# Wonderful : A Terrific Application and Fascinating Paper

Your N. Here
*Your Institution*

Second Name
*Second Institution*

## Abstract

Your Abstract Text Goes Here. Just a few facts. Whet our appetites.
TeX into TeXnicians and TeXperts.
http://www.scu.edu/˜bush
http://www.pku.edu/∼bush
mother-in-law
pages 1–12
yes —or no
0, 1 and $-1$
$1-2$
$\Delta$

## 1 Introduction

A paragraph of text goes here. Lots of text. Plenty of interesting text.

More fascinating text. Features[1] galore, plethora of promises.

## 2 This is Another Section

Some embedded literal typset code might look like the following :

```
int wrap_fact(ClientData clientData,
              Tcl_Interp *interp,
              int argc, char *argv[]) {
    int result;
    int arg0;
    if (argc != 2) {
        interp->result = "wrong # args";
        return TCL_ERROR;
    }
    arg0 = atoi(argv[1]);
    result = fact(arg0);
    sprintf(interp->result,"%d",result);
    return TCL_OK;
}
```

Figure 1: Wonderful Flowchart

Now we're going to cite somebody. Watch for the cite tag. Here it comes [**?**, **?**]. The tilde character (˜) in the source means a non-breaking space. This way, your reference will always be attached to the word that preceded it, instead of going to the next line.

## 3 This Section has SubSections

### 3.1 First SubSection

Here's a typical figure reference. The figure is centered at the top of the column. It's scaled. It's explicitly placed. You'll have to tweak the numbers to get what you want.

This text came after the figure, so we'll casually refer to Figure 1 as we go on our merry way.

## 3.2 New Subsection

It can get tricky typesetting Tcl and C code in LaTeX because they share a lot of mystical feelings about certain magic characters. You will have to do a lot of escaping to typeset curly braces and percent signs, for example, like this: “The `%module` directive sets the name of the initialization function. This is optional, but is recommended if building a Tcl 7.5 module. Everything inside the `%{, %}` block is copied directly into the output. allowing the inclusion of header files and additional C code.”

Sometimes you want to really call attention to a piece of text. You can center it in the column like this:

```
_1008e614_Vector_p
```

and people will really notice it.

The noindent at the start of this paragraph makes it clear that it’s a continuation of the preceding text, not a new para in its own right.

Now this is an ingenious way to get a forced space. `Real *` and `double *` are equivalent.

Now here is another way to call attention to a line of code, but instead of centering it, we noindent and bold it.

```
size_t :  fread ptr size nobj stream
```

And here we have made an indented para like a definition tag (dt) in HTML. You don’t need a surrounding list macro pair.

> `fread` reads from `stream` into the array `ptr` at most `nobj` objects of size `size`. `fread` returns the number of objects read.

This concludes the definitions tag.

## 3.3 How to Build Your Paper

You have to run `latex` once to prepare your references for munging. Then run `bibtex` to build your bibliography metadata. Then run `latex` twice to ensure all references have been resolved. If your source file is called `usenixTemplate.tex` and your `bibtex` file is called `usenixTemplate.bib`, here’s what you do:

```
latex usenixTemplate
bibtex usenixTemplate
latex usenixTemplate
latex usenixTemplate
```

## 3.4 Last SubSection

Well, it’s getting boring isn’t it. This is the last subsection before we wrap it up.

## 4 Evaluation

We have implemented the proposed novel algorithm demonstrated in previous sections. The implementation adds 350 SLoC to Linux kernel and 166 SLoc to Xen hypervisor while 2 SLoc in the hypervisor are modified, which aims to build a cache pool for guest page tables so as to avoid unnecessary IOTLB flushes.

This section evaluates the performance of our algorithm by running both micro- and macro-benchmark kits.

## 4.1 Experimental Setup

Our experimental platform is a LENOVO QiTianM4390 PC with Intel Core i5-3470 running at 3.20 GHz, four CPU cores available to the system. We enable VT-d feature in the BIOS menu, which supports page-selective invalidation and queue-based invalidation interface. Xen version 4.2.1 is used as the hypervisor while domain0 uses the Ubuntu version 12.04 and kernel version 3.2.0-rc1. In addition, domain 0 as the testing system configures its grub.conf to turn on I/O address translation for itself and to print log information to a serial port in debug mode.

Besides that, we have been aware that creating/terminating a process will give rise to many page table updates (e.g., from a page of Writable to a page of Page Global Directory ), upon which function iotlb_flush_qi() will be invoked to flush corresponding IOTLB entries, and this is how a process-related operation affects IOTLB-flush. Thus, a global counter is placed into the function body to log invocation times of the function and then an average counter per minute is calculated which is called a frequency of IOTLB-flush. When the logged average counter drops to zero, it means that IOTLB does not be flushed any more, indicating that no process is created or terminated then.

Because of that, we define two different settings, classified by the frequency of IOTLB-flush.

Idle-setting: Actually, when system boots up and logins into graphical desktop, lots of system processes are created, causing many IOTLB flushes. But as time goes by, the frequency of IOTLB-flush reduces rapidly and stays stable to zero level ten minutes later, shown in figure xxx, and we think that system starts to be in an idle setting, where no process creation/termination occurs and existing system daemons are still maintained.

Busy-setting: We launch a stress tool emulating an update-intensive workload to transfer the system from an idle setting to a busy one. Specifically, the tool is busy periodically launching a default browser(e.g., Mozilla Firefox 31.0 in the experiment), opening new tabs one by one and then closing the browser gracefully in an infinite loop, so as to constantly create/terminate a large number of Firefox processes, thus giving rise to frequent updates of page tables. More precisely, one iteration of the loop costs five minutes and thus the frequency of process creation/termination are xxx per minute and xxx per minute, respectively. Besides that, memory usage on an average iteration of the loop is xxx MB. Since the frequency of IOTLB-flush will become in a stable level five minutes after the tool starts to run, execution time length of one iteration is also set to the time interval.

Since page tables do not update in the idle setting, our algorithm can not play a big role in system performance. Both micro- and macro-benchmark kits are performed under the busy setting, in which micro-tests are utilized to evaluate the frequency of IOTLB-flush, CPU usage and memory usage while macro-benchmarks give an assessment on overall system performance.

## 4.2 Micro-Benchmarks

To begin with, micro-experiments are conducted in two groups. In one group called cache-disabled, the "idle" system enters into the busy setting without the cache pool enabled. On the contrary, system state changes in another cache-pre-enabled group where the tool is invoked when the cache is already enabled since system begins to run.

As can be seen from figure xxx. Y-axis represents the frequency of IOTLB-flush, corresponding to the time interval (i.e., one minute) of x-axis for the first thirty minutes that the running tool has taken up. From this figure, frequency in the cache-disabled group increases rapidly and remains stable five minutes later. By contrast, frequency in the cache-pre-enabled group drops to zero in a very short time and keeps zero level from then on. It can be safely concluded that our proposed algorithm does have a positive effect on reducing IOTLB frequency to zero quickly.

Now lets move to CPU usage that each group will take up. Specifically, each level of page table has its allocation functions and free functions, e.g., pgd_alloc() and pgd_free() and the execution time that every related function is calculated per minute. As a result, in figure xxx, allocation and free functions in three levels of page tables in the cache-pre-enabled group consumes 30% less and xxx less CPU time in nanoseconds, respectively, compared with that of the cache-disabled group, indicating that a process interacting with the pool has an advantage in saving time over one interacting with the buddy system.

Besides CPU usage, the algorithm is evaluated in the aspect of memory usage since three levels of cache pools have been built to support a fast process creation/termination. Cache pools in the cache-pre-enabled group from figure xxx takes up 250 pages(i.e., (1000K $= 250 * 4K$) $<$ 1M) at most in the long time run, only xxx percentage of the tool's consumption, which is an insignificant usage and reaches a satisfying tradeoff between CPU time costs and space size.

But what if the memory percentage is too high? it is necessary to free pages from the cache pool to the buddy system. Pages in pool will be freed if 1) a proportion between pages in use and in pool, and 2) a total number of pages in use and in pool are greater. And data from group of cache-pre-enabled by default is referred to quantify the proportion and the total number. Actually, users can modify the two factors to adjust the cache pool size through an interface. On top of that, page number beyond the proportion is freed, stated in an equation below: $\Delta$num_to_free = num_in_pool − num_in_use.

Since pre-enabling the cache is not flexible enough, we also provide another interface for users to activate the cache mechanism in an on-demand way. For instance, system has been in a busy setting for a while and then cache is enabled manually. Users may make use of this feature to better improve system performance dynamically.

Next, in a group of cache-dynamic-enabled, cache is enabled when IOTLB-flush becomes stable while the freeing mechanism is added so as to check if this group behaves like the cache-pre-enabled group, i.e., cache-dynamic-enabled group could achieve a stable and low enough level in certain aspects, namely, frequency of IOTLB-flush, CPU usage as well as memory size, and it reaches to the level quickly.

