Automating Process and Workload Pathology Detection

Ron Kaminski
Safeway Inc.

There is no perfect code, and the chance for unintended and resource-intensive misbehavior increases with complexity. We call these miscreants "pathological processes". Pathological processes are present on an astonishing number of systems and they account for a significant percentage of the user-perceived problems that generate requests for performance investigations. The combination of shrinking IS budgets, lowering (or non-existent) capacity planning head count, increasing machine count and ever-increasing software complexity means that while a well trained, experienced analyst could discover these problems, it is impossible to examine every machine in depth on a regular basis. The good news is that many of the most common pathologies have resource consumption signatures that can be automatically recognized. Once pathologies are found, modern e-mail, problem ticketing and web interfaces can speedily notify the code owners who can address the issue, often before they or their end users notice a performance impact. This paper presents examples of pathologies we have successfully detected, provides formulas we use to find them, offers some programming hints that we picked up the hard way and ends with a challenge to the reader to define, find ways to detect, and publish other process pathology signatures for the common good.

1 Introduction – The problem is massive

Senior capacity planning and system performance consultants are often tasked to find and resolve problems in emergency situations where there is a bottom line impact. About 35% of the time, the client's problems are due to undetected, miscreant processes consuming huge amounts of resources for no practical business reason. Often, the real solution is to simply stop the offending processes and fix the code, thus avoiding the million-dollar hardware expenditure that the client was expecting.

If the client originally asked for a workload growth study, the prudent analyst must now find a sample without process pathologies as a base for their analysis and models. Engagements tend to lengthen when it is difficult to find a period when "happy users" were acting as they would on a well-behaved system.

Any good performance analyst has seen so many examples of these process pathologies [Smith and Williams, 2001, 2002] that they can recognize them just by looking at a consumption graph, appearing as a "guru" to the uninitiated. In reality it is astonishingly simple to detect these pathological processes, once you know what data to collect and what to look for.

While working as a consultant, there is little reason to staunch the flow of easy money rolling in due to these process pathologies. However, at some point, the allure of consulting may wear thin and you may find yourself at a huge site, as an employee, with different motivations.

As a new capacity planner, you are often flooded with their backlog of issues, and once again, a significant number of them are caused by process pathologies. While your new boss will be impressed by your ability to spot process pathologies, at some point you may tire of the huge queue of requests that are preventing you from doing the things you want to do, like writing CMG papers. You will begin to look for ways to automatically detect and possibly trouble ticket these pathological processes.

This paper will discuss pathologies we have found, our methods of automatically detecting them and hints to use if you plan to follow our example. The challenge to you is - "Find more, and publish!"

1.1 Getting Started

To find process pathologies, you need information about process resource consumption. You will also need some method to analyze all that data, be it a vendor product, a spreadsheet or a language you are comfortable with, such as perl.

The availability and accuracy of these metrics varies wildly by operating system, and how they were obtained. You can spend many years gaining a deep understanding of the mathematical and programming complexities involved in metric collection

Collecting data efficiently and with mathematically valid sampling and summarization methods is a task that can easily overwhelm large teams of people. For most time- and personnel-constrained shops, the answer will be purchasing vendor-supplied collectors that can help you collect, reduce and use the data for reporting, modeling and provide other useful analysis tools. To be sure, you may add some custom functions, but first find a good collection and tool vendor and there will be less to do.

If you do choose to go it alone, be sure to keep collection overhead low. Performance problems can often be caused by the rotten code used to look for performance problems. Also, understand what is being collected, and what is not and then, calculate the capture ratio, which is…

$$\frac{\text{the sum of all allocated process's consumption}}{\text{the machine's total consumption}}$$

…in each of the sample periods. On most UNIX systems, these calculations demonstrate that you need to run accounting and post process it against your periodic collection technique (most try ps) and then, for each process, in each period, calculate when and where all those CPU seconds and IOs went. Yuck! Those vendor-supplied solutions are starting to look a lot more attractive, aren't they?

The biggest surprise that most analysts face is that the vast majority of numbers that spew from the various operating systems (and many tools) are virtually useless, somehow flawed, or computed in error-prone ways. Concentrate initially only on total process CPU and IOs during a sample interval. Focus on automating the easy "wins", saving time to pursue the esoteric problems later.

1.2 What are Process/Workload Pathologies?

Let's start with the hypothesis that generally, process/workload resource consumption should correlate with the volume of business functions performed. If a process or group of processes (workload) don't behave this way, something is wrong. An example might help.

