SDPS-2011
Printed in the United States of America, June, 2011

On the Applicability of Virtual Machine Migration for Proactive Failover

Peter Tröger, Andreas Polze
Hasso-Plattner-Institute
Universität Potsdam
14482 Potsdam, Germany
peter.troeger/andreas.polze@hpi.uni-potsdam.de

Felix Salfner
Department of Computer Science
Humboldt-Universität zu Berlin
10099 Berlin, Germany
salfner@informatik.hu-berlin.de

ABSTRACT: Proactive failover is a resiliency strategy that replaces classical reactive recovery by a combination of failure prediction and preventive failover. The approach can be applied to different hardware and software layers that exist in modern evolving systems – but each layer has its own costs and failover capabilities.

This paper investigates the applicability of virtual machine live migration for proactive failover, based on a generic strategy for choosing a proactive failover system level. The experimental performance analysis shows that virtual machine live migration has product-specific performance behavior with product-independent upper bounds on the total migration time. One major influencing factor is the dirty page generation rate inside of the virtual machine.

I. Introduction

Next generation processor and memory technologies will provide tremendously increased computing and memory capacities for application scaling. However, this comes at a price: Due to the growing number of transistors and shrinking structural sizes, overall system reliability of future server systems is about to suffer significantly. Analyses of large-scale systems have shown a *mean time between failures (MTBF)* in the order of 6.5 to 40 hours [1], depending on installation maturity. Google for example experiences a MTBF in the order of one hour, which is hidden from server software (and therefore the users) through a fault-tolerant middleware layer and replicating file systems [2]. This assumption especially holds for standard server hardware, e.g. X86 technologies, which is not designed from ground-up to deal with physical faults.

The traditional solution for this issue are different kinds of spatial redundancy in server operation. Various protocols cope with transient and permanent faults based on redundancy on or above the operating system level. Until in the early 2000s, these resiliency concepts mainly involved reactive recovery schemes for crash faults, based on well-known system properties like the expectable failure types and corresponding roll-back mechanisms. However, today's systems are complex, heterogeneous, evolving networked systems and infrastructures characterized by dynamic requirements and changes in the environment. This leads to increasingly fuzzy boundaries between fault classes. Next-generation resiliency strategies therefore need to be adaptable and wider in scope, without having a complete understanding of fault models and error propagation chains. This new way of achieving availability is widely known under the terms *autonomic computing* and *self-X* capabilities.

One prominent tool for achieving adaptive fault handling is *proactive fault tolerance*. While traditional solutions only *react* on failures that have already occurred, the use of proactive algorithms allows to *anticipate* failures and to act even before they occur. The concept recently gained more attention in standard server environments, were vendors provide according product features such as *Prefailure Detection and Analysis (PDA)* or S.M.A.R.T. monitoring for hard disks [3]. The majority of them is based on comparatively simple threshold approaches, were a set of corrected faults is expected to indicate an upcoming failure in the particular component (e.g. memory).

More elaborated techniques can infer complex system characteristics autonomously from different monitoring data using, e.g., machine learning techniques, and react accordingly [4], [5]. A classification of these techniques and an assessment of their effect on availability is available in [6].

One class of proactive fault tolerance techniques is concerned with *proactive failover (PFO)* to a supposedly fault-free unit before the system actually has failed. If the failover is successful downtime is avoided and hence availability is increased. The recent introduction of virtualization as another standard system layer in X86-based systems provides promising new possibilities for PFO. The virtualization layer allows for operating system - independent failover strategies based on mature virtual machine migration technologies. In the remainder of this paper, we first want to discuss the general tradeoffs that have to be considered in proactive failover architectures. Based on this general understanding, we discuss virtual machine migration as one relevant approach, based on experimental results with different virtual machine live migration products and application load scenarios.

II. The Coverage vs. Overhead Tradeoff

Todays computing environments usually comprise various system levels. A typical server environment consists of several multi-core CPUs accessing shared main memory on

a server. Several such blades are mounted to a computer rack, and each computational site hosts several of these racks. On top of that there are multiple computing sites located at various geographical locations for disaster survival. There might be additional layers (e.g., racks connected to a switch) or layers might be left out (e.g., no racks in the case of volunteer computing).

PFO can take place on various layers. In each case, a *migration object* is moved from one *failover unit* that is deemed to fail in the near future to an equivalent or at least similar unit that is supposed to be fault-free (see Figure 1).

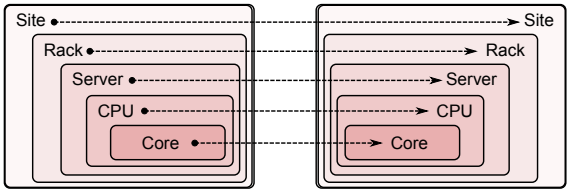

Fig. 1. Onion model for failover layers.

The appropriateness of each system level for PFO is determined by the amount and kind of available monitoring data, the kind of faults that can be anticipated based on this data, and the failover capabilities made available. Each level therefore has its own cost/benefit relationship.

Figure 1 is intentionally drawn in the form of "onion rings" similar to the fault model by [7] since each level of PFO represents a different *error containment barrier* [8] for hardware faults.