From figures xxx to xxx, frequency of IOTLB-flush in this group has a very similar trend with that of the cache-disabled group in the first five minutes, but reduces to zero quickly and stays stable, much like that of the cache-pre-enabled group. And so does CPU usage. In addition, memory size that the cache-dynamic-enabled group takes up is only 210 pages at most, also consuming a small percentage. And it is reasonable that the cache-dynamic-enabled group has less pages in the pool since a certain amount of page tables has been freed to the buddy system before the cache pool is put to use.

Also note that the expected results in group of cache-dynamic-enabled is obtained after sever trials by slightly modifying default values of the proportion and the total number. Since these two factors heavily relies on a specific scenario, they may not work for all. How to decide the factors is further discussed in future work.

## 4.3 Macro-Benchmarks

Different micro tests have shown optimizations in three aspects for the algorithm while macro-benchmarks are made use of to evaluate its effects on overall system performance. Since cache-disabled group does not apply to real case, macro tests are conducted in two groups, i.e., cache-disabled group and cache-dynamic-enabled group.

SPECint_2006v1.2 has 12 benchmarks in total and they are all invoked with EXAMPLE-linux64-ia32-gcc43+.cfg for integer computation, results of which produce figures xxx to xxx. Lets use 400.perlbench as an example to compare the two groups. The amount of time in seconds that the benchmark takes to run in cache-dynamic-enabled group is from xxx to xxx, and the ration range is from xxx to xxx, both of which are within ranges of perlbench in cache-disabled group. And this illustration also works for the rest 11 benchmarks. As a result, little difference between the groups exists, which indicates that the algorithm does not have any bad effect on system performance.

Lmbench is used to measure CPU time that process-related system commands cost (i.e., fork+exit, fork+execve, fork+/bin/sh -c), shown in figures xxx to xxx. The configuration parameters are selected by default, except parameters of processor MHz, a range of memory and mail result, since CPU mhz of our test machine is 3.2 GHz rather than the default one, memory range uses 1024 MB to save time that Lmbench-run costs and we need no results mailed. As can be seen from the figures, command of fork+exit in cache-dynamic-enabled group costs xxx in microseconds, xxx% less than that of the cache-disabled group. and other two commands also perform well. Undoubtedly, the algorithm has reduced CPU cycles.

As for I/O performance, we use netperf to evaluate the performance of network-intensive workloads. To overcome the adverse effect caused by real network jitter, we physically connect the testing machine directly to a tester machine by a network cable, and then the tester machine as a client measures a network throughput by sending a bulk of TCP packets to the testing machine being a server. Specifically, the client connects to the tested server by building a single TCP connection, test type of which is TCP_STREAM, and time length of which lasts 60 seconds. On top of that, the TCP_STREAM test of netperf is conducted for 30 times to obtain an average value of throughput. As can be seen from figures xxx to xxx, the throughput range in cache-dynamic-enabled group is within that of the cache-disabled group. Intuitively, the results indicate that the algorithm has no contribution to the performance improvement, seemingly contradicting with the results from micro experiments.

Actually, Nadav Amit [xxx] demonstrates that the virtual I/O memory map and unmap operations consume more CPU cycles than that of the corresponding DMA transaction so that the IOTLB has not been observed to be a bottleneck under regular circumstances. Thus, only when the cost of frequent mapping and unmapping of IOMMU buffers is sufficiently reduced, the guest physical address resolution mechanism becomes the main bottleneck. Furthermore, he proposes the so-called pseudo pass-through mode and utilizes a high-speed I/O device (i.e., Intels I/O Acceleration Technology) to reduce time required by DMA map and unmap operations so that IOTLB becomes the dominant factor. As a result, it is quite reasonable that netperf results with/without the algorithm are almost the same.

## 5 Acknowledgments

A polite author always includes acknowledgments. Thank everyone, especially those who funded the work.

## 6 Availability

It's great when this section says that MyWonderfulApp is free software, available via anonymous FTP from

`ftp.site.dom/pub/myname/Wonderful`

Also, it's even greater when you can write that information is also available on the Wonderful homepage at

`http://www.site.dom/~myname/SWIG`

Now we get serious and fill in those references. Remember you will have to run latex twice on the document in order to resolve those cite tags you met earlier. This is where they get resolved. We've preserved some real ones in addition to the template-speak. After the bibliography you are DONE.

## Notes

[1]Remember to use endnotes, not footnotes!