Figure 1 shows a problem-free week on a two CPU e-mail processing system. Far more e-mail is sent during normal working hours and to a lesser extent, during hours when people are awake on the weekends. If the resource consumption on the system follows this pattern, we can surmise that at least some of the processes do not fit our definition of "pathological".

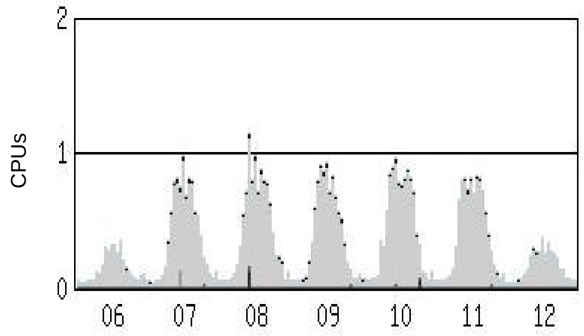

Figure 1, A Normal Week

Sadly, not all weeks are normal on this system.

Early in the next week, (Figure 2) someone "improved" a system utility, adding a check *that reads a growing log file*. Around lunch on Wednesday the 16th, an operator abnormally exited a utility, and thought nothing of it. However, his process lived on as a CPU loop, perpetually trying to find him until the machine was rebooted (due to performance complaints) a week later.

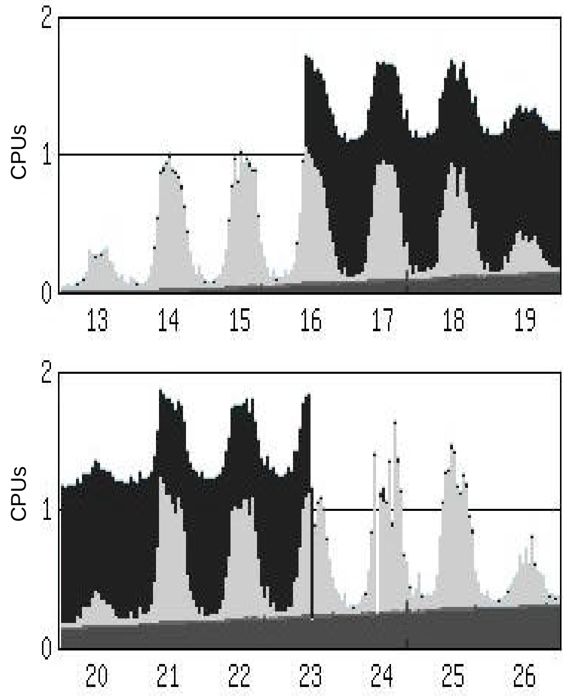

Figure 2, A Ramp Starts, A Loop Thrives

After the reboot on Wednesday the 23rd, the "improved" utility continued to read an ever-larger log file to examine the last record. No one noticed it.

Another operator caused more loops the next week, on Monday the 28th and again on Tuesday the 29th. Then mail really slowed down for ten days! (Figure 3)

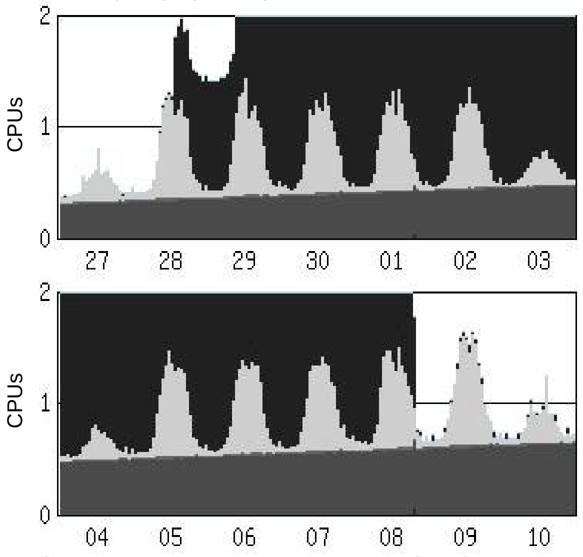

Figure 3, A Loop, Two Constrained Loops And A Big Ramp Grows Bigger

On Thursday the 8th, the loop was identified and "the problem" was "fixed". But mail still seems slow, and appears to be getting worse each day. Hey! Why does that "improved" system utility seem to increase its CPU utilization each day?