The result is that failover at a specific level cannot compensate for an error at any system level above. Faults on the lowest level include core-specific hardware structures such as L1/L2 caches. In this case, the migration of threads between cores (assuming appropriate hardware isolation) is a feasible remedy. If the problem is CPU-specific, e.g., a fault in the core interconnect, the threads must be migrated to a separate CPU in the system. Migrating processes from one CPU to another again does not help if there appears to be a problem with main memory. Higher layers might face power or networking issues and rely on whole system migration approaches for failover. This helps only if the target is outside of the error containment region, for example at another rack or site.

In conclusion, high-level failover solutions seem to offer resiliency for a larger set of fault classes. However, the caveat is that migration on higher levels involves an increasing overhead in terms of resources and time. Since the granularity/size of migration objects increases, the increased "distance" between source and destination unit results in in lower transfer throughput.

The broader coverage of possible faults in higher system layers also leads to heavier problem with the system state monitoring accuracy. The monitoring overhead and granularity increases with the system layers. The latter leads to an increasing inappropriateness of high-level monitoring data for detecting low-level system error states, since the data only reflects fault effects on the application behavior, visible in application server performance statistics and error logs.

We call the balancing act between the kind of faults to be handled and the costs to be invested the *coverage-overhead tradeoff*.

It could be argued that cross-level combinations can solve this problem (low level monitoring, high level failover). However, this works only for very well defined workloads, where the low-level monitoring can clearly identify derivations from the static load curve generated by the application.

Table I presents a list of failover levels, some representative faults on this level, and the according migration objects.

Coarse-grained migration at higher levels typically takes longer than on lower levels. One part of any migration approach is the choice of a migration target. This comes down to a classical resource scheduling and load balancing problem, which – depending on the migrant granularity – is either already solved on the system layer, or must be added to the solution.

III. Factors for Proactive Failover

The novel property of PFO is its anticipatory handling of error conditions. Although a simple time-based prediction based on previous occurrences of failures is also possible, most approaches rely on a continuous evaluation of system state for error conditions. It must be noted that the associated problem of selecting monitoring variables poses a challenge of its own. The process of collecting the monitoring data in time window Δ_d, detecting the imminent failures (prediction), and performing the preventive migration activity forms a classical control loop.

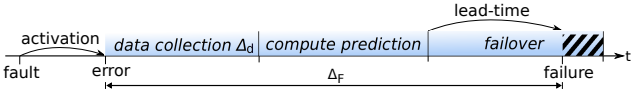

Fig. 2. Time line for proactive failover.

We assume that the total amount of time consumed for PFO is determined by the sum of time consumed for each of the three phases. It comes as no surprise that there is an upper limit for the total amount: the entire process must finish before the failure occurs. Figure 2 depicts the situation. The hatched area indicates that in this setting the total time consumed for PFO exceeds the time until failure occurrence.

Failure prediction can in general only reach a certain level of accuracy. Determining factors are, among others, the amount of data gathered (Δ_d), the computational complexity of the prediction algorithm, and the lead time, i.e., how far predictions reach into the future. The overhead comprises efforts such as data collection and computing the

TABLE I
COMPUTER SYSTEM LEVELS AND MIGRATION OBJECT EXAMPLES.

Level	Fault Location	Migration Object
Site	Site power supply, cooling, network backbone	Virtual machine over WAN
Rack	Rack power supply, switching hardware	Virtual machine over LAN
Server	Server hardware	Virtual machine over highspeed network
CPU	Shared CPU units (L3 cache)	Threads in shared memory (flushed cache)
Core	Per-Core CPU units (L1/L2 cache, registers)	Threads in shared memory (persisting L3 cache)

prediction models. Failure predictors are always imperfect, leading to false positive or false negative predictions. The goal of any prediction algorithm approach is of course to maximize the number of true positive and true negative predictions, and to reduce false predictions.

Even though a generic failure prediction approach in or below the (guest) operating system layer seems to be desirable, our earlier results indicate that - in order to give "long-time forecasts" ($> 10s$) - additional monitoring on the application layer will be crucial, which makes the predictor less generic. This holds specifically for X86-hardware and software products considered in the study presented here (see also [9]). An alternative can be hardware-centric solutions such as the zSeries by IBM, which relies on advanced event sources in all hardware units to perform *predictive failure analysis (PFA)*. Based on the zPDT mainframe simulation environment, we are currently investigating how PFA concepts can be ported to standard server technologies. Another solution is the utilization of advanced processor hardware data sources in recent X86 products such as the Nehalem EX processor.

The second major factor influencing the design of a proactive failover solution is overhead costs introduced by the additional functionality. This includes administrative costs, service interruption costs, and consumed interconnection throughput costs during operation and failover attempts. In order to make a proper analysis of these factors, two time intervals need to be distinguished: *Total migration time* expresses failover duration from start to end of the migration process whereas *blackout time* denotes the subphase in which the service provided by the migrated object is unavailable during the failover process. Both times are determined by the chosen level of migration and the type of objects to be migrated. The classical availability metric (uptime over lifetime) is hence influenced mainly by blackout time whereas total migration time determines overall feasibility of the PFO solution.

IV. HOW TO SELECT THE RIGHT SYSTEM LAYER

Our proposed approach to identifying a feasible proactive failover strategy consists of two major phases. The first phase eliminates inappropriate PFO solutions, while the second finds the best possible variant in the remaining layers. The first phase can be structured into the following steps:

1. *Choose a fault model F* to be considered. This should consider the nested nature of fault classes [7].
2. *The chosen fault type mandates a system layer* where such faults occur. The choice therefore rules out all lower system levels for migration, due to the error containment property discussed above.
3. *Define some percentage α of predicted occurrences of F that should be handled, and determine the time interval Δ_F between error and failure occurrence* (see Figure 2). We assume the corresponding random variable T_F to have a probability distribution and we need to identify Δ_F such that $P(T_F \leq \Delta_F) \geq \alpha$. This can be derived from historical data or through fault injection experiments.
4. *Use Δ_F as an upper bound for approving the feasability of a PFO strategy*. If total migration time at some level is larger than Δ_F under normal conditions, then this level sets an upper bound in the stack of system levels.
5. *Reduce the set of PFO candidate solutions* by defining (or measuring) lower bounds for the data collection interval (Δ_d) and for the time needed to compute the prediction algorithm are assumed.

Steps one and two are qualitative assessments, whereas steps three to five are quantitative ones.

In the second phase, an optimal PFO solution has to be selected from the restricted solution space. However, due to the manifold of PFO solutions (both in terms of the number of failure prediction algorithms as well as in terms of migration layers and objects), identifying an "optimal" solution is in general infeasible.

An optimal PFO approach obviously should maximize availability at minimal total overhead. Availability depends not only on the blackout time but is also linked to prediction accuracy as any false alarm causes extra (unnecessary) blackout time, and any missing alarm causes downtime due to not handling the fault. Additionally, lead time has to be greater than total migration time, and may be larger than the maximum total migration as sufficient condition.

Total overhead includes overhead in time and resources as well as administration for monitoring, prediction and migration. Given such complex optimization problem there unfortunately is no way to determine upfront a single best PFO approach for a given system and scenario. It rather requires a balanced assessment of load-dependent overhead

costs, design of experiment and assessment of prediction accuracy as well as technical feasibility checking.

V. An Example: Applicability of virtual machine migration

With the given analysis strategy for proactive failover approaches, it is not only possible to choose the right technology for a chosen fault model, but also to rank mechanisms in their suitability for on or the other fault type. As one example, we want to investigate virtual machine live migration as possible migration technology.

The first step of the PFO strategy selection is the choice of a fault model F. From the nature of virtual machine migration, it is clear that this mechanism is best suited for faults on or above system level, while outages on lower levels (e.g. core level) can be handled more efficient in intra-system approaches, such as operating system scheduling.

One typical representative for relevant system-level faults are multi-bit memory cell faults. Recent studies show that this kind of problem is on the rise, mostly reasoned by aging effects and the decreased structural sizes of such components [10]. Even though modern operating systems can deal with faulty memory pages, they can only apply limited recovery strategies. One example is the Linux memory management implementation, which (correctly) decides to kill those user-space processes that try to work with memory pages that became defective during runtime. Although this strategy maintains the integrity of the particular operating system instance and application data, it might be catastrophic from the viewpoint of service availability.

The IT industry already started to consider the anticipated future increase of memory problems with a set of new hardware monitoring facilities. Mechanisms such as the Intel *Machine Check Architecture (MCA)* or the *Predictive Failure Analysis (PFA)* offer the necessary monitoring support for hardware-corrected errors, but delegate the according preventive reaction to the higher software layers.

Permanent memory faults, if not handled accordingly, have the potential to harm the entire physical server. Cross-CPU and cross-core migration facilities do not provide the necessary error containment barriers in a classical SMP setup. In contrast to adding / removing faulty CPUs from a running virtual machine host, current hypervisors also have only limited support for memory hot plugging and removal. According to the system layer onion model shown in Figure 1, we can therefore only move up in the system stack to the point were the affected resource (here the main memory) is no longer shared. Migrating entire virtual machines across blades therefore seems to be the appropriate failover strategy for state-of-the-art in X86 virtualization software and server hardware. .

According to the presented methodology, the time interval Δ_F between error and failure occurrence must be estimated based on experience or fault injection experiments. Since the majority of memory reliability studies relates to the FIT metric (failures per billion device hours), there is only a small amount of existing data on Δ_F for memory errors. The well-known large scale study by Schroeder et al [11] at least shows that there is a strong correlation between hardware-correctable and not correctable memory faults. For nearly 80% of the uncorrectable memory faults in the system, there were correctable memory faults identifiable in the same RAM module. Even though the study does not provide insights on the average time distance, it implies correctable and non-correctable errors to both happen 'during a month'. Sources from semiconductor industries estimate error-per-bit-hour rates ranging from 4E-7 down to 5E-14, underlining the assumption that the typical Δ_F can be expected to have at least the magnitude of hours. The challenge here is to define the threshold of correctable faults that serves as activation of an error state (see Figure 2).

Other examples for relevant system-level outages with a large enough prevention time frame are physical host storage outages or overload situations, which can be treated as performance fault. Based on the given time dimension, we now want to investigate the migration time frame given by virtual machine live migration facilities.

VI. Virtual Machine Live Migration

Virtualization as a concept for isolation and multiprogramming is known since the late 60s [12] [13]. Todays data center operation more and more facilitates virtual machine *live migration* as standard solution for load balancing or physical machine maintenance. In such a setup, a *virtual machine (VM)* can be migrated during runtime from one physical machine to another physical machine without explicit interruption.

This is technically realized through a combination of shared storage for the VM disk images and a pre-copy phase, where differential updates amongst the VM working sets are transferred until a product-specific stop condition is fulfilled. In the subsequent phase, equivalent to the *blackout phase* denoted above, the VM on the source host is stopped and remaining memory pages are incrementally copied over.

The movement can take place not only because of load balancing or hardware maintenance demands, but also as a reaction on hardware error conditions detected by the environment. In most cases, this implies a rollback to and recovery from the last fault-free state of the whole virtual machine. When the suspend / resume cycle is triggered after the actual error state started to be detectable, the recovery scheme implements a classical *reactive migration*. Unfortunately, reactive migration often cannot be applied to services provided with soft real-time constrains, as the downtime involved might violate given service-level agreements such as maximum service latency or maximum number of dropped requests per client. The alternative *pro-active* recovery scheme proposed by us would move the virtual machine away to another physical host *before* a failure occurs

[9].

The migration time here is the time from requesting the hypervisor to migrate the VM until the tool chain reports the successful migration. The blackout time is the period where the VM is not responsive to network I/O due to the migration. The latter needs to be significantly smaller than the migration time in order to fulfill the live property of migration. In order to obtain a quantitative statement for the worst-case live migration time, we conducted a large set of experiments with both artificial and benchmark-based load generators. Total migration time was measured by capturing the runtime of the product command-line tool that triggers a migration. Downtime was measured by a high-speed ping (50 ms) from another host, since the virtualization products do not expose this performance metric by themselves. The downtime is expressed as the number of lost Ping messages multiplied by the ping interval. All tests were executed on hardware provided by the HPI Future-SOC lab – 8-Core Intel Xeon E5540 at 2.53GHz, 12GB RAM per machine, 2 Gigabit-NICs for the migration link and the external link. All measurements were performed both with Windows Server 2008 and Linux 2.6 installations in the guest. All machines were configured with one virtual CPU and a varying amount of (virtualized) physical RAM. In all cases, the virtualization guest tools / drivers were installed. Our study covered the hypervisors from VMWare ESX 4, Xen 3, and KVM.

Live migration, as every performance-critical software feature, is influenced by a multitude of hardware / software factors. We are aware of the fact that new product versions, node and networking hardware as well as special optimization switches can lead to better or worse results. Nevertheless, the point of our investigations is to identify major worst-case influence factors when using live migration for dependability purposes.

In the first experiments, we performed at least 10 migrations per CPU utilization degree in the guest system, with 2 GB of configured memory and a neglectible memory load. The results (see Figure 3) show a 95% confidence interval of not more than -/+ 1s derivation in the migration time for all load values. This proves the migration to be independent from CPU load factors, since the hypervisor reserves enough computational capacity for its own purposes.

Figure 4 shows the effect of different virtual machine RAM configurations on migration time. While VMWare and Xen show the expected independence of migration time and virtual machine RAM size, KVM has a linear dependency in our configuration. This implies that KVM performs a full memory movement in the pre-copy phase of migration, regardless of the actual page status. Figure 5 shows the according effect on migration performance under increasing memory utilization conditions. From a particular break-even point, the different migration strategy of KVM pays of in terms of migration time. This underlines the fact that different configurations or strategies for live migration cannot can only be rated under predefined load conditions from the application side. The experiments also give a good indication on the upper bounds for migration time under worst-case load conditions.

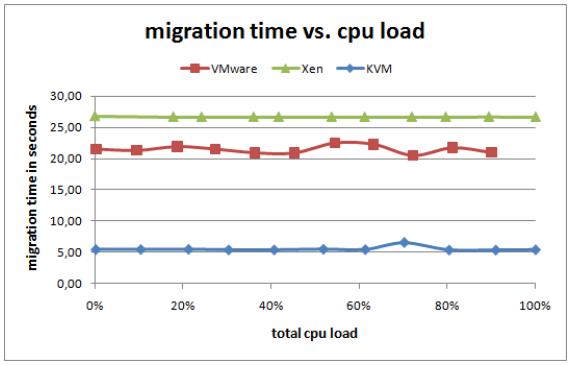

Fig. 3. Migration time for increasing guest CPU load

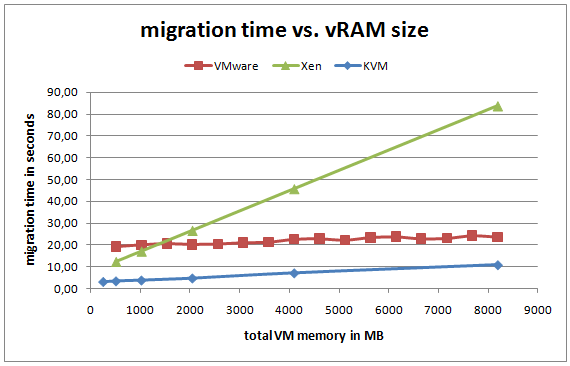

Fig. 4. Migration time for increasing virtual machine sizes

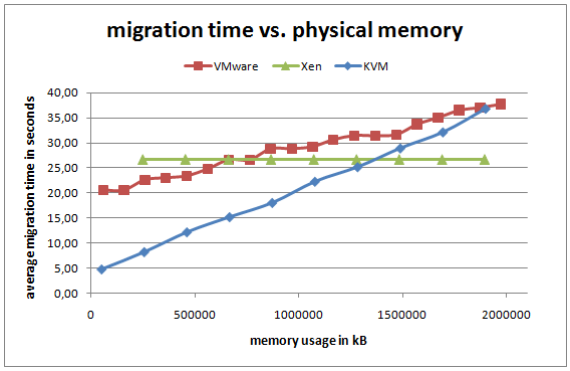

Fig. 5. Migration time for increasing memory usage

Other experiments investigated the influence of dirty page generation on migration time. We used a small dirty page generation tool, which modifies a locked set of memory pages in a round-robin fashion during the live migration. The experiments are based on the hypothesis that the dirty page rate forms a relevant independent variable for

migration time. This assumption is reasoned by the pre-copy approach applied in standard live migration facilities, were pages modified after migration start must be transmitted again, or must be deferred to the blackout phase. Products like Xen identify the write attempts by emptying the shadow page table entries at the beginning of each round [14].

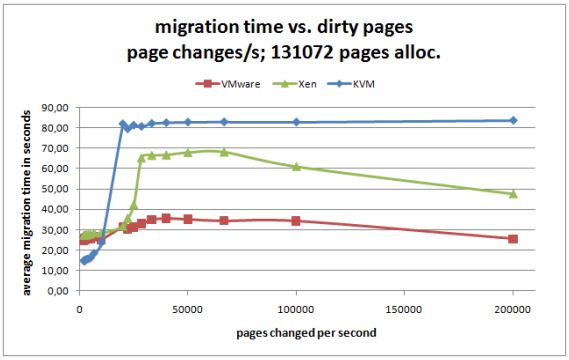

Fig. 6. Migration time under page modification load

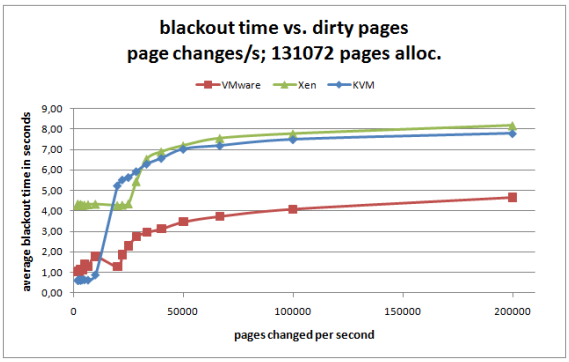

Fig. 7. Blackout time under page modification load

Figure 6 shows that for an increasing page change rate, all products have a fast ramp-up in the relative migration time. At a product-specific rate of dirty page generationboth VMWare and Xen switch the memory transfer policy so that the overall migration time decreases again. An investigation of source code and personal communication with vendors confirmed that this is reasoned by a shift of the affected memory regions to the blackout phase of the transfer. This can also be acknowledged by the according increase in blackout time in these cases, as shown in Figure 7. KVM remains constant in its migration time when the threshold value is crossed.

In order to obtain a deeper understanding of the related influence factors in dirty page load, we performed experiments with different parameter combinations for the amount of modified memory and the page modification rate per product. Figure 8, 9 and 10 show contour plots for the different combinations and the resulting migration time.

All frameworks have their own similar contour shapes for increasing virtual machine memory size, however, both Xen and KVM show linearly increasing migration times in this case. This seems to proof that VMWare purely focuses on the transfer of modified pages, since the configured virtual machine size has no relevant influence at all. There is only a small influence of the amount of modified memory to the migration time, which could be simply reasoned by dirty page testing overhead.

The other two products seem to 'give up' from a certain modification rate, by starting larger memory block transfers without deeper consideration of its modification state. The contour plots show that Xen, from the modification rate threshold point, at least still shows a dependency to the amount of memory allocated by the load generator, even though the increase rate is much higher than with VMWare. The migration time in KVM from the threshold point starts to depend on the virtual machine size only. Additional 'zoom-in' experiments showed in a 95% confidence interval that this rapid increase in the migration time for Xen and KVM is a consistent property of the products.

The shift in the behavior of Xen and KVM can be explained by the documented stop conditions for the pre-copy phase in these products. For normal operation of live migration facilities, it is assumed that only a few numbers of pre-copy rounds are necessary. As an example, the pre-copy phase of Xen stops (1) when a small-enough amount of memory is left on the source or (2) if an upper limit for the transferred data was reached or (3) if the time taken gets too long [15]. The latter factor is measured in number of pre-copy rounds. If the modification rate reaches a threshold close to the link speed, the pre-copy phase is only stopped by the second condition, so that the amount of transferred data becomes a multiple with the VM size. VMWare seems to have a smoother handling of the different load factors in a live migration setup here.

In sum, it turns out that all virtualization stacks provide a suitable amount of deterministic behavior and worst-case migration performance. Confidence intervals for the measurements taken never dropped below the 90 % threshold. Live migration therefore seems to be ready for prime time in reliable large-scale server operation. This is underlined by the fact that blackout time – the crucial factor for not sacrificing service performance agreements – also stays in acceptable boundaries. Under extreme circumstances for virtual machine size and memory usage patterns, migration time can go up to 300 seconds, while the average cases with light load vary between 10 and 30 seconds. This fits into the time frame given from the fault model examples, which makes virtual machine migration a more than feasible solution for proactive failover. A side aspect not targeted here are legal issues coming into the game when virtualization products perform automated migration decisions. Currently, the virtualization products vendors avoid such functionalities as default setting, in order not become liable for

broken service level agreements arising from malfunctioning migration features.

With the given understanding of virtual machine live migration as failover approach, the neccessary next step is the choice of a matching prediction strategy. In future work, we intend to realize a meta-prediction approach based on multiple predictors on different system layers [9]. This honors the situation that host operating system, hypervisor, and guest operating system form a multi-level system stack with different relevant information sources. A global *health indicator* is integrated in the virtualization cluster management, combining the outputs of all predictors into one coherent evaluation of the current system state.

VII. Summary

We investigated the applicability of virtual machine live migration for proactive failover in a two-step process. First, we introduced a generalized methodology for ranking proactive failover approaches, based on the assumed fault model and the failover system level. In the second step, we analyzed some results from a large-scale live migration experiment to prove the suitability for avoiding failures due to non-correctable memory hardware errors. The three investigated virtualization products show their worst-case migration time under specific dirty page load conditions. The worst-case point heavily depends on a combination of memory load factors and product types, which shows that both application-specific and virtualization product-specific stress tests are necessary for a true availability improvement through virtual machine live migration.

VIII. Acknowledgments

We would like to thank Matthias Richly from Hasso Plattner Institute for the realization and management of the experimentation environment. All hardware resources were made available by the HPI FutureSOC lab infrastructure.

References

[1] C. hsing Hsu and W. chun Feng, "A Power-Aware Run-Time System for High-Performance Computing," in *2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 1.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 29–43.

[3] E. Pinheiro, W.-D. Weber, and L. Barroso, "Failure trends in a large disk drive population," in *5th USENIX conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2007.

[4] S. Chakravorty, C. Mendes, and L. Kal, "Proactive fault tolerance in large systems," in *1st Workshop on High Performance Computing Reliability Issues, in conjunction with the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[5] S. Fu, "Failure-aware resource management for high-availability computing clusters with distributed virtual machines," *Journal of Parallel and Distributed Computing*, vol. 70, pp. 384–393, 2010.

[6] F. Salfner, M. Malek, A. Casimiro, R. Lemos, and C. Gacek, *Architecting Dependable Systems with Proactive Fault Management*. Berlin, Heidelberg: Springer Verlag, 2010, vol. 6420, pp. 171–200.

[7] F. Cristian, B. Dancey, and J. Dehn, "Fault-tolerance in the Advanced Automation System," in *20th International Symposium on Fault-Tolerant Computing (FTCS-20)*. IEEE, Jun. 1990, pp. 6–17.

[8] R. Hanmer, *Patterns for Fault Tolerant Software*, 1st ed. John Wiley & Sons, Oct. 2007.

[9] A. Polze, P. Tröger, and F. Salfner, "Timely Virtual Machine Migration for Pro-Active Fault Tolerance," in *2nd International Workshop on Object/component/service-oriented Real-time Networked Ultra-dependable Systems (WORNUS), at 14th International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC) (to appear)*. IEEE, 2011.

[10] X. Li, M. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2010.

[11] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2009, pp. 193–204.

[12] R. Adair, R. Bayles, L. Comeau, and R. Creasy, "A Virtual Machine System for the 360/40," IBM Cambridge Scientific Center Report, Tech. Rep. G320-2007, 1966.

[13] R. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, vol. 7, pp. 34–45, Jun. 1974.

[14] J. Zhu, Z. Jiang, Z. Xiao, and X. Li, "Optimizing the Performance of Virtual Machine Synchronization for Fault Tolerance," *IEEE Transactions on Computers*, vol. 99, 2010.

[15] S. Akoush, R. Sohan, A. Rice, A. Moore, and A. Hopper, "Predicting the Performance of Virtual Machine Migration," *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, vol. 0, pp. 37–46, 2010.

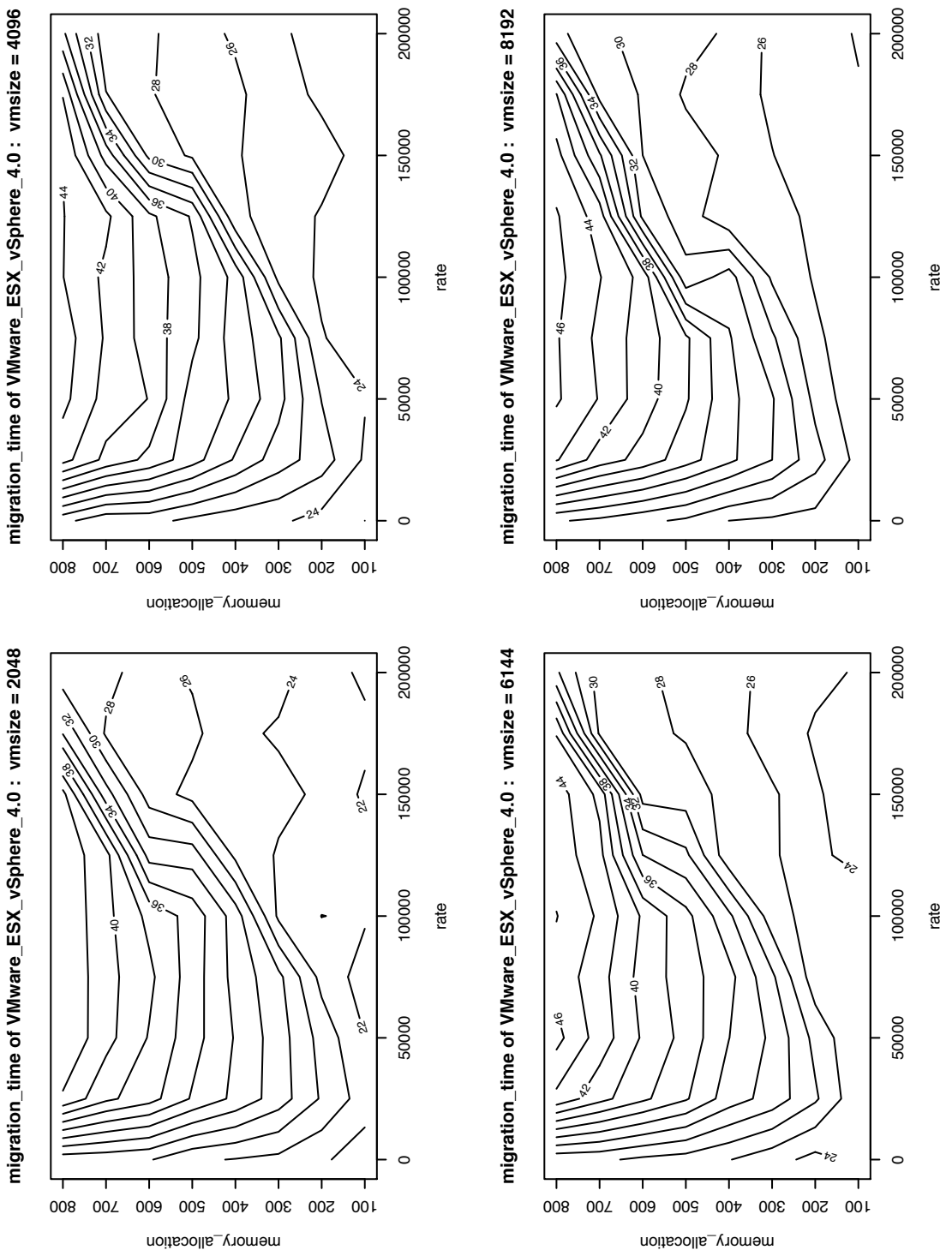

Fig. 8. VMWare migration time with varying dirty page load.

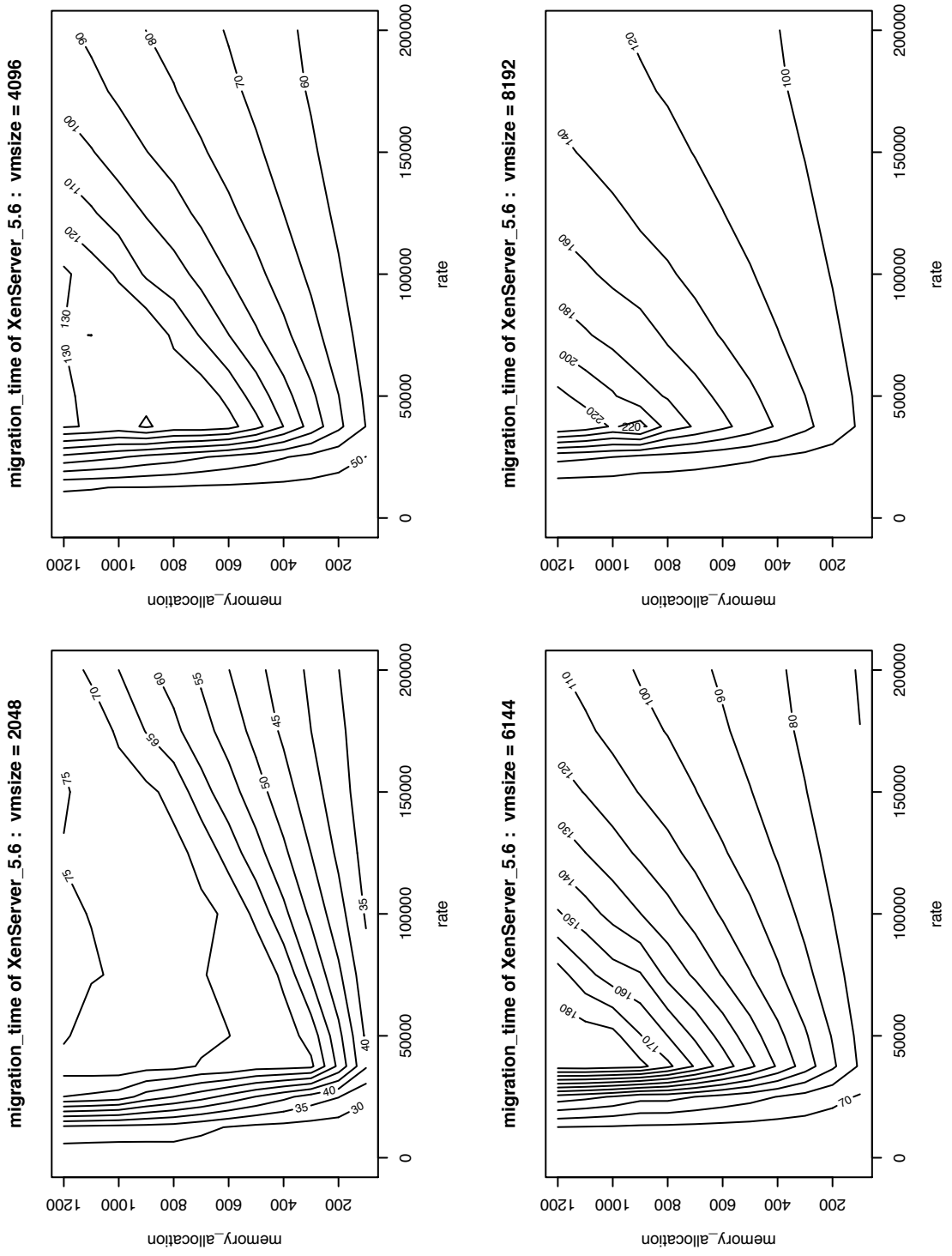

Fig. 9. Xen migration time with varying dirty page load.

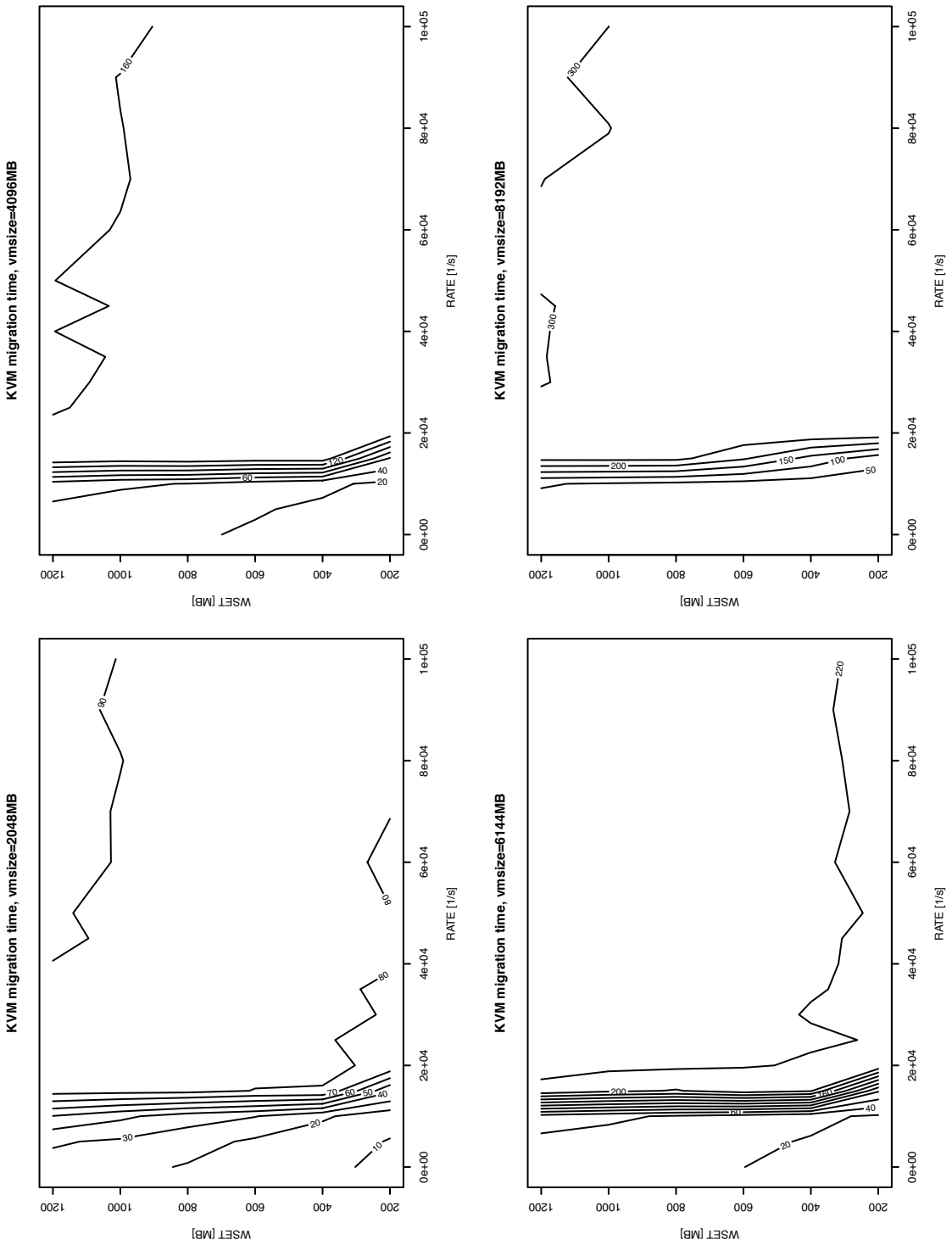

Fig. 10. KVM ProxMox migration time with varying dirty page load.