Lightweight Process Migration and Memory Prefetching in openMosix *

Roy S.C. Ho, Cho-Li Wang, and Francis C.M. Lau
Department of Computer Science
The University of Hong Kong
{scho,clwang,fcmlau}@cs.hku.hk

Abstract

We propose a lightweight process migration mechanism and an adaptive memory prefetching scheme called AMPoM *(Adaptive Memory Prefetching in openMosix), whose goal is to reduce the migration freeze time in openMosix while ensuring the execution efficiency of migrants. To minimize the freeze time, our system transfers only a few pages to the destination node during process migration. After the migration, AMPoM analyzes the spatial locality of memory access and iteratively prefetches memory pages from remote to hide the latency of inter-node page faults. AMPoM adopts a unique algorithm to decide which and how many pages to prefetch. It tends to prefetch more aggressively when a sequential access pattern is developed, when the paging rate of the process is high or when the network is busy. This advanced strategy makes AMPoM highly adaptive to different application behaviors and system dynamics. The HPC Challenge benchmark results show that AMPoM can avoid 98% of migration freeze time while preventing 85-99% of page fault requests after the migration. Compared to openMosix which does not have remote page fault, AMPoM induces a modest overhead of 0-5% additional runtime. When the working set of a migrant is small, AMPoM outperforms openMosix considerably due to the reduced amount of data transfer. These results indicate that by exploiting memory access locality and prefetching, process migration can be a lightweight operation with little software overhead in remote paging.*

1 Introduction

Process migration is the act of transferring a running process from one machine to another. It enables dynamic load balancing, improved locality of data access and inter-process communication, fault resilience, and facilitates system administration when nodes are taken offline for maintenance [14]. Because of these advantages, there had been active research on process migration during '80s and '90s, which includes DEMOS/MP [17], Amoeba [21], V [22], to name a few. Despite these efforts, process migration has not been widely used, mainly due to the complexity of implementing the migration support in commodity operating systems which were designed for stand-alone operations. Research focus was then shifted to process checkpointing (e.g., [7]), which offers a compromise between ease of implementation and versatility. With recent advances of computing technologies, however, process migration is again gaining popularity. There are three main factors that have led to this change.

1. **Cluster, grid, and mobile computing**. HPC clusters having thousands of compute nodes with changing loads; globally-scattered computing resources in grids having intermittent availability; mobile users roaming around, each interacting with multiple, mobile applications—all these factors have made modern distributed systems much larger in size and more dynamic than ever before, and this trend is continuing. Traditional mechanisms to place and re-place tasks (e.g., queueing systems and checkpointing) lack the flexibility to cope with such changes. It is desirable to have a more capable load distribution mechanism that can better utilize these resources. Process migration appears to be a promising alternative.

2. **From server-based to decentralized architectures**. Due to the factor above, server-based architectures with limited scalability have been gradually replaced by decentralized or peer-to-peer architectures. Process migration *per se* does not require server coordination (unlike checkpointing, for example, which needs a file server) and is more suitable to highly scalable environments.

*This research was supported by a Hong Kong RGC grant (HKU 7176/06E) and a China 863 grant (2006AA01A111).

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

3. **The widening gap between CPU and wide-area network speeds**. The gap between the speeds of CPUs and wide-area networks have been widening over the past decades. On the other hand, while the size of program code has been quite stable, the size of data programs process has increased dramatically due to the new demands in computational sciences, yet these data might be scattered over geographic regions (as in DataGrid [2], for example). It is increasingly expensive to copy data from or communicate with distant processes. Process migration, which emphasizes the mobility of the process itself, seems to be a promising approach to bridge the gap.

However, process migration can be an expensive operation if the entire address space (or all dirty pages) of a process is transferred to the destination node before its execution is resumed. For example, openMosix [1], an open-source distributed system that supports process migration, adopts this approach; and it can take tens of seconds to migrate a 500MB large process through a commodity network (see Section 5 for details). The long migration latency can lead to rather conservative designs of upper-level scheduling policies. For instance, [10] migrates a process only if its lifetime exceeds a certain threshold. Besides, it is also not cost-worthy to migrate the entire process if we are not sure how long computing resources will be available at the destination node; a wrong or suboptimal migration decision would require the process being migrated again, inducing even longer "freeze time" in which no computation can be done. Several studies (e.g., [16][19]) have proposed postponing the transfer of memory pages until they are accessed by migrants. While this approach keeps freeze time minimal, migrants are often executed in suboptimal performance because of the time-consuming inter-node page faults. As a result, process migration has appeared beneficial to a narrow range of compute-intensive applications. To solve these issues, it is necessary to achieve a better trade-off between the conflicting goals of low freeze time and high execution efficiency of migrants.

In this paper, we propose a lightweight process migration mechanism and an adaptive data prefetching algorithm called **AMPoM** (adaptive memory prefetching in openMosix), whose goal is to reduce migration freeze time in openMosix while ensuring execution efficiency of migrants. To minimize the freeze time, we transfer a minimal portion of the address space (the currently-accessed code, stack, and data pages) to the destination node during process migration. After the process is migrated, AMPoM analyzes the spatial locality of its memory references and iteratively prefetches memory pages from remote to hide the round-trip latency caused by page faults. AMPoM adopts a unique strategy to decide which and how many pages to prefetch. Specifically, it tends to prefetch more aggressively when a sequential access pattern is developed, when the paging rate of the process is high or when the network is busy. This advanced strategy makes AMPoM highly adaptive to different application behaviors and system dynamics. We have implemented AMPoM in openMosix and evaluated its performance using the HPC Challenge benchmark [13]. The results indicate that AMPoM can reduce the migration freeze time significantly, while maintaining the execution efficiency of migrants under all tested scenarios.

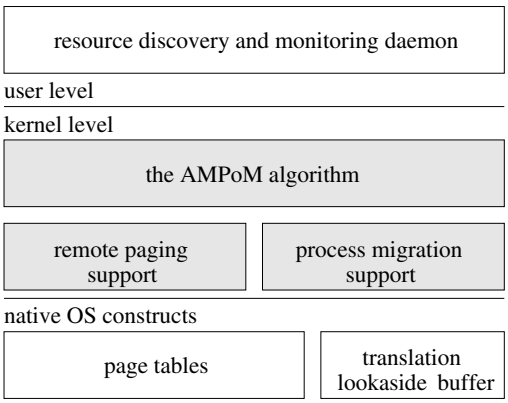

Figure 1. System design

The rest of this paper is organized as follow. Section 2 describes the modified design of openMosix. Section 3 presents the AMPoM algorithm. Section 4 provides the implementation details. Section 5 presents the evaluation methodology and the experimental results. Section 6 discusses the related work. Finally, we conclude this paper and outline several future work in Section 7.

2 System design

Our primary design objective is to support lightweight process migration in openMosix while ensuring execution efficiency of migrants. To accomplish this, we modified the process migration support in openMosix in order to minimize data transfer during migrations. In addition, we introduced two modules in the openMosix kernel, namely the *remote paging support* and the *AMPoM algorithm*. Figure 1 illustrates the overall system design; the details of each core module

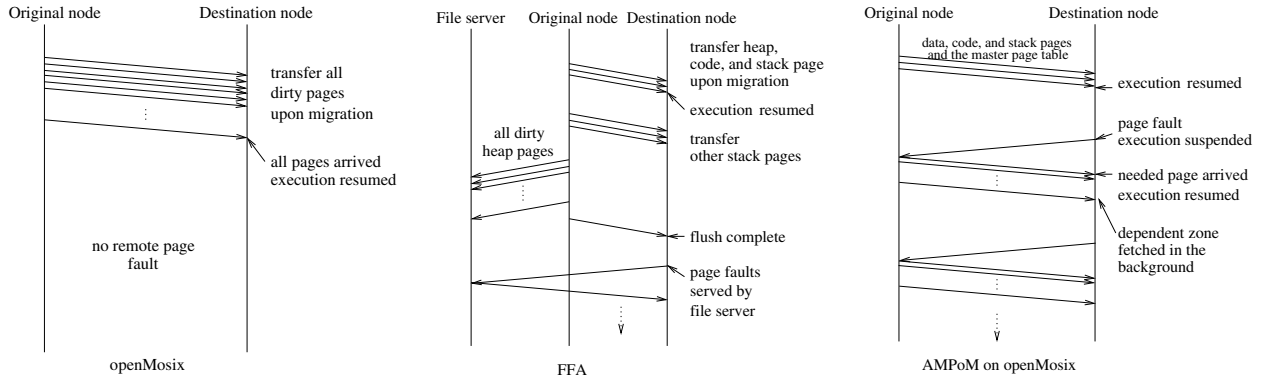

Figure 2. Different migration mechanisms

are described below.

2.1 Process migration support

In openMosix, all dirty pages in the address space are transferred to the destination node during migration. Because the dirty pages usually dominate the address space, the freeze time in this approach would grow almost linearly with the size of the address space. To shorten the freeze time, we adopt a variant of the Freeze Free Algorithm (FFA) proposed by Roush et al. [19] to decide which pages to transfer during migration. The original FFA migrates only three pages (the current data (heap), code, and stack pages) during the migration time, then resumes the execution of the migrant. After that, the original node of the process would push the remaining stack pages to the migrant, and flush all dirty pages to a file server. When the migrant encounters page faults, it would fetch the missing pages from the file server. In our design, we migrate the same three pages *and* the *master page table* (MPT) during migration, while keeping all remaining pages in the original node. When the migrant encounters page faults, it would fetch the missing pages directly from the original node, without going through a file server. Figure 2 illustrates the difference between openMosix, FFA, and our approach.

2.2 Remote paging support

The remote paging support provides a mechanism to request pages that are stored in the original node of the process. It incorporates two page tables, namely the MPT and the *home page table* (HPT). When a process is migrated, its page table in the Linux kernel will be transferred to the destination node, which will become the MPT of the migrant. At the same time, the original page table will become the HPT, and the original process instance will be switched to a "deputy" process which only answers remote paging requests and executes system calls on behalf of the migrant [1]. When a page is transferred to the migrant (either during migration or through subsequent page faults), its copy in the original node will be deleted and the HPT will be updated accordingly. When a page is created by a migrant, only the MPT needs to be updated. When a page is unmapped, however, there would be two possibilities: if the page is stored in the original node, both the MPT and the HPT will be updated, otherwise only the MPT will be updated.

2.3 AMPoM and zone partitioning

One important difference between FFA and our approach is that we exploit data prefetching to request memory pages from remote *before* they are accessed by the migrant, so as to avoid the stalls caused by the round-trip latency in remote page faults. Specifically, the proposed AMPoM algorithm dynamically identifies the current *dependent zone* of the migrant, which contains pages that are likely to be accessed in near future. If some of these pages are not stored locally, AMPoM would use the remote paging support to fetch and prefetch the missing pages from the original node. We will discuss the internals of AMPoM in Section 3.

2.4 Resource discovery and monitoring daemon

The resource discovery and monitoring daemon is a modified version of the `oM_infoD` (information daemon) in the original openMosix. It discovers CPU and network resources available in the system and monitors their current utilization. It also provides these information to our AMPoM algorithm for decision making.

3 Design of the AMPoM algorithm

Conceptually, AMPoM is a *conservative* prefetching scheme [6] which tries to prefetch *just enough* pages for the process to execute in a short period of time ahead. It adopts Algorithm 1 to determine the current dependent zone and perform prefetching for the migrant. We present the operations of the algorithm in the following sub-sections.

3.1 Definitions and notations

AMPoM identifies the dependent zone by analyzing the spatial locality of memory access in the page level. The analysis is based on a *stream* of addresses of recently-accessed memory pages recorded in a fixed-size lookback window W of length l. Specifically, W records a reference stream of pages $R = r_1, r_2, ..., r_l$, where r_i denotes the address of the page being accessed in the i-th page fault. When a page fault occurs while the lookback window is full, the first element (i.e., r_1) in the window will be discarded, all other elements will be shifted left, and the address of the newly accessed page will be appended as the new r_l. In addition to the lookback window, AMPoM maintains two other arrays, T and C. T contains the access time of each page recorded in W, therefore $T = T_1, T_2, ..., T_l$, where T_i is the access time of r_i recorded in W. $C = C_1, C_2, ..., C_l$, where C_i is the current CPU utilization when r_i is recorded in W.

Spatial locality is the tendency of applications to access memory addresses near other recently accessed addresses [5]. This definition is extended to memory pages where spatial locality is the tendency to access pages near other recently accessed pages. Specifically, a *stride* of a page reference r_p is defined as the minimum absolute distance d in W between the references to r_p and r_p+1. Therefore, a *stride-d* memory reference is $S_d = r_p, r_{p+1}, r_{p+2}, ..., r_{p+d}$, where $r_{p+d} = (r_p + 1)$, $r_i \neq (r_p+1)$ for all $p < i < p+d$, $d, p \geq 1$, and $p+d \leq l$. For example, the access stream {1,99,2,45,3,78,4} contains three stride-2 references ({**1**,99,**2**}, {**2**,45,**3**}, and {**3**,78,**4**}). Based on the stride construct, $stride_d$ is defined as the total number of page references in W which exhibit *stride-d* references. In the above example (which assumes $l = 7$), $stride_2 = 4$ because there are four pages (1,2,3,4) accessed in a stride-2 pattern. It should be noted that we consider consecutive, repeated references to the same page a form of temporal locality, therefore they are counted as a single page reference. In other words, $r_p \neq r_{p+1}$ for all $1 \leq p < l$.

Algorithm 1: AMPoM's prefetching algorithm

```
foreach page fault i do
    if pages prefetched last time have arrived then
        copy these pages to the migrant's address space;
    end
    record i in the lookback window;
    calculate the current spatial locality score;
    calculate the number of pages in the dependent zone;
    identify which pages are in the dependent zone;
    foreach page j in the dependent zone do
        if j is not stored locally then
            record j in the remote paging request;
        end
    end
    send out the recorded paging request to the original node;
    wait for i to arrive if it is not available locally;
end
```

3.2 The spatial locality score

The *spatial locality score* S of a process is defined as the summation of the fraction of strided references in W [23] (Equation 1). Because programs seldom have large d, AMPoM analyzes only up to *stride-*d_{max} references in W, where $1 < d_{max} < l$.

$$S = \sum_{d=1}^{d_{max}} \frac{stride_d}{l \times d} \quad (1)$$

Since S is a normalized score in the range of $[0, 1]$, it can be used to describe how much a process exhibits spatial locality. For example, a process only does sequential access to consecutive pages (e.g., consider the access stream: {1,2,3,4...}) has $S = 1$. Another example, {10,99,11,34,12,85} only has one stride-2 reference stream {10,11,12} (3 pages), therefore $stride_2 = 3$, $stride_d = 0$ for all $d \neq 2$, and $S = stride_2/(6 \times 2) = 0.25$.

3.3 Deciding the number of pages in the dependent zone

Based on the spatial locality score, we determine how many pages (N) are considered as "dependent" which should be made available in the physical memory in order to support the process to execute for a time period of t. Our analysis is based on an intuition that the more spatial locality a process exhibits, the

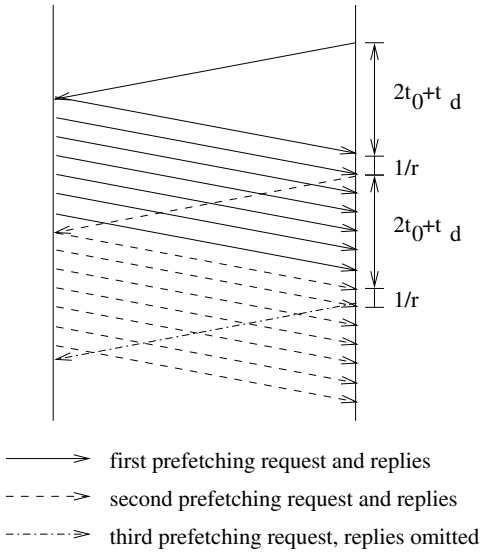

Figure 3. Network latency to be hided

more pages should be considered dependent and *read ahead*. Naturally, N should also grow with the paging rate of a process and the CPU power available to the process, because if the paging rate or the available CPU power is high, the process will be able to "consume" pages faster. N is therefore defined as follow:

$$N = \frac{c'}{c} S \times r \times t \tag{2}$$

r denotes the average paging rate of the process during the period of the lookback window W, i.e., $r = l/(T_l - T_1)$; c denotes the percentage of CPU time consumed by the process over the same period, which is calculated as $\sum_{i=1}^{l} C_i/l$; c' is the *expected* percentage of CPU time a process can possibly consume during the next period of t, which is assumed to be equal to C_l. In order to prefetch the necessary pages and make them available to the process within a time window of the network delay (i.e., to hide the round-trip latency), t should be at least equal to the round trip time between the destination and original nodes, plus the data transfer time of a single page and the time for the next analysis to take place (i.e., the next page fault). So, we define $t = 2t_0 + t_d + 1/r$, where t_0 is the network latency between the two nodes, t_d is the time to transfer a page, $1/r$ is the time when the next analysis of dependent zone occurs. The formulation of t is illustrated in Figure 3. Based on the above and from Equation 2, N is formulated as follow.

$$N = \frac{c'}{c} S \times r \times (2t_0 + t_d + \frac{1}{r}) \tag{3}$$

3.4 Deciding which pages are in the dependent zone

Equation 3 defines how many pages are considered as dependent pages. To determine which pages are dependent, we first identify the *prefetch pivots* of those *outstanding streams* of strided references in W. Specifically, an outstanding *stride-d* stream is a reference stream $S_d = r_p, r_{p+1}, r_{p+2}, ..., r_{p+d}$, where $r_{p+d} = r_p + 1$ *and* $(p + d) > l - d$. In such an outstanding stream, the prefetch pivot is $r_{p+d} + 1$. Once AMPoM identifies a prefetch pivot, it will consider the pivot and possibly the pages immediately following it (i.e., $r_{p+d} + 1, r_{p+d} + 2, ...$) as "dependent". For example, suppose $l = 10$ and the addresses of the pages accessed are $\{13, 27, 7, 8, 14, 8, 3, 15, 4, 5\}$ (5 is currently accessed), the outstanding streams are $\{14, 15\}$ (a stride-3 stream), $\{3, 4\}$ (a stride-2 stream), and $\{4, 5\}$ (a stride 1 stream), while their corresponding pivots are 16, 5 and 6, respectively. Note that the stream $\{7, 8\}$ will not be counted for prefetching because it is not "outstanding" anymore. For each pivot, AMPoM considers N/m pages immediately following it dependent, where m is the total number of outstanding streams. If there is no outstanding stream found in W, AMPoM would consider the N pages following the last referenced page (i.e., $r_l+1, r_l+2, ..., r_l+N$) dependent, imitating the read ahead policy of the Linux virtual memory manager. If a page is considered as a dependent page in multiple outstanding streams, the "saved quota" will be used to prefetch more subsequent pages.

3.5 Summary

To summarize, AMPoM adjusts its aggressiveness of prefetching according to the spatial locality score, the paging rate, and the current utilization of CPU and network resources. For processes that exhibit little or no spatial locality (i.e., $S \approx 0$), AMPoM would reduce the aggressiveness of prefetching to avoid excessive data transfer which might not be useful. To some extent, AMPoM also preserves good temporal locality as it prefetches the currently accessed code, data, and stack pages during the migration time (as described in Section 2.1). We adopt this simple heuristic because an accurate analysis of temporal locality requires expensive page protection mechanism for recording page reuse. Despite its simplicity, the experimental results in Section 5 show that AMPoM can effectively avoid page fault requests under different memory access patterns.

4 Implementation

We have implemented the AMPoM algorithm and the related supports in openMosix 2.4.26-1. We leverage openMosix's routines to handle the tedious tasks to capture and restore process states such as CPU registers, the process control block, etc. in the migration process. To shorten the freeze time, we removed the step to sending all dirty pages, and replaced it with a custom function which transfers only the MPT and the currently-accessed code, data, and stack pages. To support remote paging and prefetching, we incorporated the AMPoM algorithm and a set of network communication functions into the page fault handler in Linux. In our implementation, we maintain a lookback window of length 20. This size, although is admittedly arbitrary, is intended to be small so that the analysis overhead could be limited. In addition, we limit to search for *stride-1* to *stride-4* sequential memory accesses in the lookback window (i.e., $d_{max} = 4$). This value should be able to capture most sequential memory access because most programs perform at most two-level indirect memory references (i.e., *stride-3* at most). Besides, we also added several routines into openMosix's `oM_infoD`, which monitors the current round-trip time and available network bandwidth (t_0 and t_d in Equation 3). The round-trip time is found by measuring how long it would take to receive an acknowledgement from a remote node after a load update is sent out from the `oM_infoD`. The available network bandwidth is determined by a comparison of the current and past values of the "RX/TX bytes" fields outputted by the `/sbin/ifconfig` command. This comparison is done every time when the lookback window is "looped" once, while the time elapsed during this period can be calculated from T_1 and T_l stored in the T timer array. We use the original support of `oM_infoD` to monitor the current CPU utilization of the system.

5 Experimental results

We present in this section the results of the benchmark experiments.

5.1 Evaluation methodology

We chose the HPC Challenge (HPCC) [13] benchmark suite to evaluate the performance of AMPoM, which is composed of seven benchmark kernels: HPL, DGEMM, STREAM, RandomAccess, PTRANS, FFT, and `b_eff` (MPI latency/bandwidth test). We skipped the HPL, PTRANS and `b_eff` kernels because network communication performance in parallel programs is not the focus of AMPoM, and that the remaining four kernels are enough to represent different memory access patterns. Figure 4 presents a conceptual relationship between the four kernels used and their localities of memory reference. As shown in the figure, they represent different degrees of spatial and temporal localities that bound the behavior and performance of most applications. One might refer to [13] for the detailed operation of these four kernels.

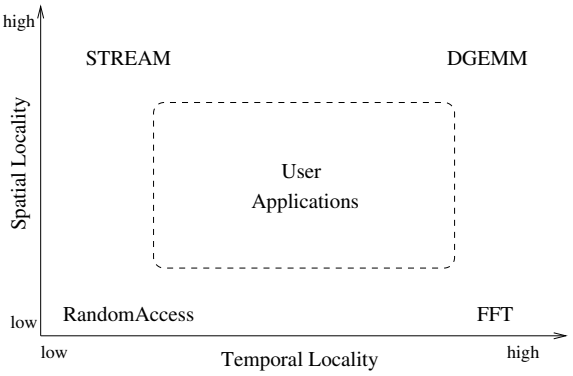

Figure 4. HPCC kernels and localities

In the experiments, each kernel was executed with different problem sizes. Table 1 lists the problem sizes specified in the configuration file of HPCC and the corresponding memory sizes. The intention of these configurations is to cover the program sizes about evenly in the range of 100MB to 500MB. In each experiment, we initiated migration right after a kernel has finished allocated the required memory. We conducted HPCC benchmark on AMPoM, and compare the results with that of the unmodified openMosix and a variant of FFA in which the same three pages (code, stack, and data) would still be transferred during migration, but all missing pages would be fetched (without prefetch) from the original node rather than from the file server. We denote this last approach as "No-Prefetch". The experiments were conducted in the HKU Gideon 300 Cluster [3]. The cluster consists of 300 Intel Pentium 4 2GHz PCs (each has 512MB RAM) interconnected by a Fast Ethernet network. The OS is Fedora Core 1 with the Linux kernel patched with openMosix 2.4.26-1. `gcc` 3.3.2 and openMosix user tools 0.3.6-2 were used in compiling the programs and initiating the process migrations. It should be noted that because the latest stable release of openMosix relies on Linux kernel 2.4, we used a rather old OS distribution and the bundled `gcc` compiler.

DGEMM	problem size	7600	10850	13350	15450	17350
	memory size (MB)	115	230	345	460	575
STREAM	problem size	7750	11000	13450	15520	17400
	memory size (MB)	115	230	345	460	575
RandomAccess & FFT	problem size	8000	11000	16000	23000	
	memory size (MB)	65	129	260	513	

Table 1. Problem and memory sizes of HPCC

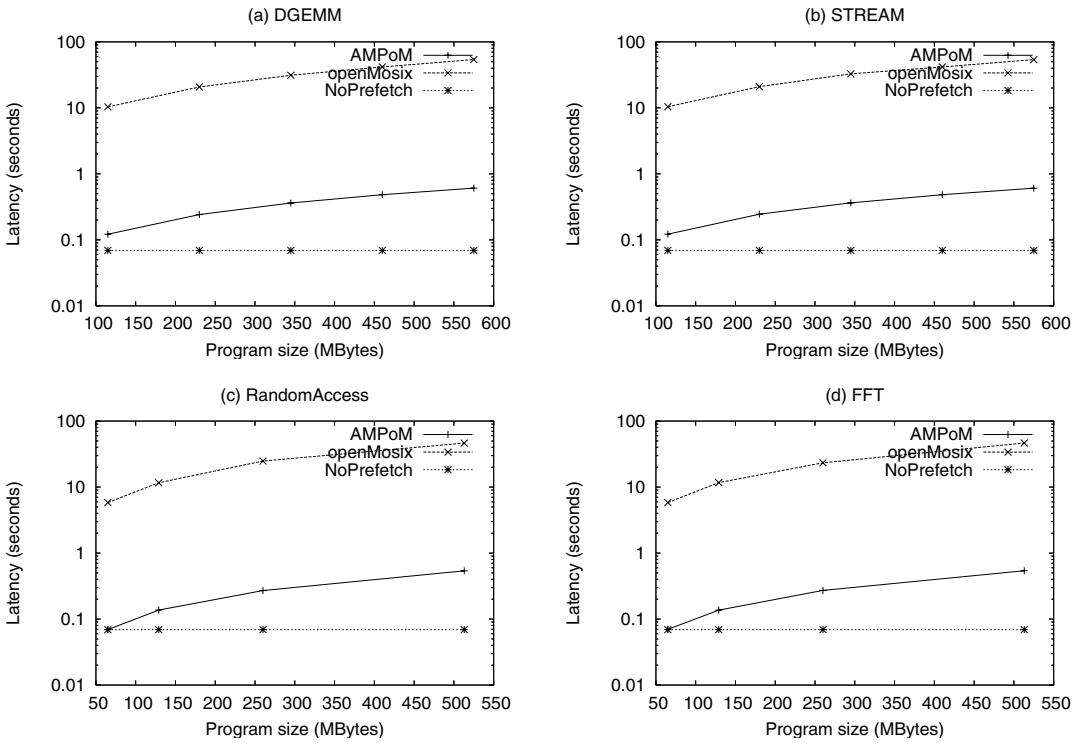

Figure 5. Migration latencies of AMPoM, openMosix, and NoPrefetch

5.2 Migration freeze time

Figure 5 reports the freeze time of the three migration schemes. As shown in the figure, openMosix's freeze time is considerably longer than the other two approaches since it has to suspend process execution and migrate all dirty pages at one time. It is also clear that openMosix's freeze time grows linearly with the program size. NoPrefetch, which migrates only three pages no matter how large the program is, shows shortest freeze times in all experiments. AMPoM's freeze times also grows linearly with the program size because AMPoM has to migrate the master page table (MPT) whose size is proportional to the number of dirty pages of the process (note: the size of an MPT is 6 bytes per page). Despite this, AMPoM is still much faster than openMosix in migration. For instance, the time to migrate a 575MB DGEMM kernel in AMPoM, openMosix, and NoPrefetch are 0.6s, 53.9s, and 0.07s, respectively.

5.3 Application Performance

Figure 6 reports the total execution time of HPCC with different migration mechanisms. As shown in the figure, AMPoM and openMosix achieve similar results, while the performance of NoPrefetch clearly lags behind. The poor performance of NoPrefetch indicates the high cost of inter-node page faults. Specifically, NoPrefetch when compared to openMosix (which does not have inter-node page faults) takes an additional 35%, 51%, 20%, 41% of time to execute the largest DGEMM, STREAM, RandomAccess, and FFT, respectively.

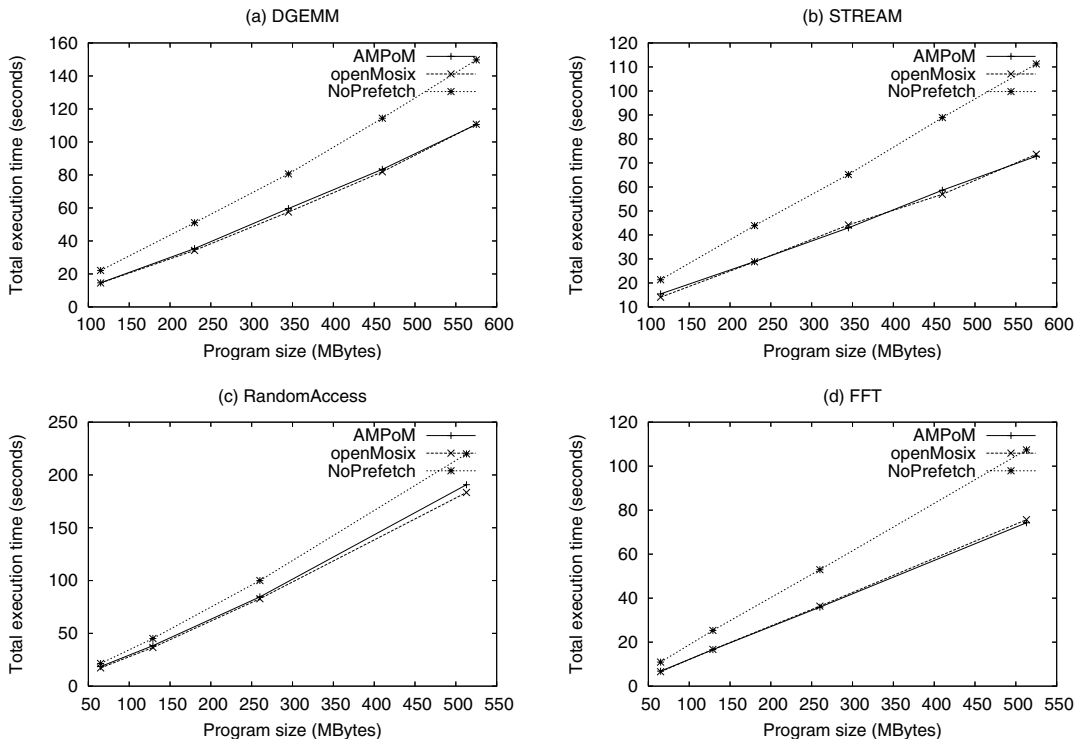

Figure 6. Application performance on AMPoM, openMosix, and NoPrefetch

By contrast, AMPoM's prefetching scheme is proved effective enough to hide most latency incurred in inter-node page faults, making its performance very close to that of openMosix which can be considered as the optimal case. The only exception is RandomAccess in which the prefetching scheme, which relies on spatial locality of memory access, fails to enhance the performance. In this case, AMPoM takes an additional 4% of time to finish the largest RandomAccess test. However, AMPoM still performs considerably better than NoPrefetch because it might still try to prefetch certain number of pages once there are some sequential accesses appear in the lookback window by chance. Although this is not the intention of the design, it resembles the characteristics of a fixed-size read-ahead policy (e.g., in Linux's buffer cache), which serves as a "baseline" of prefetching aggressiveness even when the access pattern is not clear.

5.4 Effectiveness of prefetching

Figure 7 shows the number of page fault requests in AMPoM and NoPrefetch. As shown in the figure, AMPoM's prefetching scheme manages to prevent 98%, 99%, 85%, 97% of page fault requests in the largest run of DGEMM, STREAM, RandomAccess, and FFT, respectively. The reduction of number of page fault requests leads to AMPoM's much better run-time performance compared to NoPrefetch as shown in Figure 6. It should be noted that a process in AMPoM still has to wait for the pages to come since the network speed is considerably lower than the speed of memory consumption in HPCC (which is designed to stress test memory performance). However, AMPoM's prefetching scheme saves the round trip latency of inter-node page faults by pipelining effect.

The above experiment shows that AMPoM's prefetching scheme can effectively avoid page fault requests. However, one design goal of the prefetching scheme is not to perform *excessive* prefetching. Ideally, the scheme should adjust to the memory access pattern such that prefetching is done only when it is beneficial. In this way, no system resource is wasted in unnecessary prefetching, and a migrant can be kept lightweight when it has to migrate to another node. We investigated into the ability of AMPoM to adapt to different access patterns, and the results are shown in Figure 8. As shown in the figure, AMPoM aggressively prefetches pages when a sequential pattern is clearly developed (as in STREAM), and tends to be conservative when the benefit of prefetch-

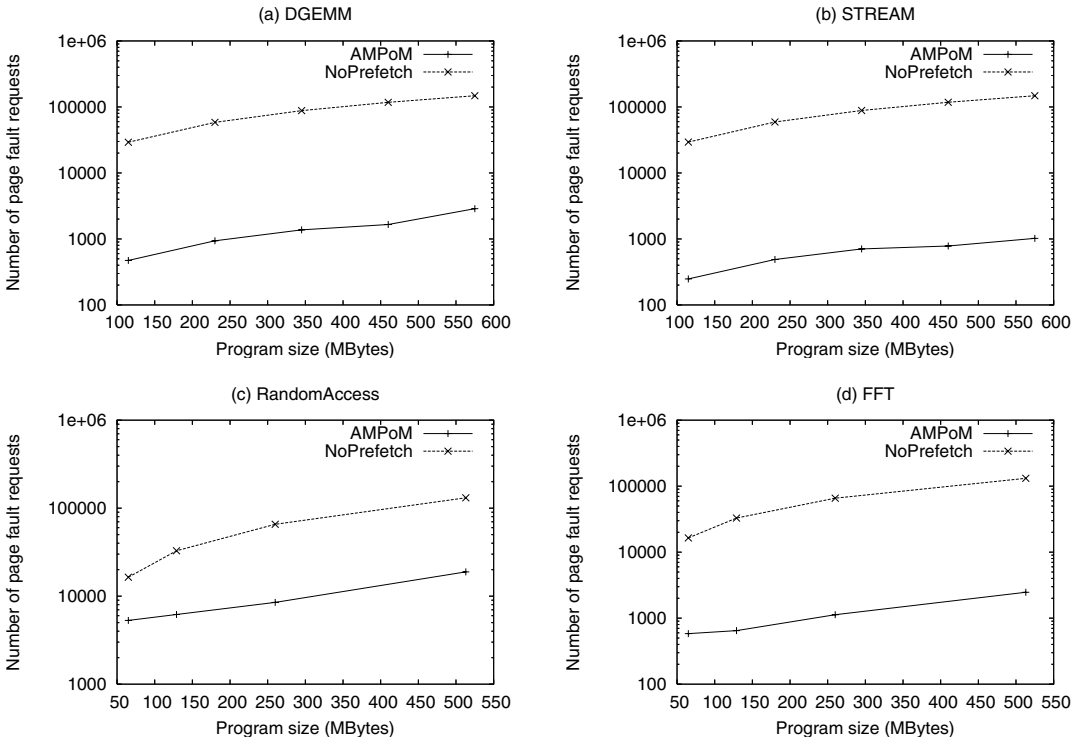

Figure 7. Number of page faults

ing is unclear (as in RandomAccess). These results closely resemble the access locality of the HPCC design as shown in Figure 4. There is an interesting point in this experiment: although AMPoM prefetches considerably less pages (per page fault) in DGEMM and FFT than in STREAM, they still execute almost as fast as openMosix (Figure 6). This is actually because DGEMM and FFT have more computation (per data item) and hence lower paging rate than STREAM [13]. This finding shows that AMPoM is able to adjust the aggressiveness of prefetching according to the paging rate. Considering the paging rate a measure of the demand of memory bandwidth of applications, AMPoM tends to increase its aggressiveness of prefetching when applications are more memory-intensive, and vice versa.

5.5 Adaptation to Network Performance

In this experiment, we evaluate AMPoM's ability in adapting to network performances. We use Linux's `iptables` and the `tc` (traffic control) module [11] to simulate a broadband network with available bandwidth of 6Mb/s and latency of 2ms. The execution time of DGEMM (115MB) and RandomAccess (129MB), expressed as the percentage increase compared to the execution time in openMosix, are shown in Figure 9. The results of STREAM and FFT (whose spatial localities are similar to DGEMM and RandomAccess, respectively) and the other problem sizes are similar, so they are omitted for brevity.

In the DGEMM experiment, AMPoM achieves a modest increase in execution time (compared to that of openMosix) when the bandwidth is decreased from 100Mb/s to 6Mb/s (101% vs. 108% of openMosix's execution time). This shows that when the spatial locality is clearly developed, AMPoM is able to maintain a low overhead (8%) in remote paging even when the network performance drops significantly. By contrast, AMPoM's performance is more sensitive to the change in network speed when the access pattern is more random (as shown in the RandomAccess tests). This is because AMPoM prefetches much less aggressively, hence more page faults, when the access pattern tends to be random. In such cases, the network latency in remote paging would dominate the execution time. However, even in this unfavourable environment, AMPoM still outperforms NoPrefetch by about 4%.

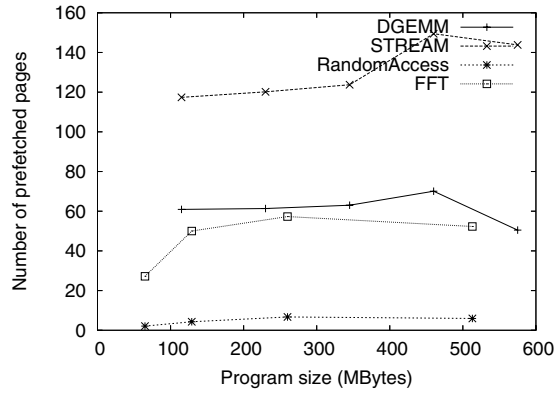

Figure 8. Prefetched pages per page fault

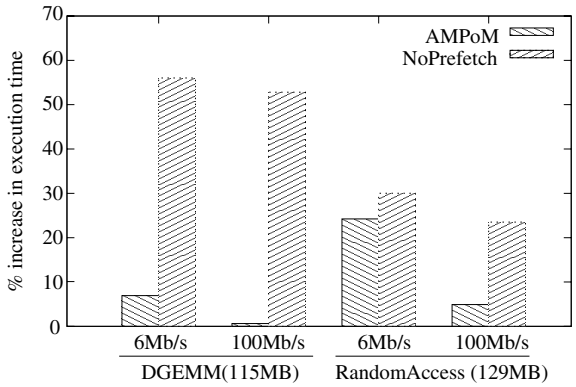

Figure 9. Adaptation to network performances

5.6 Migration of processes with small working sets

The above experiments are rather unfavourable to AMPoM because all HPCC programs access their entire address spaces, therefore both AMPoM and openMosix need to transfer the same amount of data in the process's lifetime. Indeed, AMPoM is designed to transfer data only when it is considered useful. Because of this, compared to openMosix which transfers the entire address space, AMPoM provides better support for those applications that have smaller working sets. In this experiment, we evaluate the performance of AMPoM under a more favourable case in which a process does not need to access all its address space after migration. We modified the source code of DGEMM so that it allocates 575MB of memory, but works on matrices of 115MB, 230MB, 345MB, 460MB, and 575MB large. The recorded execution time on openMosix and AMPoM are shown in Figure 10. As

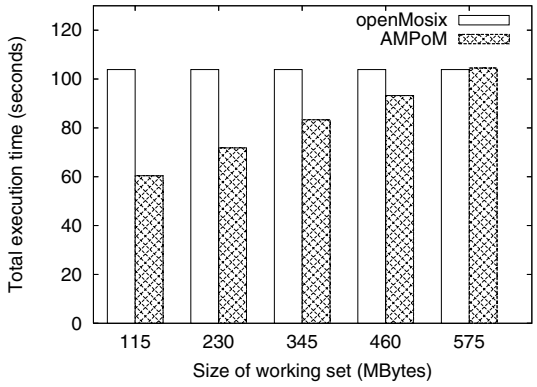

Figure 10. Process migration with smaller working sets

shown in the figure, DGEMM on AMPoM finishes faster because AMPoM fetches only the working set of the process, thus saving the communication time. This also re-confirms that AMPoM does not do excessive prefetching.

The results show that AMPoM performs especially well when the working set of a migrant is smaller than its address space. Indeed, many user applications exhibit such behavior. For example, interactive applications which need to wait for user's input are often large in size (e.g., those with graphical user interfaces), but might not require to perform all functions at one time. Another example is data-intensive applications whose migrations might be for data locality, i.e., they would allocate new pages after migration rather than using the existing ones. Besides, virtual machines that run as processes in a system also have similar characteristic.

5.7 Overheads of AMPoM

Figure 11 shows the time to determine the dependent zone, expressed as percentage of total execution time. As shown in the figure, AMPoM consumes less than 0.6% of execution time in finding the dependent zone in all test cases, while nearly all of them are less than 0.25% except one. The results show that the overhead of the AMPoM algorithm is small relative to the computation time of applications.

6 Related work

One common challenge facing most process migration supports is to migrate the virtual address space

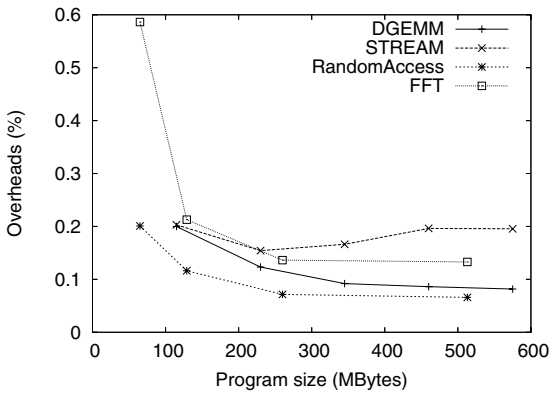

Figure 11. Overheads of AMPoM

which is most time consuming. A straightforward method is to copy the entire address space from the source to the destination nodes before the execution resumes (e.g., [17][21], etc.). Although easy to implement, this approach is wasteful since not all memory pages will be referenced again, while transferring all pages at one time also increases the freeze time of the migrating processes.

To shorten the freeze time, several research proposed to move only a portion of the address space during migration. For example, the address space of a process to be migrated in the V system [22] is pre-copied to the remote node prior to its migration, while the process is still executing in the source node. This approach, however, induces unnecessary network traffic if pages are modified after they are pre-copied. Accent [18] and OSF/1 AD 1 on Mach [16] use copy-on-reference, in which pages are moved to the destination node only when referenced. The advantage is that often a substantial proportion of pages are not subsequently referenced at all and so never need to be moved to the destination node. The disadvantage is the overhead for the process to re-establish the working set (through costly inter-node page faults) after it is migrated [21]. MOSIX [4] adopts a similar approach to migrate only the dirty pages from the original to the destination node, while the clean pages are fetched on demand. This approach, however, still suffers from long freeze time when a significant portion of the address space is dirty. Roush and Campbell [19] proposed to ship precisely the currently needed pages from each of the code, heap, and stack regions. While in this approach the freeze time is minimized, the execution efficiency is affected by the subsequent costly page faults. In contrast to these existing approaches, AMPoM intends to achieve a better trade-off between freeze time and execution efficiency of migrants.

Instead of migrating individual processes, several research has proposed migrating virtual machines. The main advantage is cleaner migrations since the migration supports can be implemented in the virtual machine monitors in which the system's and the processes' states are completely encapsulated. Prototype systems have been implemented on VMware [20][12], Xen [8], and Denali [24]. Indeed, this approach is more effective than process migration for migrating computing environments (e.g., desktop sessions) as the processes and the OS kernel can be migrated consistently. By contrast, process migration requires less amount of data transfer, which is more efficient for fine-grained, dynamic control of system loads in distributed systems. Besides, live migration of virtual machines also faces the dilemma of freeze time and execution efficiency; a better trade-off between these two factors can possibly be achieved by adopting a prefetching scheme similar to AMPoM. Specifically, AMPoM can be extended to consider memory access streams from multiple processes in a virtual machine in order to perform more effective prefetching.

Prefetching based on spatial locality [9] has a long history. It has been studied in various domains, ranging from uniprocessor and multiprocessor systems, to file systems, to databases. In general, these techniques involve identifying the timing of prefetch (e.g., pattern matching of sequential access), and how much data to prefetch (i.e, the data coverage). AMPoM's prefetching scheme is similar to them because it also relies on sequential access pattern to initiate prefetching. However, AMPoM determines the number of pages to prefetch based on an analysis of the spatial locality score (which is a variant of the score proposed in [23]), the paging rate of the application, and the processor and network utilizations, which is tailor-made for an environment with moving processes. In essence, AMPoM's strategy falls into the category of conservative prefetching [6], but with a unique technique to determine the data coverage.

7 Concluding remarks and future work

We presented the design, implementation and performance of AMPoM. The experimental results show that AMPoM can achieve a reasonably good balance between migration freeze time and execution efficiency of migrants. It should be noted that our experiments with a demanding benchmarks as HPCC tend more to reveal the *overhead* than the strengths of the proposed system. Indeed, when a migrant does not need to access its entire address space, AMPoM would outperform openMosix considerably due to the

reduced amount of data transfer. This property of AMPoM can benefit many user applications such as interactive or data-intensive applications, and virtual machines running as processes.

The results of this research open up new possibilities in the design of scheduling policies for distributed systems. For example, new scheduling policies can make use of AMPoM on openMosix to perform more aggressive migrations since the performance penalty of suboptimal decisions has been dramatically decreased. Furthermore, the concept of lightweight migration can be an integral part of a single-system-image service in the process management level in distributed systems.

Possible future work includes the design of scheduling policies that make use of AMPoM, and a tailored AMPoM for migrating virtual machines whose memory references are consisted of access streams from multiple processes. Besides, the current implementation of openMosix requires all system calls being redirected to the home node of the process, which significantly affects the performance of I/O-intensive applications. One can possibly employ some virtualization techniques (e.g., [15]) to remove this "home dependency", so that a migrant can perform system calls with little additional overhead.

References

[1] openMosix: An Open Source Linux Cluster Project. URL: http://openmosix.sourceforge.net.

[2] The DataGrid Project. URL: http://web.datagrid.cnr.it.

[3] The HKU Gideon 300 Cluster. URL: http://www.srg.cs.hku.hk/gideon/.

[4] A. Barak and A. Litman. MOS: A Multicomputer Distributed Operating System. *Software: Practice and Experience*, 15(8):725–737, 1985.

[5] R. B. Bunt and J. M. Murphy. The Measurement of Locality and the Behaviour of Programs. *The Computer Journal*, 27(3):238–253, 1984.

[6] P. Cao, E. W. Felten, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of ACM Joint International Conference on Measurement and Modelling of Computer Systems*, 1995.

[7] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with Transparent Migration Checkpointing. In *Proc. of 3rd Annual PVM Users' Group Meeting*, 1995.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proc. of 2nd Symposium on Networked Systems Design and Implementation*, pages 273–286, 2005.

[9] P. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, 33(1):64–84, 1980.

[10] M. Harchol-Balter and A. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15(3):253–258, 1997.

[11] B. Hubert. Linux Advanced Routing and Traffic Control HOWTO.

[12] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proc. of 4th IEEE Workshop on Mobile Computing Systems and Applications*, pages 40–46, 2002.

[13] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. *Lawrence Berkeley National Laboratory. Paper LBNL-57493*, April 2005.

[14] D. S. Milojicic, W. Zint, A. Dangel, and P. Giese. Task Migration on top of the Mach Microkernel. In *Proc. of 3rd USENIX Mach Symposium*, pages 273–290, 1993.

[15] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proc. of 5th Symposium on Operating Systems Design and Implementation*, pages 361–376, 2002.

[16] Y. Paindaveine and D. Milojicic. Process vs. Task Migration. In *Proc. of 29th Hawaii International Conference on System Sciences Volume 1: Software Technology and Architecture*, page 636, 1996.

[17] M. L. Powell and B. P. Miller. Process Migration in DEMOS/MP. In *Proc. of 9th ACM Symposium on Operating System Principles*, pages 110–119, 1983.

[18] R. Rashid and G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proc. of 8th SOSP*, pages 64–75, 1981.

[19] E. T. Roush and R. H. Campbell. Fast Dynamic Process Migration. In *Proc. of 16th IEEE International Conference on Distributed Computing Systems*, page 637, 1996.

[20] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proc. of 5th Symposium on Operating Systems Design and Implementation*, pages 377–390, 2002.

[21] C. Steketee, W. Zhu, and P. Moseley. Implementation of Process Migration in Amoeba. In *Proc. of 14th International Conference on Distributed Computer Systems*, pages 194–203, 1994.

[22] M. Theimer, K. Lantz, and D. Cheriton. Preemptable Remote Execution Facilities for the V System. In *Proc. of 10th ACM Symposium on Operating Systems Principles*, pages 2–12, 1985.

[23] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely. Quantifying Locality in the Memory Access Patterns of HPC Applications. In *Proc. of ACM/IEEE Supercomputing Conference*, pages 50–61, 2005.

[24] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing Services with Interposable Virtual Hardware. In *Proc. of 1st Symposim on Networked Systems Design and Implementation*, pages 169–182, 2004.