This example is based on historical consumption data (and user complaint calls!) from several actual systems, combined to show what can happen due to pathological processes that remain undetected. Some pathologies are violent, halting desired processing; these are usually immediately detected, but many lurk undetected, accumulating more and more expensive machine resources, adding more and more delay to the end user's response time. An outage might occur many days after the event that spawned the problem.

Without methods to isolate individual process consumption, these loops and ramps might have never been found. Whenever you hear phrases like "We boot this system once a week or it stops working", you have a big clue that what is really happening is that they are clearing out repeated buildups of undetected pathological processes. Wouldn't it be nice to isolate and eliminate the problems and the weekly outages for reboots?

In each of these cases, with 20/20 hindsight, we can say that the pathological processes are processes that consume resources in amounts either negatively correlated with business usage or in a pattern all their own.

1.3 Common Pathologies

There are many common process pathologies; the following is not an exhaustive list:

The Simple Loop - The simple loop is a process that takes over an entire CPU's worth of processing for each period it loops. Usually, a simple loop resides on a multiprocessor server with a lot of excess capacity, so it can continue for a long time. It may act normally for a period, then start looping due to some unique code path or environmental change. 99% of the time a loop will continue indefinitely. In figure 4, note how the first loop started on the 12th and continued until stopped late on the 14th, but because they addressed the symptom and not the cause, it recurred on the 15th. That's why repeated (we use daily) trouble tickets are great. Eventually, the programmers get sick of them and fix something.

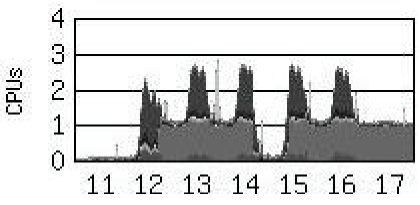

Figure 4, A Simple Repeated Loop

Note that loops do not have to stay in one CPU to be loops. In most modern, multi-processor operating systems, the loops will happily switch from one processor to another, unless someone programmatically locks the process into one. Then, it efficiently avoids context switches and loops even faster!

The Hum This one is controversial, but we believe that when consumption rises above certain thresholds, it is a pathology. Some processes are written to check an input queue of some sort for work, process any found, and then wait a bit before trying again. Sometimes, this is horrendously inefficient code, or someone reduces the wait interval to very low values in hopes of "improving performance".

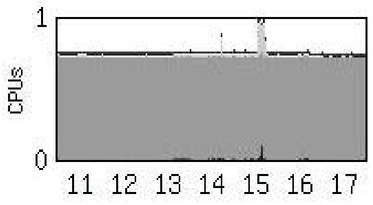

Figure 5, The Hum

In any case, the process consistently consumes significant system resources even when there is no real work for it to do. A lot of web-servers or processes whose roots are in older, non-event based systems are saddled with these. An extreme hum (we call it a shriek) looks a lot like a loop.

The Constrained Loop – In the mail system example, the single original loop could get all the resources it wanted, and it consistently took them. Starting on the 29th, and continuing until the 8th, there were more loops and real work than available resources, so the system saturated, assuring that the looping process cannot monopolize the CPU. Simply put, a constrained loop is any loop that, due to competition for scarce resources, can't get an entire CPU to itself, so it grabs all it can.

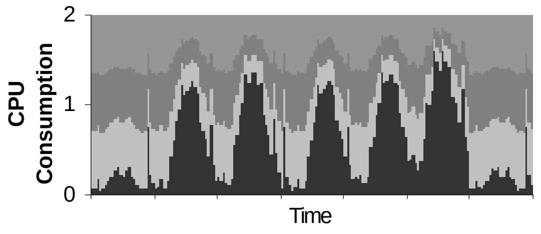

Figure 6, Three Constrained Loops On A System With Significant Real Workload

In the figure 6 above, there are three gray constrained loops on top of some real work.

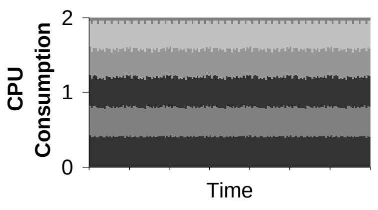

Figure 7, Five Loops No Real Workload

Constrained loops don't have to be caused by competition from real work. We have found unused, yet completely saturated Development systems with more loops than processors.

The Simple Ramp – In our mail system example, the "improved" system utility is a great example of a ramp. It, like most ramps, results from a bad programming decision such as reading to the end of a file to get the last record, and forgetting that this file is going to get huge over time. The impact of bad programming choices like this are almost never found in development, because the test cases are rarely at production sizes. Memory leaks can also form ramps when you chart the total memory allocated over time.

