read-mode is not required. Therefore, page replicas may be delivered by the page owner to several clients as long as the page is not modified. An invalidation protocol ensures that any write operation invalidates all page replicas.

Another approach provides full memory accesses for multiple writers as well as for multiple readers. When a node attempts to write, a sequencer receives the modification which is broadcast to owners. But, a software layer providing a DSVM service cannot match the performance of a normal local memory. Therefore, a developer has to be aware that too many write operations would overload his network. Using a shared memory with a high write/read ratio is unsuitable and we decided to discard this last algorithm.

2.2 Dynamic distributed manager algorithm

Each algorithm improves the DSVM model to allow more dynamic and more flexible situations (See [7] for more details). But, to tackle diskless systems, we chose to implement the dynamic distributed manager algorithm with distributed copyset, proposed by Li and Hudak. For our purpose, the page migration was very attractive as page localization is determined at execution time. The invalidation traffic is a small expense compared to the number of page transfers required by page replicas. The multiple writers algorithm which needs a broadcast facility was discarded as we assume a low write/read ratio. Moreover, the distributed copyset extension proposed by Li and Hudak in [5] offers the replication mechanism and bypasses broadcast facility requirement to invalidate copies.

Notations. Let T_P be the *true owner* of page P. T_P can access P in write mode. Several nodes may have a replica of the page in read mode. Each node maintains a table of records indexed by pages. Each record contains two fields, the *probable owner* and the *copy set* of the page. Let $PO_P(N)$ be the *probable owner* of page P for the node N and $CS_P(N)$ the *copy set* of page P for the node N. In the following sections, we shall consider a given page P. Thus, the P subscript will be omitted.

$PO(N)$ contains the last node to which the node N has delivered P in write-mode or to which it forwarded an acquisition request. Note that $PO(N)$ is not usually T as the node $PO(N)$ may have already delivered its page in the meantime. When a write-fault occurs, a page acquisition request R is sent to $PO(N)$. By following the sequence of POs, R will reach the final page owner T. Of course, each successive owner N maintains its $PO(N)$ properly at the value of the last node to which it has satisfied an acquisition request.

$CS(N)$ describes a set of owners to which N has delivered a copy of P. The specificity of this algorithm allows a copy owner to deliver on its own a copy of the page. It keeps track of this operation in its CS. Therefore, when a read-fault occurs in N, this node sends a copy acquisition request R to $PO(N)$. As in a write-fault situation, R follows a sequence of POs. But, this time, it only needs to find a node that holds a copy of P. Thus, if a node X has acquired a copy from T or from another node before, X sends a copy to N and adds N to its CS.

Let V be the set of nodes, E_P the set of arcs $(N, PO(N))$ for $N \in V$ and $G_P = (V, E_P)$ the graph of the probable owners for a given page P. In [5], Li and Hudak show that G_P is a tree of root T. When T wants to deliver a page in write-mode, it provides this page with its CS. The new page owner invalidates existing page copies by sending invalidation requests to each node recorded in the received CS. As this invalidation message contains the identity of the new page owner, each client updates its *PO* and propagates the invalidation request through the tree.

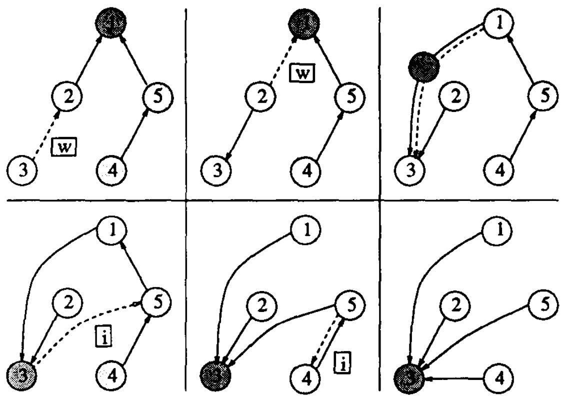

Figure 1: Write-fault in node 3

Figure 1 illustrates the different steps that occur when node 3 acquires page *P* (hold by node 1) in write-mode. Node 3 sends a write-fault request. Node 1 releases the page and sends its copy set. Then, node 3 invalidates copies using the received copy set of node 1. Plain arrows constitute the probable owner graph, dashed arrows indicate an emission of a request. *w* is a write-fault request, *H* a read-fault request and *i* an invalidation request. A dark gray node owns the page and light gray ones hold copies.

Page replacement. The fundamental point not described in [5] or in most other algorithms is page replacement. Obviously, if a node acquires too many pages in write mode, it needs to free local memory space to load new shared pages. In an operating system, unused pages are typically saved on a disk following a given strategy. Such a strategy is unsuitable in the context of diskless system ; and memory pages can only be saved in memories. Therefore, unused pages have to be saved on one of the other nodes. The difficulty is to select the "saver" node and to let other nodes know about the transfer in a minimum number of messages.

Memory partition. Our approach is to flush the page to a predefined node. Indeed, selecting a node dynamically to be saved in the backing storage the page is rather inefficient as it may require a large number of messages. Thus, each node divides into two parts the local memory dedicated to the shared memory (figure 2):

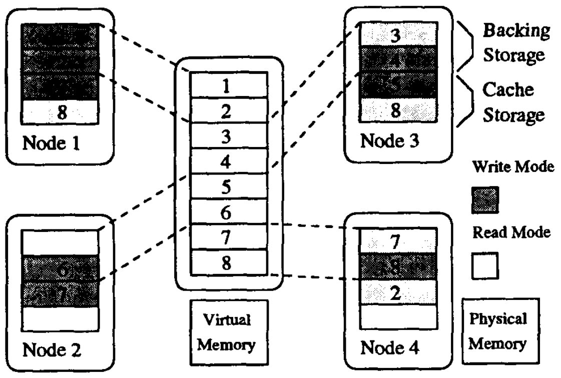

Figure 2: Memory partition

Backing storage: At initialization time, these pages are shared pages owned by the local node. We refer to this unique node (which is the page owner at initialization time) as the home node of the page. For any given page *P*, there is a unique home node H_P which is known statically by the other nodes. At execution time, when a node wants to get rid of a page, it flushes this page to the home node. Thus, this node must be able to store them in the backing storage location at any time ; The backing storage is never used to store pages or copies other than the pages or copies described at initialization time. Thus, the backing storage remains always available for flushed pages.

Cache storage : It enables the local node to load pages or copies received after an acquisition operation as a cache space does. When memory storage is needed, the node may free some of these pages and may flush them to the backing storage of their home node.

3 A new algorithm

3.1 An incremental strategy

At this level, we have to realize that the page replacement algorithm and the DSVM algorithm are no longer independent since both of them are working on the local memories. Therefore, we have to consider eventual access conflicts to the local memory and we shall analyse two approaches.

Flush without ownership. The node which flushes page *P* can be maintained virtually as the true owner of *P*. When it receives requests, it must send a dedicated request to the home node so that the home node can transmit P on request. This approach requires several exceptional situations since even the home node (which has the page without being its owner) needs to ask for the page from the page owner. Moreover, a home node may waste its memory space by owning page P in its backing storage and a copy of P in read-mode in its cache storage at the same time.

Flush with ownership. When the page owner wants to flush P, it transmits P and its CS to the dedicated home node and declares the home node to be the new PO. After reception of the page, the home node declares itself to be the true owner of the page. It updates its PO and its CS and acquires the page in read-mode. Therefore, there is no need to invalidate existing copies.

The major difficulty occurs when the home node has already sent a page request before receiving the flushed page. This problem of obsolete write operations (read operations are very similar) may introduce deadlocks as we shall explain in the next section.

Note that when the page owner wants to free memory space, it may ask the home node to acquire the future flushed page. But this solution does not optimize the number of messages and raises similar deadlocks in any case.

Deadlock situation. If we apply Li and Hudak's algorithm to figure 3, we have the following situation. Whereas node 2 decides to flush page P =3 to free local memory (after flushing, $PO(2)$ = 3), nodes 1 and 3 ask for page 3 in write mode. As $PO(1) = 2$ and $PO(3) = 2$, nodes 1 and 3 send their requests to node 2. Let us assume that the request of node 1 arrives first ; this request is forwarded to node 3. Then, $PO(2) = 1$. When the request of node 3 arrives, this request is forwarded to node 1. Therefore, we are in a deadlock situation as the two requests must wait for the completion of each other. Node 3 is the only one to know how to solve the problem as page 3 has been flushed to him. In order to unlock the situation, node 3 will force node 1 to acquire page 3. To optimize page transfer, when H needs the page and gets it because of the flushing mechanism, H becomes *true owner*. H handles its write or read operation as it applies in [5] even if H has already sent a fault request. We shall use this mechanism in our algorithm to solve the deadlock.

Obsolete requests. As described in figure 3, the home node may have sent a page request (read or write mode) in order to get a page it has received in the meantime. We have to ensure that this request which we call an *obsolete request* does not introduce errors or cycles in the probable owner graph. The obsolete request must not wander forever in the graph.

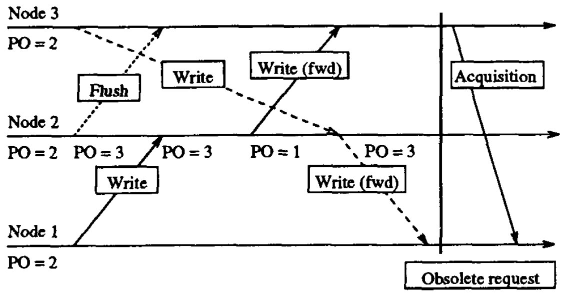

Figure 3: Deadlock situation

Firstly, in [5], nodes do not update their PO because of read mode request. Thus, an obsolete request in read mode causes no damage to the graph. A cycle may appear when the node receives such a request in write mode. But, as soon as node N has flushed the page, H becomes the root of a new tree, union of the subtree from H and a subtree constituted of the edges (N, H) and the remaining tree from N.

Lastly, we have to ensure that the obsolete request terminates and does not wander forever among the other nodes. This may happen in a very tricky situation that is too complex to describe here. We add a time stamp mechanism to our algorithm to allow nodes to detect and destroy an obsolete request.

This time stamp in our implementation is a counter, but may be implemented as a logical clock or any increasing value. For each node, we add this stamp as a new field on a given page. The same field is added to a request message or a page message if it is not already the case (some communication layers have already such a functionality). Each time the home node receives a flushed page although its page request is still in progress, this request is categorized as obsolete and the time stamp of the home node is incremented. Each node receiving any request from the home node updates its local stamp if the stamp of the received request is higher or equal to its own. If this condition is not true, the request is an obsolete request and is destroyed. Thus, an obsolete request either progresses in a subtree of a given time stamp or changes from a subtree to another subtree of a higher time stamp. The obsolete request will reach a subtree with a higher time stamp and will be destroyed. The home node may generate several obsolete requests as they do not interact with each other.

However, as the home node gets the flushed page in read-mode, copies and their time stamps are not updated. As only a copy is needed to answer a read-fault request, an obsolete read-fault request may succeed. This useless copy transfer may be avoided if H gets the flushed page in write mode and applies an invalidation mechanism. But, we prefer an useless copy transfer to this frequent invalidation mechanism.

3.2 Automata description

In the following paragraph, we sum up our algorithm. For this reason, we give the specifications of a page server as an automata. We successively give the transition table and the different definitions of events, states, actions and exceptions.

State / Event	Move	Copy	WrRd	Read	Null	InWr	InRd	InCl
Write	Stop Move	Stop Copy	Move Null	Fwd Read	Fwd Null	Stop InWr	Stop InRd	Stop InCl
Write*	Stop Move	Stop Copy	Move Null	Fwd Read	Fwd Null	Stop InWr	Stop InRd	Stop InCl
			Kill_2 WrRd	Kill_2 Read	Kill_2 Null			
Read	Stop Move	Stop Copy	Copy Read	Copy Read	Fwd Null	Stop InWr	Stop InRd	Stop InCl
Read*	Stop Move	Stop Copy	Copy Read	Copy Read	Fwd Null	Stop InWr	Stop InRd	Stop InCl
			Kill_2 WrRd	Kill_2 Read	Kill_2 Null			
Move	Store InCl	NA	NA	NA	NA	NA	NA	NA
Copy	NA	Load InRd	NA	NA	NA	NA	NA	NA
Copy*	Kill_1 Move	Load InRd	Kill_1 WrRd	Kill_1 Read	Kill_1 Null	Kill_1 InWr	Kill_1 InRd	Kill_1 InCl
		Kill_1 Copy						
Clear	ClDw_3 Move	ClDw_3 Copy	NA	ClDw_3 InCl	InCl	NA	Stop InRd	NA
	ClUp_3 Move	ClUp_3 Copy		ClUp_3 Null	Null			
ClAck	ClUp_3 Move	ClUp_3 Copy	NA	NA	NA	Start InWr	NA	ClUp_3 Null
	Wait_4 Move	Wait_4 Copy						Wait_4 InCl
Save	Load InWr	Load Read	NA	Load Read	Load Read	NA	Kill InRd	NA

Write	Acquisition request in write-mode.
Write*	Home node acquisition request in write-mode.
Read	Acquisition request in read-mode.
Read*	Home node acquisition request in read-mode.
Move	Page transfer message.
Copy	Copy transfer message.
Copy*	Copy transfer message of the home node.
Clear	Invalidation request.
ClAck	Acknowledge of an invalidation request.
Save	Flushing request.

Move	The server has sent a page acquisition request.
Copy	The server has sent a copy acquisition request.
WrRd	The server is the true page owner.
Read	The server is a copy owner.
Null	The server has neither a copy nor the page.
InWr	The server locks the page in write-mode.
InRd	The server locks a copy in read-mode.
InCl	The server locks the page during a invalidation phase.

Stop	The server blocks the request.
Store	The server loads the page and proceeds to an invalidation mecanism.
Kill	The server destroys the message.
ClDw	The server propagates an invalidation request to its copy set.
ClUp	The server propagates an invalidation acknowledge to its client.
Start	The server starts to handle the page fault.
Load	The server saves the page in its backing storage.
Move	The server sends the page.
Copy	The server sends a copy.
Fwd	The server forwards the request to its PO.
Wait	The server waits for invalidation acknowledgement.

(1)	The home node destroys a copy whose time stamp is obsolete.
(2)	The server destroys a request whose time stamp is lesser than the server one.
(3)	The server either propagates the invalidation request to its copy set or acknowledges an invalidation phase.
(4)	The server waits for the invalidation acknowledgement of each node of its copy set.

4 Elements of proof

Theorema 1 *Let D be the number of page faults in a system composed of N sites. With Li and Hudak's algorithm and for any node in the system, a page fault reaches the true page owner at the cost of $N-1$ messages. The maximum number of messages to solve D page faults is $O(N + D \times logN)$.*

Let us prove that our algorithm verifies the property:

Theorema 2 *Let S be the number of page replacements and D be the number of page faults. A page fault reaches the true page owner at the cost of N messages. The maximum number of messages to solve D page faults is $O(N + 1 + D \times log(N+1))$.*

We call T the true owner of the page which executes the first page replacement. As stated in [5], graph $G' = G - (T, PO(T))$ is a tree so that any request can reach the true owner of the page. Let G'_1 be the subtree extracted from G' whose root is H and $G'_2 = G' - G'_1 - (H, PO(H))$. After the flushing mechanism, the new graph $G'' = G'_1 \cup G'_2 \cup (T, H)$ is still a tree. Then, any request can still reach the true owner of the page.

We use induction on the number of page replacements. If the number of page replacements is zero (i.e. $S = 0$), theorem 1 is true and thus theorem 2 is true.

Let us assume that the theorem is true for S page replacements. Let us assume that $S + 1$ page replacements occur in conjunction with D page faults. Let us consider now the first of the $S + 1$ page replacements. There are two situations:

Without obsolete request This page replacement

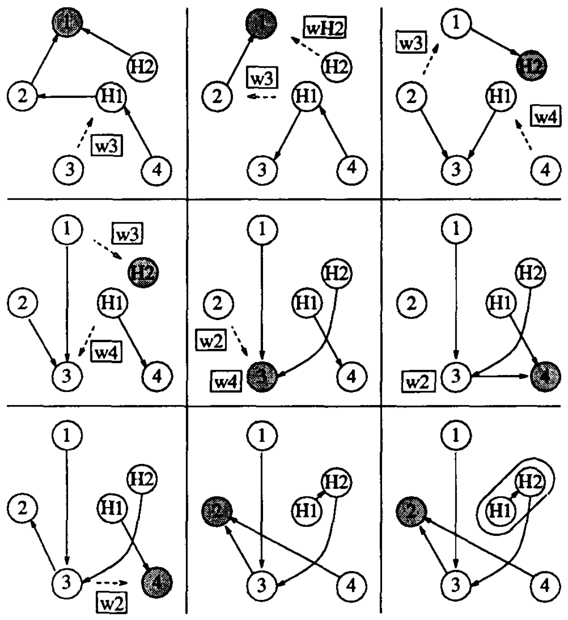

Figure 4: Without obsolete request

occurs but no page fault occurs on the home node. Figure 4 illustrates such a situation. Before node 1 flushes the page, a write-fault occurs on node 3. Then, node 4 and 2 detect a write-fault as well. For the proof, we shall construct another system of $N+1$ nodes equivalent to the current one. Node H is duplicated in two nodes H_1 and H_2. These two nodes, like the other nodes in the system, behave according to Li and Hudak's algorithm. Node H_1 has the same behavior as node H before the flushing mechanism when node H makes D_1 requests progress. Node H_2 will adopt the future behavior of H. After the flushing operation, nodes H_2 and T modify their probable owner so that $PO(T) = H_2$ and $PO(H_2) = H_2$. Before node H_2 gets the page, H_1 has forwarded D_1 requests to its probable owner. These requests progress between the nodes in the same way as they would progress in the initial system. Node H_1 behaves as a relay node since it generates no request. Its function is to forward requests to its current probable owner and to modify this value according to the incoming requests. As soon as H_2 has received the page, $PO(H_1)$ may be updated to H_2. The behavior of node H_2 can be compared to a "soothsayer" node as it sends a write request to the node which wants a page replacement. Node H_2 will get the requested page immediately. Once this page has been received, H_2 can deliver it to any requesting node. We apply the induction assumption to the virtual system which has the same properties as the one we described beforehand (except for one page replacement).

With obsolete request In this situation, the first

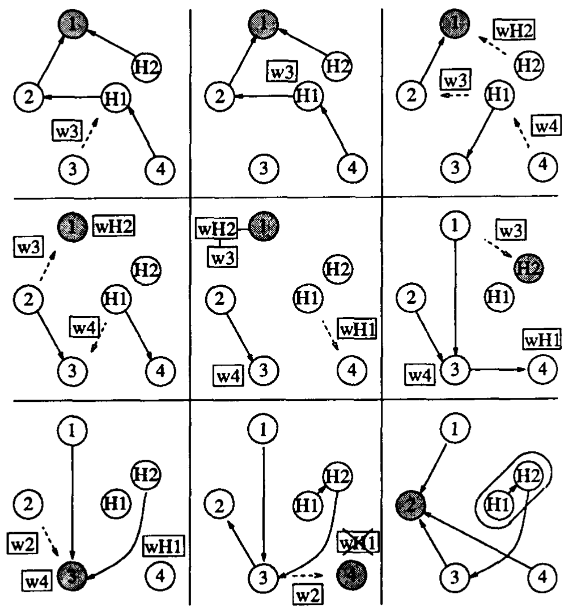

Figure 5: With obsolete request

of the $S + 1$ page replacements occurs when node H has sent itself a page fault request. Figure 5 illustrates the situation. We shall use a similar virtual system to prove the correctness of this part of the algorithm. On the one hand, according to Li and Hudak's algorithm, the request of node H follows the sequence of POs and modifies the graph. On the other, node H must block the received requests (and not forward them) so that the requests will be considered once node H has used the memory page. We have presented a solution to solve a potential deadlock situation in the original algorithm in 3. Thus, we refine the behavior of H_1 for the purpose of our algorithm. Node H_1 shall block received requests while the page has not been flushed to node H_2. As soon as node H_2 gets the page, node H_1 declares that its page-fault is solved and forwards blocked requests as $PO(H_1) = H_2$. The virtual system we have built presents the same properties as the initial one and follows the behaviour of Li and Hudak's algorithm.

Nevertheless, we have to check that the obsolete request terminates in a bound time less than N and that this request does not introduce any cycle or error in the probable owner graph. When two write-faults occur on two nodes X and Y and if request R_X is forwarded to Y, Y blocks R_X while Y has not received the page. Thus, R_X cannot bypass R_Y except if $Y = H$. In this case and in order to avoid deadlock situations, H does not wait for R_H to succeed before releasing the page. Thus, R_H is the only request that may succeed in a number of message greater than the number given in Li and Hudak's algorithm. The time stamp mechanism solves this problem.

Lemma 1 *Let V_E be the set of nodes whose time stamps is E. Let $E(N)$ be the time stamp of node $N \in V$ and R_E the obsolete request from H and whose time stamp is E. It gives:*

1. *If $E_1 \neq E_2$, then $V_{E_1} \bigcap V_{E_2} = \oslash$*
2. $\displaystyle\bigcup_{I=1}^{\infty} V_I = V$
3. *Let I be a node, then $E_I \leq E_{PO(I)}$*
4. *R_E traverse only nodes $I \in V_E$*
5. *R_E does not introduce a cycle into graph G.*

Proof: The first two properties are verified immediately. The proof of the third one can be obtained by induction on page faults. At initialization, the proposition is true. Only invalidation and write requests modify the probable owner value. Invalidation requests indicate the owner of the original page which has the highest time stamp. The proposition can be asserted for invalidation requests. So let us consider the write requests. If this request is an obsolete request generated by node H and if it arrives on node I and $E < E_I$, then this request is destroyed and $PO(I)$ is not modified. In the situation where $E_I \leq E$, once the obsolete request has traversed the node, $PO(I) = H$ and $E_H = E$ or $E + 1$ (depending on the order). If this request comes from node $J \neq H$, then after the request forwarding we have $PO(I) = J$. Nevertheless, when node J obtains the page, node $PO(J)$ gets the page time stamp and $E_{PO(J)}$ becomes maximum. One prooves proposition 3.

The future probable owners of H will get the page and E as its time stamp. As long as there is no obsolete request, probable owners of H belong to V_E. When H sends its obsolete request R_E, it will send it to $PO(H) \in V_E$, But, with the second proposition, the time stamps of the traversed nodes cannot decrease. Thus $R_E \in V_{\{E' \| E \leq E'\}}$. If R_E is forwarded to node N where $E_N > E$, then N destroys R_E. It gives proposition 4, $R_E \in V_E$.

Only R_E, the obsolete write request can introduce a cycle. $R_E \in V_E$. If $N \in V_E$ and if N forward R_E, then $PO(N) = H$ thus $E = E_N \leq e_{PO(N)} = E_H$ (Proposition 3). If one cycle is created, then a path exits between H and N. According to proposition 3, $E_H \leq E_{PO(H)} \leq \cdots \leq E_N$. Thus, $E_N = E = E_H$ and that is impossible.

As there is no cycle in V_E (as created by Li and Hudak's algorithm), the obsolete request generates $\|V_E\|$ messages at most before being destroyed. $R_E \in V_E$ and $\bigcup_{I=1}^{\infty} V_I = V$ indicates that the home node can generate several obsolete requests in parallel without waiting for an acknowledgement. Therefore, if several obsolete requests occur during page replacement, node H will send each concurrent request only to a subgraph V_E. The number of messages generated in this case is less than the number of messages for a standard page fault.

We have established that the obsolete request does interact with the graph of probable owners in a consistent way and that this request reduces the graph as the invalidation mechanism does. The number of generated messages is bound. Finally, the obsolete request follows a path that would have been similar without the page replacement. The major differences are the lack of page transfers towards the home node and the destruction of obsolete requests.

Li and Hudak assert that a set of D concurrent page faults can be handled as an equivalent set of D sequential page faults. In fact, when a page request arrives on node N which has a page fault itself, the page fault will not be considered until all page faults of N are terminated. This decomposition does not apply in our algorithm because of the peculiar behavior of node H, but we proove that the time stamp mechanism solves this problem. As we introduce new virtual nodes, we can apply the complexity to this system: each node reacts as the nodes in [5].

To establish the number of required messages in the case of D page faults in a system composed of N nodes, Li and Hudak use Tarjan's results [6] in the case of the union of sets. These algorithms use a notion that is similar to probable owner notion. Li and Hudak base their complexity computation on Tarjan's results. Therefore, their formula can be reused in our context even if the problem is slightly different. The formula does not take into account the reasons for modifying the probable owner but it takes into account the fact that each node tries to become the root of the graph. The changes of probable owner through forseing has no impact on the complexity as long as requests follow the same path as in the original algorithm. As we have established that the algorithm behavior can be simulated by adding a virtual node, we can apply the result to a system composed of $N + 1$ nodes. Thus, we can determine the formula in theorem 2.

5 Conclusions

We have proposed and proved a new algorithm for DVSM management. We have adapted Li and Hudak's algorithm by incorporating an original strategy for page replacement. In fact, our algorithm combines the simplicity and efficiency of Li and Hudak's with architecture generalization (i.e., no assumption on the existence of disk units). We have evaluated the complexity of our solution and found it to be in the same order as the initial algorithm. Moreover, the programmer can benefit from the shared memory paradigm on massively parallel machines and also on embedded systems. These classes of architecture are traditionally denied DVSM implementation as they do not always offer disk units. The user can profit by selecting the most adequate paradigm for his application (i.e., either the message passing approach or the shared memory scheme) and this option reduces sofware implementation complexity.

We have implemented our algorithm as a reusable sofware component and have offered the user high level memory operations [3]. This component is an element of a more general repository which holds various entities for distributed system programming and education [4]. Among the key features, the programmer can find : distributed programming paradigms (DVSM, message passing with high level primitives and the Linda model) and distributed control algorithms (e.g., mutual exclusion, termination and deadlock detection). Our main goal consists in proposing numerous software components so that an application can be partly built from these tested elements.

The research in progress aims at developing our sofware repository and also at using our DVSM component as a foundation. We are currently investigating the use of DVSM as the communication facility for our Linda implementation and we are considering fault tolerance issues. We also plan to use our DVSM modules for the implementation of the shared variables of Ada 9X.

References

[1] H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computing systems. *ACM computing surveys*, 21(3):260–322, septembe 1989.

[2] G.Andrews and F. Schneider. Concepts and notations for concurrent programming. *ACM computing surveys*, 15(1):3–44, 1983.

[3] Y. Kermarrec and L. Pautet. A distributed shared virtual memory for Ada 83 and Ada 9X applications. In *Proceedings of the TRI Ada 93 conference*, Seattle, Washington, September 1993. ACM SigAda.

[4] Y. Kermarrec and L. Pautet. Ada reusable software components for education in distributed systems and applications. In *Proceedings of the 7th SEI conference on Software Engineering Education*, number 750 in Lectures Notes in Computer Science, pages 77–96, San Antonio, Texas, January 1994. ACM IEEE, Springer Verlag.

[5] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM transactions on computer systems*, 7(4):321–359, november 1989.

[6] R.E.Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of ACM*, 31(2):245–281, April 1984.

[7] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, pages 54-63, May 1990.

[8] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on parallel and distributed systems*, 3(5):540–554, september 1992.