Motivating Elastic Operating Systems in the Cloud

Amit Gupta
University of Colorado at Boulder, amit.gupta@colorado.edu

Follow this and additional works at: https://scholar.colorado.edu/csci_gradetds
Part of the Computer Sciences Commons

Recommended Citation
https://scholar.colorado.edu/csci_gradetds/64

This Thesis is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.
Motivating Elastic Operating Systems in the Cloud

by

Amit Gupta

Bachelor of Engineering, Electrical and Electronics Engineering, 2003,

VTU, Belgaum, India

Advanced Diploma, Embedded System Design, 2005,

Center for Development of Advanced Computing, Pune, India

M.S, Computer Science, 2013,

University of Colorado at Boulder

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2013
This thesis entitled:
Motivating Elastic Operating Systems in the Cloud
written by Amit Gupta
has been approved for the Department of Computer Science

Dr. Richard Han

Dr. Eric Keller

Dr. Shivakant Mishra

Date __________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
With almost ubiquitous reliance of our society on the cloud and datacenter today, the industry is continuously striving to push the envelope in making on-line services more autonomously scalable i.e., “Elastic”\(^1\). Attempts to achieve this goal have been fragmented by being architected on a per-application basis.

Elastic, scalable frameworks like Map-Reduce provide a good instance of such fragmentation. There are huge classes of problems that do not fit into this framework, and therefore derive none of the benefits of elastic scalability! We thus see a lot of time and effort spent in re-engineering this \textit{elasticity wheel}. The massively parallel hardware architectures imminent upon us will have to satisfy the stringent performance demands of tomorrow’s datacenter applications. Further more, the evolution of traditional operating systems (OS) has been incrementally layered on top of design assumptions that are outdated today. This has led them to becoming highly optimized, complex projects that are unable to readily incorporate the emerging requirement of elasticity. We therefore contend that elasticity should be a \textbf{first order design concern of operating systems}.

In this thesis, we primarily attempt to motivate the requirements and opportunities to make OS-es better suited for future datacenters by building elasticity into them as a system service available to generic applications. We question the design of a key OS abstraction, the \textit{Process}, more specifically the \textit{Page Table}. We reason through a possible change to this data structure along with its implications on cloud based applications. We present some encouraging results of our preliminary experiments and conclude by highlighting future research paths we’d like to pursue and their associated challenges. This work was part of a collaborative research effort which appeared under the title “Towards Elastic Operating Systems” in the \textbf{USENIX XIV Workshop on Hot Topics in Operating Systems (HotOS), May 13-15, 2013.}

\(^1\) i.e., Being able to expand and contract across several physical resources in a dynamic manner.
Dedication

I would like to dedicate this work to my incredibly supportive parents Dr. Rani Gupta & Dr. Vijay Prakash Gupta for their unconditional love and support throughout my life and in this endeavor. My elder brothers Piyush Gupta, Dr. Prashant Gupta and their wonderful wives Srividya Subramanyam and Dr. Himani Agrawal have all been the strongest pillars of support. Piyush Bhaiya, I could not have done it without your guidance and support. Thank You!

I’m truly blessed to have these amazing people in my life.

They are indeed the giants whose shoulders I get to stand on.
Acknowledgements

I owe a huge debt of gratitude to my advisor Dr. Richard Han for convincing me to embark on a project as deep and as wide in scope as this. His vision for this project was a beautiful mix of inspiring and daunting. His periodic words of encouragement never let me lose belief in my abilities and always made me try harder.

I would like to thank Dr. Eric Keller, whose insights and clarity of thought truly catalyzed any of my efforts. He made me aware of my blind spots on several occasions and our project really did start taking flight when he came aboard.

I feel very fortunate that to have taken my first foray into research with both of them at the helm mentoring me. My desire to pursue a career in research has only been strengthened by the valuable lessons learned under their guidance. I would also like to thank Dr. Shivakant Mishra for serving on my committee and for his valuable feedback.

I would like to thank the Department of Applied Mathematics and Dr. Thomas Hauser, Director of Research Computing for generously supporting my education for the last 2 years. Last but not the least I would like to thank the University of Colorado at Boulder for being my home away from home these last 3 years. It has been an amazing experience to have received an education here. I leave to begin the rest of my career with nothing but fond memories of these beautiful surroundings and the rewarding time spent here.
## Contents

### Chapter

1. **Introduction** 1
2. **Related Work** 5
   - 2.1 FOS & Barrelfish 5
     - 2.1.1 Drawbacks of brand new OS-es 8
   - 2.2 Distributed Shared Memory 9
   - 2.3 Single System Image Operating Systems & vNUMA 10
3. **Stretching a Process** 11
   - 3.1 ElasticOS with Elastic Page Tables 11
   - 3.2 Our Model contrasted with prior work 13
4. **Our Experiments and Results** 19
   - 4.1 Locality Patterns and Run-time detection 19
     - 4.1.1 Looking inside an application 19
     - 4.1.2 Simulation of Application Execution on ElasticOS 20
     - 4.1.3 Application Partitioning: Our initial attempt with Static Algorithms 23
     - 4.1.4 Multinode Adaptive LRU algorithm 24
   - 4.2 Candidate Applications and Workloads 24
   - 4.3 Simulation Results 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>Initializing SQL tables before the experiment</td>
<td>53</td>
</tr>
<tr>
<td>C.2</td>
<td>SQL Benchmark Driver</td>
<td>59</td>
</tr>
<tr>
<td>C.3</td>
<td>Page Trace Tool in INTEL PIN Framework</td>
<td>62</td>
</tr>
</tbody>
</table>
Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Comparison Estimates of Various Systems for <strong>MySQL</strong></td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison Estimates of Various Systems for <strong>apache2</strong></td>
<td>33</td>
</tr>
</tbody>
</table>
Figures

Figure

1.1 An Elastic OS in a data center .................................................. 4

2.1 FOS Architecture Overview .................................................... 6

2.2 Barrelfish ................................................................. 7

2.3 Accesses in Traditional DSM Systems ................................. 9

3.1 Elastic Page Tables .......................................................... 12

3.2 Accesses in ElasticOS ......................................................... 13

3.3 A “stretched” Process consisting of (a data page + a thread) on each node. It illustrates elasticity in multithreaded computation and memory where threads operate on independent data. If the threads happen to be I/O intensive, a similar layout would illustrate elasticity of I/O ........................................ 16

3.4 Even a single thread of execution can also be stretched. We see here that a single threaded Process, can stretch its memory by acquiring pages on both nodes and jumping execution between them as required. This illustrates elasticity in memory for a single thread and hints at how locality in access patterns can be used for page placement on nodes. ................................................................. 17

3.5 Here we see multiple stretched Processes running in an ElasticOS. Threads accessing the same data pages co-locate on one node forming a natural grouping. When on the same node, the conventional techniques for concurrent access may be leveraged. 18
4.1 The table above shows an example trace from instrumenting an application with our PIN tool. Below it we have an our graph representation (derived from this trace) the data pages accessed.

4.2 Shown here is a spring layout graph representation of the data page accesses trace obtained by instrumenting an idle MySQL daemon. The daemon was instrumented for 60 seconds including its start-up/initialization activity. Seen here (colored) are 2 “clusters” of locality i.e., where members of one group are often accessed together.

4.3 Multinode adaptive LRU algorithm

4.4 Simulation of the multinode adaptive LRU algorithm for a 334 second MySQL execution trace. There were 3287 data pages in this test. The optimal threshold was at 20% for which the number of jumps + pulls was 9 per second.

4.5 Simulation of the multinode adaptive LRU algorithm for a 393 second Apache2 execution trace. There were 606 data pages in this test. The optimal threshold was at 40% for which the number of jumps + pulls was 457 per second.

4.6 Execution Context Switch experiment to measure state transfer overhead

A.1 ElasticOS Architecture.
Chapter 1

Introduction

There are two dominant trends visible in the landscape of computing in the last decade. The first is the emergence of the cloud as a platform for computation, software & infrastructure services. The second is the increasing number of cores on a single die in processing chips [42]. In view of slower improvements in serialized computation speeds [43], individual cores will get faster, but not at the rate we have enjoyed over the last decade. Instead we’ll begin seeing a greater number of cores on the same chip, quickly leading to massively parallel machines. This trend is already visible upon surveying off the shelf commodity computers today.

The ease of the Symmetric Multi Processor (SMP), Shared Memory programming model for the generic programmer when compared to the complexity of parallel programming models favors the former to remain in use by developers. These are therefore clear indicators of the emerging gap between the applications and the infrastructure that runs them. The layer that sits between these two, the Operating System (OS) was also designed for a small number of cores and tends to have inherent inefficiencies when the number of cores is drastically increased.

Therefore we are clearly at a juncture where the design of operating systems needs to be rethought. There will have to be more innovations in operating systems to provide the same quantitative leaps in application performance. The need of the future will be an operating system for the datacenter comprised of massively parallel hardware.

A key property of cloud-based based systems is elasticity, namely the ability to dynamically provision resources to applications on demand in order to scale up/down to accommodate changing
requirements for CPU, memory, storage, and network bandwidth. Realizing elasticity in cloud applica-
tions today is often a cumbersome process, requiring applications to integrate with services like
elastic load balancers and/or be rewritten to accommodate distributed frameworks like map/reduce
or cluster-based operating systems.

In particular, there are four general steps that a cloud based application must go through to
achieve elasticity. First, the application must be partitioned into independent units as the current
granularity of adaptation is either on the order of an entire virtual machine or a process that does
not overload a single machine. Second, the developer must use a monitoring system and write
extra logic to implement their own heuristic to trigger expansion or contraction of number of VMs.
Third, the developer must implement or configure an existing system to distribute the workload
among the partitioned application instances. Fourth, the developer must incorporate mechanisms
to overcome partitions that have some shared state.

To put these steps in context, consider three examples: (i) a web server, (ii) a database, and
(iii) a computational job. To realize an elastic web server, one must instantiate a load balancer
(e.g., HAProxy [14]) in a separate VM, configure that load balancer for how to direct the incoming
requests, monitor request rates and utilize the cloud API to add/remove VMs which run the
web server software, and utilize an object store, rather than an in-memory data structure, to
deal with dynamically generated content. To realize an elastic database, we can look to MySQL
which has implemented a custom clustering technique [30]. In particular, query request handlers
(SQL nodes) are separated from data nodes and require the developer to configure the cluster
configuration. Extra mechanisms internal to the MySQL cluster extension ensure state is consistent
across the replicated nodes. For elasticity, a developer needs to couple this with a scaling service
and write custom logic to integrate with the MySQL cluster configuration. Finally, to realize
elasticity in computational processing, the developer must write the application with distinct units
of computation that can fit within a given machine and use a framework such as HTCondor [21]
to run remote processes on available servers, XtreemOS (with Kerrighed) [18, 29, 28] to spawn
processes locally and then migrate the process to different machines, or Map-Reduce to run a given
computation on a subset of data.

We can see that the current limitations imposed by the cloud today could lead to missed opportunities for elasticity. By limiting elasticity to fixed application-specific units based on what can run on a single machine, we crucially miss the opportunity to elastically scale to take advantage of the collective distributed resources provided by many cloud machines. Recent approaches in improving SMP scalability [40] and elasticity [35] tend to lean in favor of specific target application types. Instead of requiring a great deal of additional application specific services, frameworks, or targeting a specific class of applications, we should provide to all applications a generic OS service that allows them to automatically elasticize processing, memory and I/O, without code modification or additional frameworks/scripting.

In this thesis, we therefore introduce the concept of ElasticOS, which enables a process (or even a single thread) to stretch its associated resource boundaries across multiple machines automatically, expanding and contracting (as seen in Figure 1.1) on demand without requiring the application to be re-designed or configured with a complex combination of additional tools and frameworks. Our initial implementation within Linux 3.2 and a study of a MySQL execution trace provide hope that the ElasticOS vision is achievable.
Figure 1.1: This diagram shows a hypothetical Elastic OS in a datacenter. Ideally, it should be able to expand and contract to consume/surrender physical resources, as demanded by the workload on it but with its applications none the wiser.
Chapter 2

Related Work

To help reinforce the context that Operating Systems need to be redesigned for tomorrow's datacenters, let's examine some currently active, noteworthy peer efforts in this space.

2.1 FOS & Barrelish

The first project is Factored Operating System (fOS) [42]. It has many of the same motivations as we do (i.e. hardware becoming increasingly parallel and most platforms moving to a cloud-like infrastructure). Their key components as illustrated in Figure 2.1 are:

- Microkernels
- Message passing
- Fleets

There is one microkernel running on each core. Collectively, they form a network of cooperating microkernels, which form a substrate over which the rest of the system is built. The primary function/feature is to export a message passing mechanism for (both kernel-to-kernel, and userspace) communication. The message passing layer functions as the basis for all forms of IPC on the system. This interface is agnostic to whether the communicating entities are on the same node or across several nodes. An added advantage of the message passing mechanism is that it can be built on top of hardware that supports message passing or shared memory. The interface remains the same across fundamentally different architectures.
Figure 2.1: This diagram (taken from [42]) shows the overview of FOS design. Its comprised of a fleet of operating systems services sprinkled selectively over some of the cores, and communicating using message passing.
Figure 2.2: This diagram (taken from [6]) shows the overview of the multikernel Barrelfish design. It’s comprised of a group of “share nothing” microkernels spread over many (possibly heterogeneous) cores.
The last component organizes each operating system service as a group of daemons (or “a fleet”) spread evenly between the cores available. Each daemon runs on a dedicated core, and the nearest relevant daemon is reached by an application to access/request a specific operating system service.

The second significant project worth examining is Barrelfish [6]. Its design comprises of a network of microkernels each running on a core like a “cpu-driver” (as seen in Figure 2.2). The kernels are designed to not share any state with each other in-order to be decoupled and fast. Additionally this design incorporates heterogenous cores. It too, relies on message passing between these kernels and makes communications explicit instead of worrying about global cache coherency.

### 2.1.1 Drawbacks of brand new OS-es

Both these projects are building upon a clean slate as far as OS design in concerned. If we were to start from scratch today, these would both be promising designs indeed. Their primary drawback is however that they require a change in the programming model for the application writer. To be fair, both these projects are not meant to be immediate replacements for operating systems that power future datacenters but are rather meant to showcase interesting ideas for the new emerging architectures and spark interesting questions in the OS community.

The reality however is that very large expensive cloud application stacks exist in deployment today. They are powered by conventional operating systems, often Linux. They also host many large, complex, sometimes legacy applications that were written and designed for these OS-es. We therefore contend that the industry is not going to abruptly replace these current expensive deployments with (re)written applications on a brand new operating system.

We think the pragmatic approach to solve this problem is to investigate how the current OS design can be modified to be able to handle the requirement of elasticity, without requiring a change in the programming model. Such a hypothetical OS, if realizable, will be the need of the hour for datacenters.
2.2 Distributed Shared Memory

The late 80’s and 90’s saw a surge of research efforts [7, 8, 19, 34, 37, 41] in the area of Distributed Shared Memory (DSM). The goal in these systems was to export a single “unified” view of virtual memory to be used for IPC by parallel applications running across multiple nodes. Some restricted the shared view of memory across nodes to certain address ranges designated as “shared regions”, while others extended the idea to the entire address space. Upon examining their designs, we can see that a key feature used in them was replication of data objects (of varying granularity depending on the implementation e.g., pages for page based DSM systems, data objects for object based DSM systems, blocks for fixed size block based systems). The idea behind replication and caching was to make most accesses “local” and thereby not have to pay the cost of going over the network for each access. However, to keep these replicas current and maintain correctness of the system, consistency protocols were employed which came with the expensive cost of coherency messages exchanged over the network (see Figure 2.3). Each successive implementation experimented with the strictness of the consistency protocols to make gains in scalability. Eventually however research efforts in this area waned as no universally acceptable solution was found to the scalability limitations.

2.3 Single System Image Operating Systems & vNUMA

Chronologically the next (i.e., in the 2000’s) research efforts in this direction took the form of Single System Image (SSI) operating systems like OpenMosix [20], OpenSSI [33] and more recently Kerrighed [29] and XtreemOS [28]. They aimed at providing to applications a “unified” abstraction of one machine which was in-fact comprised of the collective resources of several physical machines. Therefore, when examining their memory subsystems we see that they subsumed the idea of DSM [22, 23, 25]. They also employed full process migration to spread CPU and memory load across nodes of a cluster. There is more recent similar work, namely vNUMA [9], that pushes this idea down to the hypervisor level attempting to give the OS the same abstraction of having one
Figure 2.3: Accesses in Traditional DSM Systems
large machine. However they have all faced similar scalability problems due to sharing a common design feature with DSM. We’ll further elaborate on this when contrasting it with our approach in the next chapter (Chapter 3).
Chapter 3

Stretching a Process

3.1 ElasticOS with Elastic Page Tables

To achieve elasticity, we propose to implement an ability to stretch processes elastically on demand as memory, CPU, network and storage demands increase. In particular, we propose to implement memory elasticity via the concept of elastic page tables (seen in Figure 3.1). Conventional page tables map a virtual page number within an address space to a physical page number in memory under the standard assumption that all of the physical memory is local. However, if we relax this assumption and permit different code and data pages from the same process to be placed or spread out across many machines, then this enables us to realize elasticity by adaptively changing the placement of pages across machines in response to the execution flow and data page access patterns.

One basic use case would involve for example a node that has run out of local memory and begun thrashing. In this case, a new cloud machine with available memory could be found, and some of the data pages in the current process could be moved to the newly available memory, thus stretching the process across two or more machines simultaneously. In this way, a process such as a large in-memory database could expand to operate over the collective memory of a large number of rack-mounted machines. Conversely, as demand drops, the process would contract to occupy less physical memory over fewer machines. While this may raise concerns about performance, we are inspired by prior studies that have shown that accessing memory on another machine across the network is faster than accessing disk, for example for swap space [26].
Figure 3.1: This diagram shows a hypothetical Elastic Page Table. We see that by adding an extra column for page location, in combination with the valid bit, allows us to locate a page to local/remote memory and local/remote swap disk.

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Phy. Addr</th>
<th>Valid</th>
<th>Node (IP addr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>1</td>
<td>Localhost</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>0</td>
<td>Localhost</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>1</td>
<td>128.138.60.1</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>0</td>
<td>128.138.60.1</td>
</tr>
</tbody>
</table>

Local Mem | Swap space | Remote Mem | Remote Swap
3.2 Our Model contrasted with prior work

A key mechanism in our proposed approach for ElasticOS is to move thread execution context towards its frequently accessed data pages (as seen in Figure 3.2) to amortize access cost.

What makes this approach fundamentally different is that we choose not to use data page replicas for the purpose of page access optimization, thereby avoiding the network cost of coherency messages. Of course, data page replicas will exist for fault tolerance purposes, but will be outside the application execution path. They will therefore have looser consistency requirements that do not block the application. Our interest is in investigating a model in which there will be one “active” copy of data pages – i.e., the copy that threads read/write to. The “active” data page set spread over several nodes is therefore like the distributed equivalent of a working set of data pages of a typical process. Additionally the read-only nature of code pages allows them to be replicated on every node without coherency costs. Thus our “stretched” process, by virtue of the Elastic Page Table is now an abstraction that spans the resources of several nodes i.e., A process can now be comprised of a collection of data pages spread across several nodes, code pages replicated on all

---

Figure 3.2: Accesses in ElasticOS
nodes, and execution context that hopscotches around as required.

This is in stark contrast with the design seen in Distributed Shared Memory (DSM) & its derivative systems, mentioned in Chapter 2 (and as seen in Figure 2.3), where cheap access cost was achieved by replicating data and maintaining consistency through the use of coherency protocols. An additional differentiating feature we should take note of is, a part of the previous DSM related work kept execution context fixed on a node. While other SSI related projects did explore moving execution context, they did so under the umbrella of full process migration used for CPU load balancing. From the vantage point of data accesses however, they all (and even newer projects [9]) retain the paradigm of data being moved (and cached) closer to execution streams on-demand and being kept current by the use of costly coherency mechanisms.

We think exploring our design choice to dynamically move thread execution context to follow data could potentially reveal several advantages for ElasticOS, some of which are:

- The first being that a generic “stretched” application process can indeed span beyond the resources (all resources, not just memory) of a single machine, without being rewritten, thereby providing native elasticity to user-space.

- The system will be able to reactively co-locate fragments of code and data that are strongly coupled and in doing so, naturally align available resources with any inherent thread level parallelism in the application

- Doing this has the nice property, particularly with multi-threaded applications, of possibly forming natural independent partitions of the application that can each coalesce on a different node and concurrently continue executing.

- This opens the door to new optimizations in the system that are possible because of the ability to move thread execution context – e.g., using lock wait time to co-locate a waiting thread to a machine where the lock owner is executing

- Such co-location could have two fold benefits of
* Reducing data access traffic (which is a promising scaling property)

* Being able to leverage existing, well engineered mechanisms in modern operating systems that handle concurrent access of data between threads/processes

With this scheme we see that a process can “stretch” in multiple dimensions (memory, computation and network and storage I/O). Figures 3.3, 3.4 and 3.5 illustrate our vision of elasticity along different dimensions and multiple stretched processes sprawl efficiently across resources on multiple nodes.

In some sense, a subset of our investigation (elastic memory) does revisit the problem domain of DSM. An analogy from past systems research that encourages us is that of virtualization technology from the 70s was revisited in the late 90s and led to a great wave of innovation in cloud computing. Therefore we think with advances in technology such as faster networks, new applications beyond high-performance computing, and with a new goal of elasticity in cloud computing, this new research direction for Operating Systems is both interesting and promising to explore. Having explained that we are fundamentally different from DSM, we remain humbly open to the numerous lessons we could learn from previous extensive DSM efforts on minimizing network traffic e.g., Munin DSM system [8], sending only the “diffs” in coherence messages to reduce communication and false sharing.
Figure 3.3: A “stretched” Process consisting of (a data page + a thread) on each node. It illustrates elasticity in multithreaded computation and memory where threads operate on independent data. If the threads happen to be I/O intensive, a similar layout would illustrate elasticity of I/O.
Figure 3.4: Even a single thread of execution can also be stretched. We see here that a single threaded Process, can stretch its memory by acquiring pages on both nodes and jumping execution between them as required. This illustrates elasticity in memory for a single thread and hints at how locality in access patterns can be used for page placement on nodes.
Figure 3.5: Here we see multiple stretched Processes running in an ElasticOS. Threads accessing the same data pages co-locate on one node forming a natural grouping. When on the same node, the conventional techniques for concurrent access may be leveraged.
Chapter 4

Our Experiments and Results

Is Elasticity Achievable?

Now that we have described the ElasticOS vision as we see it, we are faced with the research task of examining feasibility challenges. To gauge the feasibility of the ElasticOS vision, we posed 2 fundamental questions:

- **Locality Patterns and Run-time detection:**
  
  Do common applications exhibit locality patterns in their page accesses that an ElasticOS can exploit? If so, can these patterns be detected and exploited at run-time?

- **Cost of execution context jumps/data page relocations:**
  
  *(Since execution context jump is a key mechanism that we’re going to rely on)* How expensive an operation is it? What are the typical latency costs of an execution context jump/page relocation on today’s hardware?

We designed our experiments around these 2 important questions. The experiment descriptions, methodology and results are presented in the following sections:

4.1 **Locality Patterns and Run-time detection**

4.1.1 **Looking inside an application**

To answer the first part of the question of whether common applications display patterns of locality in their page access patterns, we need to peek inside an application in some manner. We
were able to do so using Intel's PIN framework [15, 24]. It is a dynamic binary instrumentation tool that uses just-in-time compilation to introduce instrumentation code into an application at runtime. Using its hooks we were able to write a tool to obtain a trace of the page access patterns seen in a running application by monitoring the values of the instruction pointer and the memory operands of an instruction. An example of such a trace is as shown in Figure 4.1. Remembering our model of being able to cheaply replicate an applications code pages (because they are read-only) on all participating nodes, we're particularly interested in learning the patterns behind data page accesses. We came up with an undirected graph representation to capture the sequence of data pages accesses that show up in a trace, also shown in Figure 4.1.

Using the same procedure, we attempted to ascertain what such a graph would look like for a real world application. Shown in Figure 4.2 is the interesting structure we see in this graph for an idle MySQL daemon. The trace and graph were obtained just as described above. We used the spring layout algorithm from the NetworkX library to render this graph. The spring layout is an algorithm to generate the visual layout of all the nodes of the graph prior to rendering it. It runs an internal simulation of the nodes and edges like a molecular system with attracting and repelling forces that eventually dampen out and the steady state (3-dimensional) positions of the nodes are what get rendered. From our experience we found that this layout visually best represents a dense graph with a thick edge structure.

4.1.2 Simulation of Application Execution on ElasticOS

After obtaining an applications trace as described in sub-section 4.1.1, Figure 4.1 we tried to simulate the overhead that would be encountered by the application, had it executed on an ElasticOS. More specifically, we did the following in a Python simulation script:

- We predetermine fixed points of scalability (i.e., number of physical nodes under the ElasticOS),
- We make a first pass through the trace to learn the unique data pages that are accessed
Figure 4.1: The table above shows an example trace from instrumenting an application with our PIN tool. Below it we have an our graph representation (derived from this trace) the data pages accessed.
Figure 4.2: Shown here is a spring layout graph representation of the data page accesses trace obtained by instrumenting an idle MySQL daemon. The daemon was instrumented for 60 seconds including its start-up/initialization activity. Seen here (colored) are 2 “clusters” of locality i.e., where members of one group are often accessed together.
and then partition/distribute them across those nodes (e.g., page 4 resides on node 3, page 1 resides on node 5 ...)

- Lastly we walk through the trace (the second pass), keeping a count of how many times we would have had to jump execution across nodes in order to get to a data page \(^1\). This is followed by generating plots of our results.

4.1.3 Application Partitioning: Our initial attempt with Static Algorithms

We initially investigated static partitioning algorithms like METIS [17, 16] and its parallel versions [38] and some of our own simple partitioning techniques (i.e., partitioning sequentially, partitioning randomly) as a means for ElasticOS to discover such clusters of page groups. We found that the number of transitions for these algorithms was quantitatively very high (e.g., an example of the order of numbers we saw were 10000 per second, which would likely kill the feasibility of any distributed system with a comparable network transaction rate). After some spent effort we came to some realizations on the deficiencies and drawbacks of this approach:

- It requires full a-priori knowledge of the application trace. This is not convenient as in ElasticOS, we would like the operating system to learn this “on-the-fly” from a running application with no prior instrumentation or modification.

- Static analysis/other PL techniques would again not be scalable as we are targeting the generic application.

- Sophisticated algorithms like METIS take too long to run. Given our on-the-go requirement for forming such partitions, this is again less than ideal.

- Most importantly, all of these algorithms are Static i.e., a page gets assigned to a node and remains there for the lifetime of the application. We think this was the key insight into understanding why the number of transitions obtained was unreasonably high.

\(^1\) In the rest of this document, we’ll refer to a movement of a datapage as a “datapage relocation”, movement of execution context as “execution context jump” and when referring to either generically as “transitions”
It is our goal that an in-memory application, that is elastic, would eventually form natural groupings of data pages that are frequently accessed together on each node. As access patterns of the application temporally change, these groupings can change. In order to detect these changes “on-the-fly” an ElasticOS would have to employ a page placement algorithm which partitions the application adaptively. Therefore, we concluded from this exercise that for a feasible ElasticOS, an dynamic adaptive strategy for page placement would be required.

4.1.4 Multinode Adaptive LRU algorithm

As an initial algorithm, we designed the Multinode Adaptive LRU algorithm (illustrated in Figure 4.3) that works as follows: when an execution stream references a page on a remote node, we initially choose to move the page to the location of the execution stream – that is, pull the remote page to the current node. Eviction of a page in order to accommodate a pulled page is done on a least recently used (LRU) basis i.e., the working set of a process on each node over time becomes an LRU based grouping from which pages are accessed together on that node. Upon repeated data page “pulls”, beyond a temporal threshold (for our experiment we used a % of pages pulled from a remote node in the last 10 pages accessed, but other definitions are also plausible), we move the execution context to the target node. In doing this we are likely to exploit locality in the subsequent page accesses.

Lower threshold values make the algorithm more responsive in moving execution context, which can hurt performance by moving away from locality prematurely. Higher thresholds make the algorithm more sluggish in moving execution context, and can cause an application to be late in exploiting locality at a remote node.

4.2 Candidate Applications and Workloads

To test our new algorithm, we selected 2 candidate applications which we believe are good candidates to elasticize:
Figure 4.3: Multinode adaptive LRU algorithm
MySQL Daemon: This is a memory intensive, real world application that is widely used.

* In our experiments MySQL was being driven by a customized MySQL-Bench [31] workload.

apache2: This is a web server application that spawns off handler threads to service requests while being I/O intensive. It too enjoys a large user base.

* In our experiments apache2 was driven by the command-line `ab` Apache HTTP Benchmarking tool.

The workloads themselves are further described in Appendix B.

Relevant snippets of code we used are in Appendix C

They are both clearly applications whose utilities would be tremendously increased if they were able to benefit from elasticity without having to be rewritten.

4.3 Simulation Results

We conducted 2-node simulation experiments for these applications and workloads as described above to see how our algorithm fares with these real world applications. We measured how many execution context jumps and data page pulls took place. The simulation was repeated for various threshold values.

The results for MySQL are seen in Figure 4.4 and for Apache2 in Figure 4.5.

Both sets of results show that as we vary the threshold, we can tradeoff between execution context jumps and data page pulls. We also see that the optimal tradeoff is found for our MySQL experiment at Threshold = 20% and for our Apache2 experiment at Threshold = 40%. Its apparent that this optimal point is not just application specific but also workload specific. Temporally, this makes it a moving target. Finding the right “sweet spot” between this tradeoff in real time, is one of the problems we plan to investigate in our future research.
Figure 4.4: Simulation of the multinode adaptive LRU algorithm for a 334 second MySQL execution trace. There were 3287 data pages in this test. The optimal threshold was at 20% for which the number of jumps + pulls was 9 per second.
Figure 4.5: Simulation of the multinode adaptive LRU algorithm for a 393 second Apache2 execution trace. There were 606 data pages in this test. The optimal threshold was at 40% for which the number of jumps + pulls was 457 per second.
4.4 Cost of execution context jumps/data page relocations

As other researchers have shown, it is extremely fast to pull a data page from across the network [32]. In order to verify that performing an execution context jump to a remote machine is likewise fast, we set out to measure this within the Linux 3.2 kernel by implementing an initial version of a stretched process. This demonstrates that we will be able to realize ElasticOS with minimal changes to a commonly used operating system, but lacks many of the features of our overall architecture. Our test process consisted of a simple program that loops and increments a counter stored on a data page and a second counter stored in a register. We wrote a system call for this test process to invoke the movement of execution context and a data page to a location where we had pre-distributed its code pages. The actual state transfer mechanism was implemented within our initial version of the ElasticOS Manager component (described further in Appendix A), and was done over a persistent TCP connection. This test verified the feasibility of execution state transfer – e.g., the counters would count to 10 on one machine, then execution state would be transferred, and the counters would resume counting up 11-20 on the other machine, followed by another state transfer and a count up 21-30 on the original machine, and so forth as shown in Figure 4.6. The average measured latency of a state transfer was approximately 0.4 ms over a gigabit Ethernet link.

4.5 Inferences & Takeaways

Based on these results, we conclude that our algorithm was effective, for these particular workloads, in being able to maintain a reasonable page grouping that dynamically captures some application page access locality. We intend to explore this design space further in our search for an algorithm that adapts well to various applications and realistic workloads.

The 0.4 ms latency represents a pause in the execution of a single thread (parallel execution within the entire application ensures that this is not simply idle time). As such, this results in a slowdown of the application as compared to the case where a single machine has infinite memory.
Figure 4.6: Execution Context Switch experiment to measure state transfer overhead
(and therefore never swaps to disk).

More specifically:

- **MySQL**: For the threshold of 20%, the combined transitions (jumps + pulls) of our MySQL trace was approximately 9 per second. Using 0.4ms for each, implies a delay of 36ms for every 1 second of actual execution, or a slowdown of 3.6%

- **Apache2**: For the threshold of 40%, the combined transitions (jumps + pulls) of our Apache2 trace was approximately 457 per second. Using 0.4ms for each, implies a delay of 182ms for every 1 second of actual execution, or a slowdown of only 18.2%

While these slowdown estimates are reasonable, in our final system we do not expect slower execution, but instead opportunistic improvements over operations such as swapping to disk.

To envision the Multinode Adaptive LRU Algorithm at its best performance against the NSWAP system, lets consider a hypothetical example of a sequential workload (i.e., pages are accessed in sequential order, one after another and looping around in the end). Lets say we have “n” pages in our working set which are split over 2 nodes and we create a partition of the first “n/2” nodes residing on node 1 and the next “n/2” nodes residing on node 2. If we step through the execution of this contrived workload, we’ll see that NSWAP causes “n/2” page pulls and “n/2” page evicts, thereby causing about “n” network operations on sequence of page accesses. The multinode adaptive LRU algorithm will only cause Threshold page pulls before jumping to the second node and thereafter exploiting all the locality there. After the end of the sequence, it will again cause Threshold page pulls before jumping back to node 1. The threshold value is always a small percentage of the working set size (e.g., Threshold = 20%) Therefore we see that in one sequence of page access for the workload, NSWAP causes “n” network operations where Multinode Adaptive LRU causes \( \frac{20}{100} \cdot n/2 = n/10 \) network operations (lesser than NSWAP by a factor of 10).

To see how much our algorithm stresses the network when running real world applications, lets follow this thought and calculate rough estimates of overheads in ElasticOS compared with other systems (i.e., Swapping to Disk and Swapping to network i.e., NSWAP [32]). We first make
the following observations:

- The rough access time for one 4K (page sized) operation from hard disk as reported in recent research [36] can be shown to be 3.3ms (i.e., 300, 4K transactions per second).

- Correspondingly if we go over the network for the same access (i.e., in case of NSWAP or when ElasticOS pulls a data page or when ElasticOS jumps one execution context), the delay is 0.4ms as established earlier.

- In the Multinode Adaptive LRU algorithm, when threshold values are very high and the number of jumps is zero it implies that an execution context jump never took place and the application only pulled pages in. This is therefore reduces to NSWAP behavior.

- The number of operations in the NSWAP case and regular SWAP disk are always equal. The only difference being in NSWAP, the page is fetched over the network.

Table 4.1 shows the performance comparison estimates for the MySQL run and Table 4.2 shows the performance comparison estimates for the apache2 run on various systems (Regular SWAP Disk, NSWAP, ElasticOS running the Multinode Adaptive LRU Algorithm) \(^2\). The term “Operations” in the header refers to either:

- Fetching a page from local disk i.e., regular SWAP space

- Fetching a page from remote memory i.e., NSWAP or ElasticOS data page pull

- ElasticOS execution context jump (with 4k i.e., 1 page worth of state being transferred)

We therefore see from the Tables that Multinode Adaptive LRU Algorithm does perform strictly better than NSWAP in both cases, but only marginally. It appears that the intuition of the algorithm works, but the gains are only incremental on the real workloads that we ran compared

\(^2\) Please note that the Overheads listed in these Tables are the rates or overheads “per second”, which can accumulate quickly over an applications up-time. Therefore small differences here lead to large differences in reasonably long running applications
Table 4.1: This table shows comparison of MySQL performance estimates on SWAP, NSWAP and Multinode Adaptive LRU. Notice that ElasticOS has a 10% improvement over NSWAP and about 90% improvement over regular SWAP space.

<table>
<thead>
<tr>
<th>System</th>
<th>Time (ms)/4K operation</th>
<th>Number of Operations/second</th>
<th>Total Overhead/second of execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWAP</td>
<td>3.3</td>
<td>10</td>
<td>33</td>
</tr>
<tr>
<td>NSWAP</td>
<td>0.4</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>ElasticOS</td>
<td>0.4</td>
<td>9</td>
<td>3.6</td>
</tr>
</tbody>
</table>

to the “ideal” sequential case. This implies that for these workloads its not doing a much superior job than NSWAP in detecting page access locality. This is an area we intend to investigate in the future. From our experience, we think one or more of the following issues are suspect in explaining these numbers:

- **Are there locality patterns in these workloads?**: The questions we’d like to investigate here are:
  
  * When comparing to the “ideal” case of the sequential workload, do our workloads display some form of locality?
  
  * How close is this workload to a sequential workload Vs a random access workload?

- **Could we potentially create a better driver workload?**: The performance of this algorithm and its optimal threshold is workload and application dependent. Perhaps it might be possible to create a workload driver for each of these real work applications that

Table 4.2: This table shows comparison of apache2 performance estimates on SWAP, NSWAP and Multinode Adaptive LRU. Notice that there is only a marginal improvement over NSWAP and about 88% improvement over regular SWAP space.

<table>
<thead>
<tr>
<th>System</th>
<th>Time (ms)/4K operation</th>
<th>Number of Operations/second</th>
<th>Total Overhead/second of execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWAP</td>
<td>3.3</td>
<td>458</td>
<td>1511.4</td>
</tr>
<tr>
<td>NSWAP</td>
<td>0.4</td>
<td>458</td>
<td>183.2</td>
</tr>
<tr>
<td>ElasticOS</td>
<td>0.4</td>
<td>457</td>
<td>182.8</td>
</tr>
</tbody>
</table>
better creates locality in their page accesses. We may need to redesign the workloads we used to drive these tests.

- **Could we tune the algorithm better?:** Locality patterns can take place in several different time scales. The temporal window size we used for our algorithm was fixed. Perhaps this parameter of the Multinode Adaptive LRU algorithm can be better tuned to capture locality for these workloads.

- **Could we define a better algorithm?:** It may be that our algorithm is too crude and not refined enough to catch finer grains of locality in these workloads.

- **Is the workload size sufficient?:** It could be that our experiment is partitioning too small a problem size i.e., of “n/2” of the working set size is too small, there is no sufficient locality on either node to make a difference to number of times we go over the network.

- **Temporal characteristics of jumps:** The average bandwidths usage averaged over roughly 5 minutes in these tests was almost similar for NSWAP and ElasticOS. We would also like to compare the peak bandwidths i.e., which is more bursty and which is more evenly distributed.

Clearly this area needs more work to refine how we are going to detect locality in an ElasticOS and it will definitely be a central focus of our immediate work going forward. However, we should note that with the right selection of threshold value, ElasticOS (Multinode Adaptive LRU) can be reduced to NSWAP behavior. Therefore, if we are able to auto-tune threshold intelligently & adaptively in an eventual ElasticOS, the lower bound of our performance will be that of NSWAP.

Compared to regular SWAP space operations however, we see that ElasticOS estimates are significantly better (as are NSWAP’s). It is this performance gap between swapping to local disk and network operations that we hope to exploit when using execution context jumps to minimize or avoid altogether application any slowdown. We do however hope to find a way increase the performance gap between ElasticOS and NSWAP in our future work.
The experiments confirmed our hunch about a dynamic algorithm being better suited for the purposes of page placement in an ElasticOS. It also gives us reasonable optimism about the search space of network overheads we are likely to encounter when building a real ElasticOS.

4.6 Limitations of our experiments

Our trace experiment is approximate by nature. Of course, an instrumented application is going to run slower than when run natively. Although we haven’t formally quantified it, we did empirically compare it to the same experiment run on the same application uninstrumented to find that the overall runtime overhead was not more than 10%-20% of native runtime. An uninstrumented application running natively however may have some different temporal characteristics.

The trace is also approximate in the sense that for multi-threaded application like the ones we instrumented, the trace itself represents the interleaved page access patterns of all threads of the application.

We should also note that the overhead of state transfer in applications running on an ElasticOS would vary depending on application activity to some extent rather a fixed amount each transition. We therefore don’t have a full quantitative measure yet of how much state would need to be exchanged over the network.

Lastly we’d like to mention that our counts of execution context jumps + data page relocations are averaged over the experiment run. We have not investigated its temporal characteristics like when it spikes and how (un)evenly its distributed over the runtime of the application.

All of these limitations also serve as possible points for improvements we could make to these experiments. While our experiments are interesting and can be further enhanced in their precision like we have highlighted above, they have helped lead us to some fundamental insights about approaching ElasticOS design (e.g., revealing the important jumps Vs pulls tradeoff) that will help us when attempting an implementation.
Chapter 5

Future Research Areas

There are a host of additional exciting open research questions that need to be solved in order to fully realize the promise of ElasticOS.

5.1 The Page Placement Algorithm

Given the discussion in the previous chapters, a ripe and potentially fruitful area for future research would be that of Page Placement Algorithms. We’d like to be able to enhance our Multinode Adaptive LRU Algorithm or create an alternative algorithm that is more sophisticated. In the former case, the issue of automatic, application specific, adaptive threshold detection is an interesting area to explore as already mentioned earlier. This area will perhaps have the greatest influence on how applications perform in an ElasticOS. Reactive algorithms that adapt to improve application performance and are some of the desirable qualities in the vision we are chasing.

5.2 Page Placement: Distributed Decision Making

Since ElasticOS spans multiple machines, then the page placement policies are distributed across many computers. An open question we expect to explore is to what extent should this elastic decision-making be centralized in say a master-slave(s) arrangement, or be fully decentralized.
5.3 Resource Discovery

With ElasticOS we can stretch a process across multiple machines. Knowing what machines there are and what resources are available is an important open problem. Further, this information on availability and reservation of resources cannot be volatile but must persist for some time. Even more, how this works within the hosted cloud computing model is an open question. Currently, offerings such as Amazon EC2 offer customers the ability to lease individual VMs of fixed sizes. A customer could run ElasticOS within its collection of VMs and expand and contract at some coarse timescale. Alternatively, a cloud provider could enable a finer granularity of resource allocation – effectively placing the elasticity within the hypervisor.

5.4 Elastic I/O

Stretching virtual memory across a network provides only one dimension of stretching. We also envision stretching of other computer resources such as CPU and I/O (namely network I/O). Network I/O presents an interesting challenge due to addressing issues. In particular, a packet directed to the process has a destination IP address that, due to the current technology, will be directed to a single server. Outgoing packets can come from any of the stretched machines. Stretching the I/O without custom virtual appliances is a significant research direction to explore. We intend to examine and leverage advances in software-defined networking (such as with OpenFlow [27]) as an interface to achieve this. Encouraged by contemporary research [35] in this direction that targets network middleboxes, we would like to extend network elasticity more generally to a large range of applications.

5.5 Fault Tolerance

Stretching a process over multiple machines has the potential to increase the application’s susceptibility to failure. To combat this, we intend to investigate snapshot techniques, rollbacks, distributed check-pointing, and replication. In addition, we will strive to place fault-tolerance
functions off the critical path of normal execution to minimize their impact on performance. We hope to explore questions like the limits and technical challenges of spreading an application out or complexities reeling an application back in.

There are of course numerous other related areas that we could add to this list but are skipping for brevity.
Chapter 6

Conclusion

This work introduced the concept of elasticity in an operating system by stretching the execution of a single process or thread over many machines without requiring new programming models or cumbersome new scripting logic. The idea of elastic page tables was introduced thereby implying a revised abstraction for the process in an operating system. We studied the internal structure (page level access patterns) of two real world applications, MySQL and Apache2.

We reasoned towards needing to partition the application across several nodes using a page placement strategy and arrived at the revelation of needing an adaptive one. We presented our adaptive page placement strategy called the Multinode Adaptive LRU Algorithm, which pulls data pages towards the thread execution stream by default and upon exceeding a temporal threshold of such pulls, moves the execution stream to the node where the desired data page is located. In doing so it gains the ability to take advantage of page locality and forms reasonable partitioning of the applications pages across nodes. Further, it can be extended to any number of nodes.

Our experiments with this algorithm helped reveal an important characteristic of ElasticOS, namely the tradeoff relationship between execution context jumps and data page pulls for an application (i.e., when one decreases the other increases). Our experiments also revealed the important fact that there exists a threshold where this tradeoff is optimal (i.e., produces minimal network traffic) and that this optimal threshold value itself is application and workload dependent. Another set of experiments on Linux 3.2, helped us estimate the network overhead in transferring execution context/relocating a data page between nodes on todays hardware. These experiments
collectively show that with optimal selection of the threshold, common real-world applications would not suffer a heavy network cost to execute on an ElasticOS. We are encouraged by these results and intend to pursue our next steps in designing a provably better page placement algorithm and an initial implementation of the elastic page tables functionality in Linux 3.2.

We’ve highlighted some exciting new research issues in elasticity for OS researchers like us to solve down the road (e.g., like page placement algorithms, adaptive threshold detection, fault tolerance, dealing with stacks in a stretched process, open files/sockets, DLL’s, resource discovery and I/O elasticity). Beyond the ones we’ve already mentioned, we believe there are a wealth of open research topics that exist within the space of ElasticOS, some which we may not even have stumbled upon yet.

As we continue our efforts in this area, these investigations give us the impression that ElasticOS is an rich area full of research challenges and interesting problems while being immensely promising in its potential impact.
Bibliography


Appendix A

ElasticOS Architecture

This section presents a possible architecture that we could use for ElasticOS. The design presented here is joint work by Dr. Eric Keller, Dr. Richard Han and Ehab Ababneh, coauthors of our USENIX HotOS 2013 paper [13].

The central concept to achieving an ElasticOS is that a given address space for a process (i.e., a collection of pages) can be spread across multiple machines automatically. For this, ElasticOS needs to make decisions as to which pages should be placed on which machines and the mechanisms to transition execution from one machine to another. Shown in Figure A.1 is the initial design of the ElasticOS architecture with a number of components to achieve the necessary functionality.

A.1 Elastic Page Table for Tracking Remote Pages

The traditional page table provides a mapping between virtual addresses and physical addresses. Since virtual pages do not have to be currently in physical memory, a valid bit in the page table entry indicates whether the page is in memory or not. Traditionally, this implies that the page is on disk and must be swapped from disk when it is needed. With ElasticOS this implied relationship does not hold as a page not in memory may actually be located on a remote machine. We extend the traditional page table to include an extra field to note the machine that is holding the page – nominally shown as an IP address for illustration as any addressing scheme can be used. Note that the machine holding the page may be the local machine, which indicates the page is on the local disk (i.e., valid bit set means in memory, not valid and machine ID is local means on
Figure A.1: ElasticOS Architecture.
disk, not valid and machine ID is remote means the page is on a remote machine). Of course, the processor’s memory management unit (MMU) will not (currently) recognize the ElasticOS modified page table entry, so it’s implemented as a separate data structure, but used internally by the operating system as a single table. The page table will differ for each machine as a page that’s local on one machine will be remote on another – maintaining the page to machine mapping falls to the ElasticOS Manager discussed below.

A.2 Modified Page Fault Handler to Transition Execution

As ElasticOS expands to other machines automatically. In doing so, pages may be unavailable on the current machine. When the page is not currently in memory (i.e., the valid bit in the page table entry is not set), the processor will trigger a page fault. The traditional page fault handler must be modified to handle the new case where a page fault does not necessarily mean fetch from disk. If the machine ID in the elastic page table indicates the page is local, then the page fault handler will proceed as normal and swap from disk. If the page is remote, there are two possible actions that can occur: (i) pull the remote page across the network to the local memory (as with network-based memory swapping [26, 32]), (ii) transition execution to the other machine. In either case, with ElasticOS we will put the process in the suspended state and notify the ElasticOS manager, which will perform the action.

A.3 ElasticOS Manager for Intelligent Orchestration

Separate from the low-level page handling, the ElasticOS manager (EOM) provides the intelligence in the ElasticOS. There are two basic functions of the EOM: (i) make page placement decisions across the machines, and (ii) perform the execution transition.

A.4 Page placement decisions

It is the responsibility of the EOM to determine page clusterings and move pages between machines as necessary. This is executed as a long running function that will continually monitor the
state of the entire execution and act accordingly – moving pages between machines and updating
the page table structure accordingly. One such indicator may be thrashing, where the same page
is being swapped to and from disk many times in a short period of time. The EOM will take this
as an indicator that the working memory set is greater than the physical memory size and that
expansion to another machine should occur. The EOM will also decide how a page fault for a
remote page should be handled – i.e., pulled to the local machine or transition execution to the
other machine.

A.5 Transition execution

The decision to transition execution to another machine is based on the overall page placement
policy – e.g., based on analysis whether this will lead to execution that accesses more remote
pages or whether this is a single page but future execution will mostly be on the current machine.
In the case where a transition is determined, the page fault handler had put the process in a
suspended state. The EOM must then copy to the remote machine the necessary register state
such that execution can continue in the exact same state. Memory pages may also be copied over
to prevent needing to pull the pages when they are accessed on the remote machine. Note that this
transitioning of execution is distinctly different from process migration [39] where the entire process,
and all of its associate memory, are migrated to another machine. In contrast, in ElasticOS, the
process is simply stretched across multiple machines and state is synchronized during a transition.

A.6 Kernel Hooks to Enable User-space Management

Kernel-level functionality should be restricted to only what is needed due to the position of
the kernel. Putting complexity in user-space not only increases security and stability of the system
but also provides an easier mechanism to add functionality. As such, the ElasticOS Manager resides
in user-space and the ElasticOS kernel must provide hooks to enable this user-space management.
While the full extent of the API is still to be determined, we can discuss some initial functions.
First, the ability to get and set pages is needed to perform the page movement as a result of the page
placement decisions and during execution transitions. Second, access to register state of a process is needed during the transition of execution. Finally, counters and various other measurements need to be exposed to enable intelligent page placement decisions (e.g., to help detect thrashing).

In this role as a user-space implementation, the ElasticOS manager is akin to the management OS in virtualization technology (e.g., dom0 in Xen) – the hypervisor provides access to the low level functionality of the hypervisor, but does not contain the intelligence to carry out complex actions.

For example, in live VM migration, the migration is initiated through commands in the management OS which opens a connection to another machine’s management OS, potentially over ssh. The management OS will then perform the data transfers by interacting with the hypervisor to access the pages and find out which pages were modified.
Appendix B

Workloads for our experiments

This section provides more information on the workloads that drove our experiments whose results were presented in Chapter 4.

B.1 SQL-Bench Workload

SQL-Bench is a benchmark shipped with the MySQL source code. It is a collection of Perl scripts meant to test various SQL transaction types (creating/deleting tables, inserting, deleting entries, various complex queries etc.) and collective benchmark the performance of a MySQL database. This benchmark is single-threaded and measures the minimum time for the operations performed. We chose to customize one of their scripts for our purposes.

To setup our test, we first create 2 tables containing 10000 entries each. The relevant code snippets is shown in Appendix C, Section C.1.

The driver to the MySQL daemon that constitutes our test itself basically accesses an entry from each table 1000 times. What we are attempting to do is to create a pattern of locality in data page access patterns (since 2 entries across these 2 tables are likely to reside on different data pages) that show up in the trace. The relevant code snippet is shown in Appendix C, Section C.3.

Prior to doing this however the script accesses all entries to do a “burn-in” so that the tables are read from disk into memory prior to the test beginning (i.e., This part of the run are excised from the trace when running the simulation of SQL execution on ElasticOS).
B.2 Apache HTTP benchmark tool - ab

ab is a command line benchmarking tool for stressing Apache HTTP server. It is used to essentially test/demonstrate how many requests per second the apache2 server can serve.

Its available on most Unix/Linux machines and its manpage “man ab” is the best reference that gives elaborate details on all of its options.

A quick overview however, is:

`ab [options] [http[s]://]hostname[:port]/path`

Options are:

- `-n requests` Number of requests to perform
- `-c concurrency` Number of multiple requests to make
- `-t timelimit` Seconds to max. wait for responses
- `-b windowsize` Size of TCP send/receive buffer, in bytes
- `-i` Use HEAD instead of GET
- `-X proxy:port` Proxyserver and port number to use
- `-k` Use HTTP KeepAlive feature
- `-d` Do not show percentiles served table.
- `-r` Don’t exit on socket receive errors.
- `-h` Display usage information (this message)
- `-Z ciphersuite` Specify SSL/TLS cipher suite (See openssl ciphers)
- `-f protocol` Specify SSL/TLS protocol (SSL3, TLS1, or ALL)

The options we use:

`ab –k –t 300 –c 50 –n 50000000 http://localhost/`

We arrived at our choices after some experimentation and empirically observing that they stress the natively running webserver sufficiently (i.e., begin to cause more divergent transaction...
processing times).

Or in other words we chose the following:

- Run the test for 300 seconds to get a run comparable to the MySQL benchmark test.

- We chose upto 50 million transactions for the test.

- We chose to have 50 concurrent persistent TCP connections over which these transactions would be spread.

- The webpage served up to clients was http://localhost/.
Appendix C

Code Snippets

C.1 Initializing SQL tables before the experiment

This code drives the initialization to setup the MySQL as described in Chapter 4.

```perl
#!/bin/bash

# Create needed tables

# bench1

goto select_test if ($opt_skip_create);

print "Creating table\n";

$dbh->do("drop table bench1" . $server->{'drop_attr'});

do_many($dbh,$server->create("bench1"),

["region char(1) NOT NULL",
 "idn integer(6) NOT NULL",
 "rev_idn integer(6) NOT NULL",
 "grp integer(6) NOT NULL"],

["primary key (region,idn)",
 "unique (region,rev_idn)",
 "unique (region,grp,idn)"]);

if ($opt_lock_tables)
```
{
  do_query($dbh, "LOCK TABLES bench1 WRITE");
}

if ($opt_fast && defined($server->{vacuum})) {
  $server->{vacuum}(1,$dbh);
}

###
### Insert $opt_loop_count records with
### region:"A" -> "E"
### idn: 0 -> count
### rev_idn:count -> 0,
### grp:distributed values 0 -> count/100
###

print "Inserting $opt_loop_count rows\n";

$loop_time=new Benchmark;

if ($opt_fast && $server->{transactions}) {
  $dbh->{AutoCommit} = 0;
}

$query="insert into bench1 values (":
$half_done=$opt_loop_count/2;
for ($id=0,$rev_id=$opt_loop_count-1 ; $id < $opt_loop_count ; $id++,$rev_id--)
{
  $grp=$id+3 % $opt_groups;
  $region=chr(65+$id%$opt_regions);
  do_query($dbh,"$query $region ,$id , $rev_id , $grp")
};
if ($id == $half_done)
{
    # Test with different insert
    query="insert into bench1 (region, idn, rev_idn, grp) values (";
}

if ($opt_fast & $server->{transactions})
{
    $dbh->commit;
    $dbh->{AutoCommit} = 1;
}

$end_time=new Benchmark;
print "Time to insert ($opt_loop.count): ",
timestr(timediff($end_time, $loop_time)," all") . \\

if ($opt_lock_tables)
{
    do_query($dbh,"UNLOCK TABLES");
}

if ($opt_fast & defined($server->{vacuum}))
{
    $server->vacuum(0,\$dbh,"bench1");
}

if ($opt_lock_tables)
{
    do_query($dbh,"LOCK TABLES bench1 WRITE");
}

#bench2
goto select_test if ($opt_skip_create);
print "Creating table\n";
$dbh->do("drop table bench2" . $server->{'drop_attr'});

do_many($dbh, $server->create("bench2",

    ["region char(1) NOT NULL",
    "idn integer(6) NOT NULL",
    "rev_idn integer(6) NOT NULL",
    "grp integer(6) NOT NULL"],
    ["primary key (region, idn)",
    "unique (region, rev_idn)",
    "unique (region, grp, idn)"]));

if ($opt_lock_tables)
{
    do_query($dbh, "LOCK TABLES bench2 WRITE");
}

if ($opt_fast && defined($server->{'vacuum'}))
{
    $server->vacuum(1, $dbh);
}

###
### Insert $opt_loop_count records with
### region:"A" -> "E"
### idn: 0 -> count
### rev_idn:count -> 0,
### grp:distributed values 0 -> count/100
###

print "Inserting $opt_loop_count rows\n";

$loop_time=new Benchmark;
if ($opt_fast && $server->{transactions})
{
    $dbh->{AutoCommit} = 0;
}

$query= "insert into bench2 values (" ;
$half_done=$opt_loop_count/2;
for ($id=0, $rev_id=$opt_loop_count-1 ; $id < $opt_loop_count ; $id++, $rev_id--)
{
    $grp=$id*3 % $opt_groups;
    $region=chr(65+$id%$opt_regions);
    do_query($dbh,"$query'".$region"', $id , $rev_id , $grp")");
    if ($id == $half_done)
    {  
        # Test with different insert
        $query= "insert into bench2 (region, idn, rev_idn, grp) values (" ;
    }
}

if ($opt_fast && $server->{transactions})
{
    $dbh->commit;
    $dbh->{AutoCommit} = 1;
}

$end_time=new Benchmark;
print "Time to insert ($opt_loop_count): " .
timestr(timediff($end_time, $loop_time), "all") . "\n\n";
if ($opt_lock_tables)
{
    do_query($dbh,"UNLOCK TABLES" );
if ($opt_fast && defined($server->{vaccum}))
{
    $server->vaccum(0,$dbh,"bench2");
}

if ($opt_lock_tables)
{
    do_query($dbh, "LOCK TABLES bench2 WRITE");
}
C.2 SQL Benchmark Driver

This code drives the actual test of MySQL as described in Chapter 4.

```
print "Testing the speed of selecting on keys that consist of many parts\n";
print "The test−table has $opt_loop_count rows and the test is done with $columns
ranges.\n\n";

###
### Connect and start timing
###
$dbh = $server−>connect();

###
### Do some selects on the table
###

select_test:
# amits test
if ($limits−>['group.functions'])
{
    #To make sure its all in memory
    fetch_all_rows($dbh,"select * from bench1");
    fetch_all_rows($dbh,"select * from bench2");

    print "Warm up complete\n";
    sleep(5);

    while(1) {
```
## First access table1

### First ensure that the table is read into memory

```perl
$fetch_all_rows($dbh,"select sum(idn+$tmp),sum(rev_idn-$tmp) from bench1");
```

```perl
printf "accessing table bench1\n";
```

```perl
for ($i=0; $i < 1; $i++) {
    my ($tmp); $tmp=1000;
    #fetch_all_rows($dbh,"select * from bench1 where idn=6");
    $loop_time=new Benchmark;
    for ($tests=0 ; $tests < 1000 ; $tests++) {
        #fetch_all_rows($dbh,"select sum(idn+100),sum(rev_idn-100) from bench1");
    }
    fetch_all_rows($dbh,"select * from bench1 where grp=6");
}
```

## Then access table2

### First ensure that the table is read into memory

```perl
$fetch_all_rows($dbh,"select sum(idn+$tmp),sum(rev_idn-$tmp) from bench2");
```

```perl
printf "accessing table bench2\n";
```

```perl
for ($i=0; $i < 10; $i++) {
    my ($tmp); $tmp=1000;
    #fetch_all_rows($dbh,"select * from bench1 where idn=6");
    $loop_time=new Benchmark;
    #for ($tests=0 ; $tests < $opt_loop_count ; $tests++)
    for ($tests=0 ; $tests < 1000 ; $tests++)
        { #fetch_all_rows($dbh,"select sum(idn+100),sum(rev_idn-100) from bench1");
```
62 fetch_all_rows($dbh,"select * from bench2 where grp=88");
63     }
64 
65 }
66     }
67 }
C.3 Page Trace Tool in INTEL PIN Framework

This is one of the variants of our tool within the INTEL’s PIN framework that allows us to get a trace dump of page accesses. We ran various variants of this tool with changes to get specific accesses (e.g., libraries, stack accesses) etc. We include it here to give an idea of the tool used to instrument our candidate application. For instructions on how to run such a tool within the PIN framework itself, please refer to the documentation available on the Intel’s website [15].

/*BEGINLEGAL

Intel Open Source License

Copyright (c) 2002–2012 Intel Corporation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of the Intel Corporation nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ‘AS IS’ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE INTEL OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

ENDLEGAL */
/*
* This file contains an ISA-portable PIN tool for tracing memory accesses.
*/

#include <stdio.h>
#include "pin.H"
#include <time.h>
#include <string.h>

//#define DONT_LOG_MULTIPLE_ACCESSSES_BETWEEN_THESAME_CODEPAGE_ANDDATAPAGE
//#define LOG_APPLICATION_INSTRUCTIONS_ONLY
//#define TAG_STACK_ACCESSSES
//#define LOG_LIBRARY_ACCESSRESSES_TO_HEAP
#define APPLICATION_NAME "apache2"
//#define APPLICATION_NAME "mb-1"

FILE * trace; 

unsigned long last_code_page_frame_seen, last_memory_page_frame_seen;

struct timespec clock_time_when_entering_page, clock_time_when_leaving_page;

/* ............................................................................ */
/* Print Help Message */
/* ............................................................................ */

INT32 Usage ()

{  
  PIN_ERROR("This Pintool prints the IPs of every instruction address/memory address accessed\n"  
    + KNOB_BASE::StringKnobSummary() + "\n");  
  return -1;  
}

// Calculate Page Frame Number i.e which page frame in virtual address space
unsigned long CalculatePageFrameNumber (VOID *addr)
{
  unsigned long page_frame_number;
  page_frame_number = 1 + (unsigned long) addr / 4096;
  return page_frame_number;
}

unsigned long calculate_elapsed_time()
{
  unsigned long ns_elapsed;
  unsigned long seconds_elapsed;
  
  // calculate time delta
  clock_gettime(CLOCK_MONOTONIC, &clock_time_when_leaving_page);
  
  ns_elapsed = clock_time_when_leaving_page.tv_nsec - clock_time_when_entering_page.tv_nsec;
  seconds_elapsed = (clock_time_when_leaving_page.tv_sec -
                    clock_time_when_entering_page.tv_sec);
  
  ns_elapsed += (seconds_elapsed * 1000000000L);
  
  // reset clock

clock_gettime(CLOCK_MONOTONIC, &clock_time.when_entering_page);

if(ns_elapsed > 0) {
    return ns_elapsed;
}
else {
    return 0;
}

// This function is called before every instruction is executed
// and prints the IP
VOID printip(VOID *ip, int stack_access_flag)
{
    unsigned long page_frame_number=0;

    const char *image_name;
    IMG image;

    PIN_LockClient();

    image = IMG_FindByAddress(ADDRINT(ip));

#ifndef LOG_APPLICATION_INSTRUCTIONS_ONLY
    if(IMG_Valid(image)) {
        image_name = IMG_Name(image).c_str();
    }
#endif
```c
#ifdef LOG_APPLICATION_INSTRUCTIONS_ONLY
    /* if ((strstr(image_name, "simple_code") != NULL) ||
        (strstr(image_name, "libc") != NULL) ||
        (strstr(image_name, "ld-2") != NULL)
    ) {
     
if ( (strstr(image_name, "pinbin") == NULL) &&
        (strstr(image_name, "pthread") == NULL) &&
        (strstr(image_name, "librt") == NULL) &&
        (strstr(image_name, "libelf") == NULL) &&
        (strstr(image_name, "libdwarf") == NULL) &&
        (strstr(image_name, "mysql") == NULL) &&
        (strstr(image_name, "c++") == NULL) &&
        (strstr(image_name, "libdl") == NULL) &&
        (strstr(image_name, "libm") == NULL) &&
        (strstr(image_name, "libgcc") == NULL) &&
        (strstr(image_name, "simple_code") == NULL) ){
     */
      
    if ( (strstr(image_name, APPLICATION_NAME) != NULL) ) {
    
#endif
     // calculate page frame number
     page_frame_number = CalculatePageFrameNumber(ip);
     
    // check if code page has changed
    if(page_frame_number != last_code_page_frame_seen) {
     // record page frame change
     last_code_page_frame_seen = page_frame_number;
     // then log code page frame number
```
// printf(trace, 'C\%lu\t\%lu\n', last_code_page_frame_seen,
calculate_elapsed_time());
// printf('\%s\n', image_name);
//LOG('C'+decsstr(last_code_page_frame_seen)+'\t'+decsstr(
calculate_elapsed_time())+'\t'+IMG_Name(image).c_str()+'\n');

#ifdef TAG_STACK_ACCESSSES
if(stack_access_flag == 1) {
LOG('C'+decsstr(last_code_page_frame_seen)+'\t'+decsstr(
calculate_elapsed_time())+'\t'+stack_access+'
');
stack_access_flag = 0;
}
#else
LOG('C'+decsstr(last_code_page_frame_seen)+'\t'+decsstr(
calculate_elapsed_time())+'\n');
#endif
#endif
#ifdef LOG_APPLICATION_INSTRUCTIONS_ONLY
#endif
#endif
#endif

PIN_UnlockClient();
// Print a memory read record
VOID RecordMemRead(VOID *ip, VOID *addr, int stack_access_flag)
{

unsigned long code_page_frame_number=0;
unsigned long memory_page_frame_number=0;

const char *code_image_name, *data_image_name;

IMG code_image, data_image;

PIN_LockClient();

code_image = IMG_FindByAddress(ADDRINT(ip));
data_image = IMG_FindByAddress(ADDRINT(addr));

#ifdef LOG_APPLICATION_INSTRUCTIONS_ONLY
if (IMG_Valid(code_image) && IMG_Valid(data_image)) {
    code_image_name = IMG_Name(code_image).c_str();
data_image_name = IMG_Name(data_image).c_str();
}
#endif

#ifdef LOG_LIBRARY_ACCESS_TO_HEAP
    // If this is a library access to a heap
    if ((strstr(code_image_name, 'lib') != NULL) ||
    #endif
}
(strstr(data_image_name, 'lib') != NULL) || (strstr(data_image_name, 'ld-2') != NULL))

//IMG_Valid(data_image) != true

// calculate page frame numbers
memory_page_frame_number = CalculatePageFrameNumber(addr);
code_page_frame_number = CalculatePageFrameNumber(ip);
LOG('C'+decstr(last_code_page_frame_seen)+'t'+'M'+decstr(memory_page_frame_number)+'t'+decstr(calculate_elapsed_time())+'
library_data_read'+data_image_name+'
'+hexstr(ADDRINT(addr))+'
'+RTN_FindNameByAddress(ADDRINT(ip))+'
');

last_code_page_frame_seen = code_page_frame_number;
last_memory_page_frame_seen = memory_page_frame_number;
}
#endif

#ifdef LOG_APPLICATION_INSTRUCTIONS_ONLY

/* if ( (strstr(image_name, 'simple_code') != NULL) ||
   (strstr(image_name, 'libc') != NULL) ||
   (strstr(image_name, 'ld-2') != NULL)
   ) {

   if ( (strstr(image_name, 'pinbin') == NULL) &&
   (strstr(image_name, 'pthread') == NULL) &&
   (strstr(image_name, 'librt') == NULL) &&
   (strstr(image_name, 'libelf') == NULL) &&
   (strstr(image_name, 'libdwarf') == NULL)
   )
   */

   //...

#endif
(strstr(image_name, 'mysql') == NULL) &&
 (strstr(image_name, 'c++') == NULL) &&
 (strstr(image_name, 'libdl') == NULL) &&
 (strstr(image_name, 'libm') == NULL) &&
 (strstr(image_name, 'libgcc') == NULL) &&
 (strstr(image_name, 'simple_code') == NULL) &&
 (strstr(image_name, 'libc') == NULL)) {
 */
    if (strstr(code_image_name, APPLICATION_NAME) != NULL) //
        //
        // (strstr(data_image_name, 'mysql') != NULL)
    ) {
#endif

    // calculate page frame numbers
    memory_page_frame_number= CalculatePageFrameNumber(addr);
    code_page_frame_number= CalculatePageFrameNumber(ip);

#ifdef DONT_LOG_MULTIPLE_ACCESSSES_BETWEEN_THESAME_CODEPAGE_ANDDATAPAGE
    if( (memory_page_frame_number != last_memory_page_frame_seen) || (code_page_frame_number != last_code_page_frame_seen) ) {
#endif

    // log code page frame numbers
    //fprintf(trace, "C\%lu\"M\%lu\"t\%lu\n\", code_page_frame_number,
            memory_page_frame_number, calculate_elapsed_time());
    //LOG(\"C\"+decsstr(last_code_page_frame_seen)+\"t\"+\"M\"+decsstr(
            memory_page_frame_number)+\"t\"+decsstr(calculate_elapsed_time())+\"t\"+
            IMG_Name(image).c_str()+\"n\") ;
#endif

#ifdef TAG_STACK_ACCESSSES
    if(stack_access_flag == 1) {
        LOG(\"C\"+decsstr(last_code_page_frame_seen)+\"t\"+\"M\"+decsstr(}
memory_page_frame_number)+'\t'+decsr(calculate_elapsed_time())+'\t'+'
'+'stack_access]+'\t'+decsr(calculate_elapsed_time())+'\n');

stack_access_flag = 0;

else {
#endif

LOG('C'+decsr(last_code_page_frame_seen)+'\t'+M'+decsr(
memory_page_frame_number)+'\t'+decsr(calculate_elapsed_time())+'\t'+'n');

#endif

// fflush(trace);

// record page frame change
last_code_page_frame_seen = code_page_frame_number;
lst_memory_page_frame_seen = memory_page_frame_number;

#endif DONT_LOG_MULTIPLE_ACCESSSES_BETWEEN_THE SAME_CODEPAGE_AND_DATAPAGE

#endif

#endif LOG_APPLICATION_INSTRUCTIONS_ONLY

PIN_UnlockClient();

}
VOID RecordMemWrite(VOID *ip, VOID *addr, int stack_access_flag)
{
    unsigned long code_page_frame_number=0;
    unsigned long memory_page_frame_number=0;

    const char *code_image_name, *data_image_name;

    IMG code_image, data_image;

    PIN_LockClient();

    code_image = IMG_FindByAddress(ADDRINT(ip));
    data_image = IMG_FindByAddress(ADDRINT(addr));

    #ifdef LOG_APPLICATION_INSTRUCTIONS_ONLY
    if(IMG_Valid(code_image) && IMG_Valid(data_image)) {
        code_image_name = IMG_Name(code_image).c_str();
        data_image_name = IMG_Name(data_image).c_str();
    }
    #endif

    #ifdef LOG_LIBRARY_ACCESS_TO_HEAP
    // If this is a library access to a heap
    if ( ((strstr(code_image_name, 'lib')) != NULL) ||
         (strstr(code_image_name, 'ld')) != NULL )
        &&
    #endif
(strstr(data_image_name, "lib") != NULL) || (strstr(data_image_name, "ld-2") != NULL))

// &&(IMG_Valid(data_image) != true)
{
    // calculate page frame numbers
    memory_page_frame_number = CalculatePageFrameNumber(addr);
    code_page_frame_number = CalculatePageFrameNumber(ip);
    LOG("C"+decr(last_code_page_frame_seen)++"\\t"+"M"+decr(
        memory_page_frame_number)++"\\t"+decr(calculate_elapsed_time())++"\\t"+
        library_data_write++"\\t"+data_image_name++"\\t"+hexstr(ADDRINT(addr))++"\\n")
    last_code_page_frame_seen = code_page_frame_number;
    last_memory_page_frame_seen = memory_page_frame_number;
}
#endif

#define LOG_APPLICATION_INSTRUCTIONS_ONLY

/* if ((strstr(image_name, "simple_code") != NULL) ||
    (strstr(image_name, "libc") != NULL) ||
    (strstr(image_name, "ld-2") != NULL)
) {

    if (
        (strstr(image_name, "pinbin") == NULL) "
        (strstr(image_name, "pthread") == NULL) "
        (strstr(image_name, "librt") == NULL) "
        (strstr(image_name, "libelf") == NULL) "
        (strstr(image_name, "libdwarf") == NULL) "
        (strstr(image_name, "mysqld") == NULL) 
    }
(strstr(image_name, "c++") == NULL) &&
(strstr(image_name, "libdl") == NULL) &&
(strstr(image_name, "libm") == NULL) &&
(strstr(image_name, "libgcc") == NULL) &&
(strstr(image_name, "simple_code") == NULL) &&
(strstr(image_name, "libc") == NULL)) {
  
  /*
   *  
   *   if ( (strstr(code_image_name, APPLICATION_NAME) != NULL)
   */
   
   // (strstr(data_image_name, "mysql") != NULL)
   
   } 

#endif

// calculate page frame numbers
memory_page_frame_number= CalculatePageFrameNumber(addr);
code_page_frame_number= CalculatePageFrameNumber(ip);

#ifdef DONT_LOG_MULTIPLE_ACCESSSES_BETWEEN_THESAME_CODEPAGEANDDATAPAGE

  if( (memory_page_frame_number != last_memory_page_frame_seen) ||
      (code_page_frame_number != last_code_page_frame_seen) ) {
#endif

// log page frame numbers
// fprintf(trace, "C%lu\tM%lu\t%lu\n", code_page_frame_number, 
                memory_page_frame_number, calculate_elapsed_time());

//LOG("C"+destr(last_code_page_frame_seen)+"M"+destr(
   memory_page_frame_number)+"t"+destr(calculate_elapsed_time())+"t"+
   IMG_Name(image).c_str()+"n");

#endif //LOG_STACK_ACCESSSES

  if(stack_access_flag == 1) {

LOG("C"+destr(last_code_page_frame_seen)+"M"+destr(
       memory_page_frame_number)+"t"+destr(calculate_elapsed_time())+"");

}
t'+'stack_access''+'n'');  
    stack_access_flag = 0;
} else {
#ifdef 
    LOG('C'+decsr(last_code_page_frame_seen)+'t'M'+decsr(memory_page_frame_number)+'t'+decsr(calculate_elapsed_time())+''
#endif

#ifdef TAG_STACK_ACCESSSES 
}
#endif

// fflush(trace);
// record page frame change
last_code_page_frame_seen = code_page_frame_number;
last_memory_page_frame_seen = memory_page_frame_number;

#ifdef DONT_LOG_MULTIPLE_ACCESSES_BETWEEN_THESAME_CODEPAGE_AND_DATAPAGE 
}
#endif

#ifdef LOG_APPLICATION_INSTRUCTIONS_ONLY 
}
#endif

PIN_UnlockClient();
}
// Is called for every instruction and instruments reads and writes
VOID Instruction(INS ins, VOID *v)
{
// Instruments memory accesses using a predicated call, i.e.
// the instrumentation is called iff the instruction will actually be executed.

// The IA-64 architecture has explicitly predicated instructions.
// On the IA-32 and Intel(R) 64 architectures conditional moves and REP
// prefixed instructions appear as predicated instructions in Pin.

int stack_access_flag = 0;

#ifdef TAG_STACK_ACCESS
if (INS_IsStackRead(ins) || INS_IsStackWrite(ins)) {
    stack_access_flag = 1;
}
#endif

UINT32 memOperands = INS_MemoryOperandCount(ins);

/* If other type of instruction i.e no of memory operands == Zero */
if (memOperands == 0) {
    // Insert a call to printip before every instruction, and pass it the IP
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip, IARG_INST_PTR, IARG_END,
                   stack_access_flag);
}

// Iterate over each memory operand of the instruction.
for (UINT32 memOp = 0; memOp < memOperands; memOp++)
{
    if (INS_MemoryOperandIsRead(ins, memOp))
    {

INS_InsertPredicatedCall(
    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
    IARG_INST_PTR,
    IARG_MEMORYOP_EA, memOp,
    IARG_END, stack_access_flag);
}

// Note that in some architectures a single memory operand can be
// both read and written (for instance incl (%eax) on IA-32)
// In that case we instrument it once for read and once for write.
if (INS_MemoryOperandIsWritten(ins, memOp))
{
    INS_InsertPredicatedCall(
        ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
        IARG_INST_PTR,
        IARG_MEMORYOP_EA, memOp,
        IARG_END, stack_access_flag);
}

#define REMOVE_STACK_ACCESS

//
} /* -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-*/

VOID Fini(INT32 code, VOID *v)
{
    //fprintf(trace, "\n'\n');
    //fflush(trace);
    //fclose(trace);
}

/* -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-*/
int main(int argc, char *argv[])
{

    last_code_page_frame_seen = 0;
    last_memory_page_frame_seen = 0;

    // start clock
    clock_gettime(CLOCK_MONOTONIC, &clock_time.when_entering_page);

    if (PIN_Init(argc, argv)) return Usage();
    // to access routines by name
    PIN_InitSymbols();

    //trace = fopen("page_trace.out", "w");
    // trans_jitter = fopen("trans_jitter.out", "w");

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}