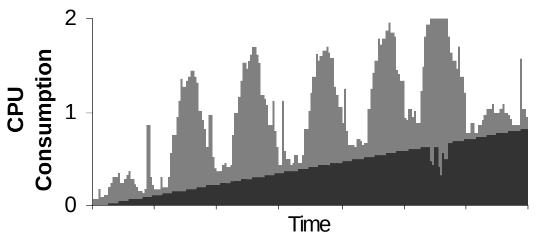

Figure 8, The Simple Ramp

Ramps can have many different slopes. Single day ramps are often easy to see. Slowly increasing multi-day loops like the example above are insidiously creeping up a little bit at a time and are often missed. We recommend looking at both short term (daily) and long term (monthly+) views of process resource consumption to detect these.

The Bumpy Ramp – Most ramps are really bumpy ramps, which are ramps that do have periods of negative slopes. How can this be?

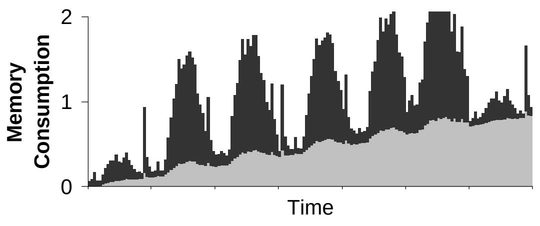

Figure 9, The Bumpy Ramp

Imagine a process with a slow memory leak. During normal processing, it allocates memory due to real demand and then frees it. Often this pattern follows business use. If the leakage rate is smaller than the de-allocate rate when business demands fall, you will have periods of negative slope. Left undetected, this one will eventually become a problem.

Bumpy ramps are usually due to a single continuous process, but if the process is reset or terminated and then restarted, you really have an example of …

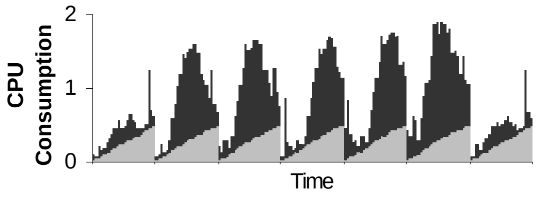

Figure 10, The Saw Tooth

The Saw Tooth – Ramps that reset periodically show up as a "saw tooth" utilization profile. If a response time increase or service interruption happens at the peaks, such as a disk filling up unnoticed we define it as a pathology. Since multiple processes can contribute to the growing ramps, this is often a workload pathology. Often, the reason people do not detect pathological "saw tooth" patterns is that they are examining too small of an interval, or too few. Growing log files, database rollback segments filling disks, and repeated memory leaks are often sources of saw tooth patterns. This may take time, but once diagnosed, the cure can be obvious.

Many More – Any process that requires repeated human intervention to detect and correct might be a workload or process pathology that we can automatically find.

2 The Challenge

Devise simple algorithms that efficiently detect process pathologies from data sources.

We repeatedly tried and failed to find "absolute" algorithms that found all problems. Our breakthrough came when we decided to divide and conquer, i.e. develop simple algorithms to find one type of pathology, most of the time. It is okay to miss a few. You will be quite busy with the ones you do find. Later, as things calm down, try more complex strategies.

2.1 Rules of the Chase

Ideally the solutions would:

- Follow a "single data collect, multiple use" doctrine
    - o Collect process data
    - o Turn on accounting if needed
- Embrace simplicity
- Find a large percentage of the problems
- Provide most, if not all, information needed to address the pathology at the time of notification.
- Embrace parameter files to minimize the need to change code; it is error prone and tedious
- Embrace fuzzy logic, if needed. Example: If a process meets 3 of 5 criteria, it might be a pathology
- Use tools that are commonly available, the cheaper the better
    - o Perl rules!
    - o Spreadsheets are ubiquitous, but macros can be tough to keep running long term
- Combine notifications into a minimal set of messages
    - o Limit numbers of email notifications
- Consider a FYEO (For Your Eyes Only) class…
    - o Don't write tickets for yourself
- Run private (FYEO) for a while before going public.
    - o Nothing is worse than false positives!
    - o Build guru status by mysteriously finding all this weird stuff that everyone else misses!
- Denny's Law - Never alert on something that you can't explain to someone paged at 3:00 AM. [Brewer]
- Ron's Law -Never add over a thousand nodes to your automated check system on a Friday afternoon or before you take a vacation!

2.2 Criteria for Success

- It is not a "who's method is better" argument. If your method works at all in your situation, it is a great method!
- Strive for low, or no, false positives
- Seek simplicity, low resource consumption, and elegance
- Always code for exceptions! Always! Notification fatigue due to repeated false positives will kill your effectiveness!
- Write for the whole world; comment your code
- You don't have to be perfect! You just have to try

Our perl-based solution checks all processes during all hours on all 2000+ nodes and averages about three seconds per node.

3 How we currently do it

At Safeway Inc., we run automatic process pathology detection tests on nearly 2000 AIX, LINUX, Solaris and various flavors of Windows distributed systems nodes each day. It is not uncommon to find 8-12 pathologies every day, and sometimes many more! Since we started noticing and alerting automatically, our requests for in-depth performance investigations have dropped off dramatically. It is nice to spot a problem before the users do!

Start a new test on a subset of nodes, and widen it out when it is proven. Run FYEO for a while, and do the pre-training, then warn the support staff about what is coming. Expect to spend significant phone time explaining why the programmer on the other end of the line should care about the problem. Graphic evidence accompanying your calm, yet firm, explanations is indispensable, so get good at generating graphs quickly.

3.1 How To Detect The Simple Loop

Theory: A process that uses an entire CPU for an extended period of time is often not desirable. Detect and report loops that exist for extended periods of time

Practice: The real world is less pure. The simple loop is a great place to start, because there are so many of them.

When you start hunting you will quickly notice that it is almost impossible for a process to get 100% of a CPU, indeed, on some operating systems, a loop would be lucky to get 87%. Once we decided to forget precision and learn to love brute force, we found that specifying a mean and allowable variation worked quite well.

Exception Note: Whatever automated checks you do, and especially in the case of Simple Loops, you will encounter exceptions. Program in exception handlers from the start, or expect embarrassing interruptions in notification while you wrestle with your code. In our shop, we have several statistical packages that run off "in memory" databases to calculate amazing things that enable us to serve you better. These can run for many hours, and they look a lot like a loop. Discussions with the analysts that run them helped us find that they also can get stuck on bizarre queries, and the analysts wanted to be notified when that happened. Working together, we decided that any time one of these loops ran for 16 or more hours out of 24, it deserved a ticket. With this exception in place, there are no "false positives", and resources aren't wasted on runaway queries any more.

Some multithreaded database processes that are just busy enough can look like loops at the process accounting level. Be ready to find exceptions with processes like sqlserver.

Parameters We Use: Earlier in "Rules of the Chase" we mentioned that you should probably approach these searches via parameters. Here are the ones we use for simple loops:

- Function (process_loop)
- Operating System (AIX, HPUX, Linux, Solaris, WindowsNT, Windows2000, etc)
- Allowed Deviation%
- Loop Mean
  - Example: 0.05 Allowed Deviation with 1.00 Loop Mean finds any process whose CPU consumption was between 95% and 105% of an entire processor
  - Why a lower *and an upper* limit?
    - The lower says "at least this busy"
    - The higher says, "no busier than", and helps weed out busy multi-threaded processes like sqlserver.
  - *Greater than 1*?
    - It happens. Remember, when sampling computers, there is always sample error, and sometimes there can be more CPU attributed to a process than there were seconds available.
- Calculation method (span i.e. it must loop for a span of time)
- Hours per day to qualify (i.e. the process must loop for at least 8 hours in 24 to trigger)
- Output choice (mail, trouble ticket system or file, node history file, other)
- Loop File name (if written to a file) or whatever method you use to interface with your trouble ticketing system
- Mail Recipients

Actual Loop Checker Parameter Examples:

- process_loop,Linux,.05,1,span,8,summary_mail node_history trouble_ticket, /a_directory/ticket_logs/loops,ronmail\@the_firm.com linux_dudemail\@the_firm.com
- process_loop,Windows2000,.08,0.92,span,8,summary_mail node_history trouble_ticket, /a_directory/ticket_logs/loops, ronmail\@your_firm.com dennymail\@your_firm.com

Parameters We Use for Loop Exceptions:

Exception handling for "Simple Loops" is easy! All you need to do is add hours.

Example: If a normal loop triggers at 8 hours in 24, this one has to loop for 8 more (16 total) before it triggers. What happens if you add 24, or any number higher than (24-(hours-per-day-to-qualify))? The process *never* triggers.

- Function (process_loop_exception)
- Operating System (AIX, HPUX, Linux, Solaris, WindowsNT, Windows2000, etc)
- Exception process name
- