PortageLive: Delivering Machine Translation Technology via Virtualization

*Patrick Paul, *Samuel Larkin, •Ulrich Germann, *Eric Joanis, *Roland Kuhn

*National Research Council Canada
{First Name}.{Last Name}@nrc-cnrc.gc.ca

•University of Toronto and
National Research Council Canada
germann@cs.toronto.edu

Abstract

We describe and report initial results on using virtual machines as a vehicle to deploy machine translation technology to the marketplace. Virtual machines can bridge the gap between the computing infrastructure typically used in research environments and commodity PCs typical of office environments. A key component is the compact representation of the underlying databases and models in *tightly packed tries*, which allows us to run state-of-the art translation technology on regular office PCs.

1 Introduction

The transfer of information technologies from the research lab to the end user has traditionally been an expensive and tedious process. In addition to the need to ensure robustness and reliability of the underlying technology, and the effort necessary to integrate it into the end user's work flow, it often also involves porting software implementations from one platform to another. Research prototypes are frequently implemented on Unix or Linux platforms, whereas corporate IT infrastructures often rely primarily on Windows.

The considerable cost of either porting software from one platform to another, or purchasing and maintaining the hard- and software required to run new technology in its 'native' format constitutes a considerable obstacle to fast deployment of cutting-edge research results to the marketplace. It also makes it expensive for potential customers to assess the value that such new technology could add to their business processes.

In this paper, we argue and demonstrate that *virtualization* is a viable and cost-effective way of transferring machine translation technology from the research lab to the real world. In the next section, we first briefly describe the *Portage* machine translation system developed at the National Research Council Canada. Section 3 introduces the concept of virtualization. In Section 4 we discuss the design and use of *PortageLive*, an instantiation of *Portage* as a virtual machine. We tested *PortageLive* in a number of scenarios, from small, compact machines that could run on a laptop to parallelizations on computer networks composed of workstations that are typical of office environments. The results of these experiments are reported in Section 5. Section 6 concludes the paper.

2 Portage

Portage (Sadat *et al.*, 2005) is a state-of-the-art system for phrase-based statistical machine translation (Koehn *et al.*, 2003). The advantage of statistical approaches to machine translation is that they require much less human labor than language-specific systems carefully designed and constructed by human experts in particular languages or language pairs. Instead of human expertise, the system relies on existing collections of translations to learn how to translate. Under active development since 2004, *Portage* has been employed to translate between a wide variety of language pairs, involving such languages as Arabic, Chinese, English, Finnish, French, German, and Spanish. Special research attention has been given to translation from English into French and vice versa, and from Chinese to English.

In the training phase, *Portage* learns from large

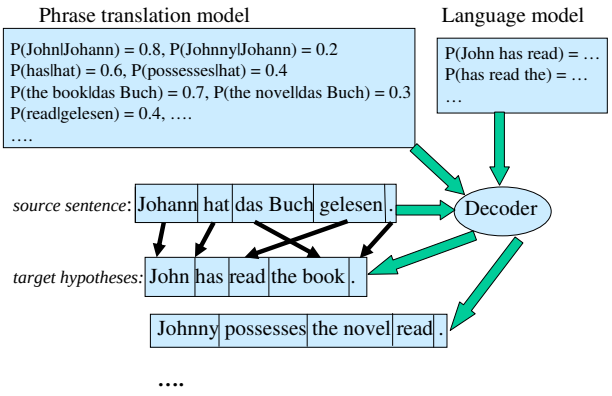


Figure 1: Phrase-based statistical machine translation with *Portage*

collections of bilingual text how short contiguous sequences of words (*"phrases"*) in one language (*source*) are most often translated into phrases in the other language (*target*). Pairs of phrases and their translations as observed in the training data are stored in a probabilistic dictionary of phrase-level translations, the *phrase table*. During actual translation (*"decoding"*), the input is segmented into contiguous groups of words for which translations are listed in the phrase table. The corresponding translation fragments are then concatenated to form the translation of the entire input sentence. Since most phrases have more than one likely translation, the decoder uses probabilistic models of phrase translation, phrase reordering and target language fluency to rank partial translation candidates during decoding. Unpromising candidates are dropped; the other ones are explored further. At the end, the decoder outputs the best-scoring "full" hypothesis, i.e., a translation covering the entire input sentence. Optionally, the system can produce "n*-best lists*" of the n best translations found during the docoding process. Figure 1 illustrates the process.

3 Virtualization

In the context of computing, *virtualization* (Popek and Goldberg, 1974) is the abstraction of computer resources. In the realm of personal computers, the concept is most prominent in the form of virtual memory: the operating system (OS) pretends to have more main memory than it actually has. The *virtual memory manager* maintains an index of virtual memory pages[1] and stores on hard disk those pages that contain data but are currently not in active use. When such memory pages are eventually accessed, they are swapped into memory against others that are written to disk or simply replaced, if nothing has changed since the last time they were read. Like a good valet parking service, a virtual memory manager manages a precious resource (fast but expensive main memory) in a way that ideally gives a large number of customers the convenience of having (almost) immediate access to this resource.

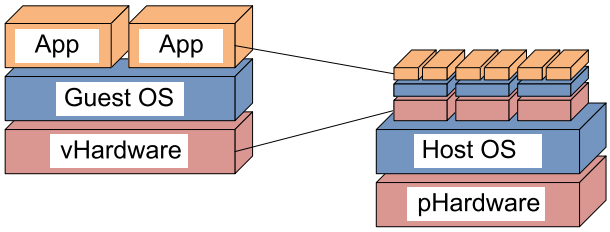


Figure 2: Virtual machines (*guests*) running as applications on a physical *host* machine.

Virtual machine monitors (VMMs) go one step farther. Running as an application on a *host* OS, they emulate an entire computer running an instance of an embedded *guest* OS, translating hardware requests from the guest OS into system calls to the host OS. They can also translate outside requests into calls to the embedded host system, thus creating a virtual node on a computer network. Since every virtual machine is simply an application instance from the host OS's point of view, multiple nodes can be emulated on a single physical computer, as shown in Figure 2.

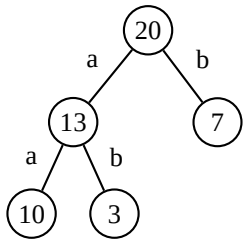
Virtualization has received increased attention in recent years as a way of reducing the hardware cost and the ecological impact of large computing facilities. Since not every host is 100% busy at all times, it is more efficient to have multiple hosts share physical hardware resources.

Apart from technologically advanced virtualization solutions aimed at optimizing resource use of IT infrastructures from large multi-user business servers to enterprise computing centers, there are also lightweight versions for single desktop computers. There are two typical uses for virtual machines

[1] A memory page is a contiguous block of memory, often about 4KB in size.

total count	20
a	13
aa	10
ab	3
b	7

(a) Count table



(b) Trie representation

0	13	*offset of root node*
1	10	*node value of* ‘aa’
2	0	*size of index to child nodes of* ‘aa’ in bytes
3	3	*node value of* ‘ab’
4	0	*size of index to child nodes of* ‘ab’ in bytes
5	13	*node value of* ‘a’
6	4	*size of index to child nodes of ‘a’ in bytes*
7	a	*index key for* ‘aa’ *coming from* ‘a’
8	4	*relative offset of node* ‘aa’ $(5 - 4 = 1)$
9	b	*index key for* ‘ab’ *coming from* ‘a’
10	2	*relative offset of node* ‘ab’ $(5 - 2 = 3)$
11	7	*node value of* ‘b’
12	0	*size of index to child nodes of* ‘b’ in bytes
13	20	*root node value*
14	4	*size of index to child nodes of root in bytes*
15	a	*index key for* ‘a’ *coming from root*
16	8	*relative offset of node* ‘a’ $(13 - 8 = 5)$
17	b	*index key for* ‘b’ *coming from root*
18	2	*relative offset of node* ‘b’ $(13 - 2 = 11)$

(c) Trie representation in a contiguous byte array. In practice, each field may vary in length.

Figure 3: A count table (a) stored in a trie structure (b) and the trie's sequential representation in a file (c). As the size of the count table increases, the trie-based storage becomes more efficient, provided that the keys have common prefixes (from Germann *et al.* (2009)).

on desktop computers: first, to provide a sandbox, a safe testbed for software development that allows a machine to crash without crashing the actual computer or interfering with the outside world. And second, to run two operating systems concurrently on a single computer. Since the physical host computer and the guest running as a virtual machine can communicate via networking protocols, it is possible, for example, to emulate a Linux server on a PC running Windows. Virtualization technology thus allows deployment of software developed on one OS on another without the need to port software across operating systems.

The benefits of virtualization of course come at a price: the additional layer of abstraction also adds a layer of indirection, which can result in a loss of performance compared to a native system.

4 Virtualizing *Portage*

4.1 Compact, fast-loading models

Statistical machine translation is resource-intensive in many respects. The decoder needs fast access to language models and phrase tables, and large search graphs have to be stored in memory to keep track of partial translation hypotheses. To ensure fast access to the underlying databases (i.e., language models and translation tables), it is highly desirable to keep them in main memory; disk access and even network access to models distributed over a network of computers are several orders of magnitude slower than accessing a computer's main memory. Especially language models, which are queried millions of time during the translation of a single sentence, should be kept in memory wherever possible. While virtual memory allows us to go beyond the limitations of physical memory occasionally, the cost of memory swapping is so high that it is worth spending some effort on reducing the memory footprint of model storage.

Both language models and phrase tables associate information with token sequences that have a considerable amount of overlap (i.e., common subsequences). A *trie* (Fredkin, 1960), or *prefix tree* is a well-known data structure for storing such collections of sequences with common prefixes. Each sequence is represented by a single node in a tree with labeled arcs; each path from the root node to a node in the tree spells out the respective token sequence, as shown in Figures 3a and 3b. Thus, prefixes com-

mon to multiple sequences in the collection need to be stored only once. Retrieval of information is linear in the length of the key.

Tries can be represented in linear byte sequences (e.g, a file) as shown in Figure 3c. Relative offsets within the byte string are used to indicate links between parent and child nodes.

In our implementation of tries (Germann *et al.*, 2009), we save space by using symbol-level compression. Token labels are represented by a unique integer IDs, which are assigned in decreasing order of token frequency: the more frequent a token is, the lower its ID. Token IDs and file offsets are stored in a variable-width encoding scheme that represents each number in base-128 notation. The full range of base-128 digits (0–127) can be represented in 7 bits; the eighth bit in each byte is used to indicate whether or not more digits need to be read.

We call this technique *tight packing*. The space savings are considerable. Language models encoded as tightly packed tries consume less space than language models in ARPA text format that have been compressed with the *gzip* utility, yet still provide direct access in time linear in the length of the key. More importantly, they have been shown to require only about a quarter of the physical memory needed, for example, by the SRI language modeling tookit (Stolcke, 2002) for representing identical language models. Unlike in-memory implementations like the SRI toolkit, a tightly packed language model does not even need to be fully loaded into memory. In practice, actual memory use can be as little as 10-15% of the physical memory needed by the SRI toolkit (Germann *et al.*, 2009).

We achieve this by using the *memory mapping* mechanism provided by most modern operating systems. A file on disk permanently storing the trie structure is mapped directly into a region of virtual memory. When an application accesses that region of virtual memory, the virtual memory manager transparently takes care of loading the respective data pages from disk if necessary. This approach has several advantages. First, by using a representation that can be mapped directly into memory, we do not need to spend any time rebuilding the structure at load time or copying data from the OS-level file cache (which buffers the data read from and written to disk in order to provide faster access). Second, since we are using a read-only structure, the virtual memory manager can simply drop pages from physical memory if memory swapping is inevitable. Otherwise, all changes to the data structures stored in memory would have to be written to disk when memory swapping takes place (although that might happen in the background, only putting additional load on the disk without an immediate effect on average run-time). Third, only those data pages that are in fact being accessed during translation are actually read from disk. And fourth, the initialization delay due to model loading between starting the decoder and being ready for translation is almost entirely eliminated. With in-memory implementations of the underlying models, the *Portage* decoder has an initialization lag of about one to two minutes. With tightly packed tries, the decoder is ready to translate after about two seconds. As relevant pages are loaded into memory, translation is slower at first, but almost achieves the same throughput once the relevant pages are available in physical memory.

4.2 Protecting the host against excessive resource demands of the translation engine

Since translation is such a resource-intensive task, implementing translation engines as VMs does not only help bridge the gap between different operating systems but also allows us to shield the host system from excessive resource demands of the translation engine running on a guest VM. This is particularly relevant in a scenario where computing resources need to be shared between time-critical applications (e.g., interactive use of office software) and a translation process that is possibly running in the background. Without virtualization, the operating system has two choices in responding to resource demands from an application: granting them (which may lead to heavy memory swapping if there is not enough physical memory to accommodate all the virtual memory requested), or denying them, which typically means that the requesting application crashes or aborts. Virtualization allows us to put a wall around an application: we can limit the maximum amount of physical memory available to the application (or more precisely: the amount of 'physical' memory in the virtual machine on which that application runs) without forcing the application to abort if it needs more memory than it has been alot-

Table 1: Corpus statistics for the training corpus used in the experiments

	Fr	En
data source	*Canadian Hansard*	
running words	113,539,694	101,274,018
vocabulary size	211,053	205,221
sentence pairs	5,239,985	*same*
phrase table		
# of phrase pairs	57,155,768	*same file*
file size		
tightly packed	975 MB	*same file*
.gz	1.6 GB	*same file*
Language model:		
unigrams	211,055	—
bigrams	4,045,364	—
trigrams	5,871,066	—
4-grams	8,689,908	—
5-grams	9,712,388	—
file size		
tightly packed	274 MB	—
ARPA.gz	290 MB	—

Table 2: Translation times on a single VM and with 5-way parallelization on 5 VMs running on 5 distinct hosts.

		1st run		best run	
VM/#[a]	**RAM**	**sec.[b]**	**w/s[c]**	**sec.[b]**	**w/s[c]**
1 / 1	512 MB	2,917	7.3	2,917	7.2
1 / 1	1 GB	457	46.7	112	189.6
1 / 1	2 GB	390	54.6	105	202.3
5 / 1	512 MB	1,066	20.0	886	24.0
5 / 1	1 GB	285	74.7	55	388.0
5 / 2	1 GB	224	95.2	59	360.4
5 / 1	2 GB	141	151.0	32	667.7
5 / 2	2 GB	92	231.7	22	959.8
5 / 4	2 GB	172	123.8	24	905.2

[a] # of VMs (on different hosts) / decoder instances per VM
[b] End-to-end time for translation of 21,290 words (En to Fr)
[c] Words per second

Host machines: Dell Optiplex 755 w/ 3.25 GB of RAM; Intel Core 2 DUO E8500 processor 3.16GHz; OS: Windows XP Professional SP3. Virtualization software: VMware Server version 2 emulating a machine with 1 or 2 CPUs. Guest OS: CentOS 5.2.

ted. In this case, the guest VM running the application will start swapping memory internally, with only marginal effects (increased disk load) on applications running outside the virtual machine.

4.3 Accessing the translation engine

Virtual machines can be accessed via an IP address like other machines on a local network. This allows us to use standard communication protocols such as TCP/IP, SSH, HTTP or the SOAP framework to communicate with the translation engine. In the experiments reported in the next section, we used an SSH connection to send the test input through a translation pipe.

5 Experiments

We tested *PortageLive* in a variety of scenarios to determine the trade-offs between resource allocation and translation speed. The top part of Table 2 shows translation speeds for English-to-French translation on VM machines with 512, 1024 and 2048 MB of memory running on a 32-bit host under Windows XP Professional, using *VMware Server*.[2] The statistics for the corpus underlying the system are shown in Table 1. Due the hard limit of 4 GB on the size of virtual memory on 32-bit machines, we were not able to run very large systems in these experiments, such as our Chinese-English system, which requires more than 4 GB just for the data tables, not to mention the memory required to explore the translation hypothesis space. Nevertheless, our English-to-French system is by no means small — the training corpus is about five times as large as the French-English subset of the popular Europarl corpus (Koehn, 2005).

In our experiments, we translated a test corpus of 21,290 words of running English text repeatedly to gauge the difference in performance between a freshly started system and a system that has been running for a long time, so that all data can be assumed to be cached in the OS's file cache, memory permitting. If sufficient memory is available to keep all models in memory (2GB for this particular translation engine), the translation engine speeds up over time. At first it is sluggish, because model data needs to be transferred from disk into memory. Later on, translation is much faster, because most

[2] `http://www.vmware.com/products/server/`

of the (virtual) memory pages needed for translation are already in physical memory.

In this particular configuration of an English-to-French translation system, translation speed suffers dramatically when less than 1 GB is available to the translation engine. Notice that there are practically no caching effects on repeated runs: virtual memory pages are constantly being swapped in and out of memory. At 1 GB, there is a noticeable performance loss compared to a VM with 2 GB, but most of the time is still spent translating, not swapping memory.

What do these numbers mean in practice? The most important result, we think, is that this experiment suggests that it is feasible to run an SMT engine on a single state-of-the-art desktop or laptop computer concurrently with other applications, including interactive use (email, word processing, etc). Since the virtual host can be accessed like another machine on a local network through a number of networking protocols (cf. Section 4.3), it is possible to integrate translation capabilities easily into applications. Instead of sending a request on another host on the Internet or Intranet, the request is sent to the virtual machine hosted on the same computer. This is an attractive option for providing machine translation in situations where no network connections are available.

Another possible application is to use computers in an existing office network for distributed translation processing. When the burden of translation is distributed over many shoulders, even inefficient "small" VMs can collectively achieve satisfactory overall translation speeds. We can think of two scenarios. One is to have many small translation servers running permanently as virtual machines on a number of physical computers (including desktop computers used for regular office work) on a Local Area Network. A central translation management server pushes small translation tasks to the servers and gathers the results. The other one is a scenario where VMs are started when a particular physical machine is idle, e.g. in lieu of a screen saver. The VM then pulls a translation request from a central server, processes it and returns it to the central server, which collects the results.[3] The first scenario offers better predictability of the translation volume that can be handled; the second scenario can be used to better utilize existing machines during idle periods without interfering with ongoing tasks when they are in use.

We simulated the first scenario by running 5 virtual machines on identical desktop computers that also serve as our office desktops computers. The results are shown in the bottom part of Table 2. The first thing to observe is that even though translations were run simultaneously on 5 machines, we generally do not achieve 5 times the translation speed. This is again due to the overhead of having to load data gradually from disk into memory. Noteworthy is the good performance of running two decoders in parallel on an emulated 2-CPU machine with 2 GB or RAM. What is happening here? Due to memory mapping, the two processes share the memory space that contains the models, so that one decoder benefits from earlier page lookups of the other, and vice versa. Interestingly, if we run four decoders in parallel on a single 2-CPU VM, the performance suffers. We assume that this is due to race conditions, especially with respect to disk access.

6 Conclusion

The goal of our experiments was to evaluate the feasibility of using virtualization to aid the technology transfer from the research lab to users of machine translation. Our experiments have shown that this is a viable option. A key component is the use of compact representations of the underlying data bases, as described in Section 4.1, which allow us to fit all our models in the 4 GB memory limit of 32-bit machines.

Virtualization allows fast transfer of new technology to existing and potential customers and end users without significant investments in porting software between operating systems, or in hard- and software to run the technology in its 'native' environment. This is a particularly attractive option during exploratory stages of technology assessment in a business environment. Moreover, it allows easy deployment of complex software installations. From the user's perspective, a virtual machine comes as a single file that is 'played' by the virtualization software.

[3] This strategy is well-known from public-benefit grid computing projects such as *BOINC* (`http://boinc.berkeley.edu/`).

References

Fredkin, E. 1960. “Trie memory.” *Communications of the ACM*, 3(9):490–499.

Germann, U., S. Larkin, and E. Joanis. 2009. “Tightly packed tries: How to fit large models into memory, and make them load fast, too.” *Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*. Boulder, CO, USA.

Koehn, P. 2005. “European parliament proceedings parallel corpus.” *MT Summit X*. Phuket, Thailand.

Koehn, P., F. J. Och, and D. Marcu. 2003. “Statistical phrase-based translation.” *Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL ’03)*, 48–54. Edmonton, AB, Canada.

Popek, G. J. and R. P. Goldberg. 1974. “Formal requirements for virtualizable third generation architectures.” *Commun. ACM*, 17(7):412–421.

Sadat, F., H. Johnson, A. Agbago, G. Foster, R. Kuhn, J. Martin, and A. Tikuisis. 2005. “PORTAGE: A phrase-based machine translation system.” *ACL Workshop on Building and Using Parallel Texts*, 133–136. Ann Arbor, MI, USA. Also available as NRC-IIT publication NRC-48525.

Stolcke, A. 2002. “SRILM — an extensible language modeling toolkit.” *International Conference on Spoken Language Processing*. Denver, CO, USA.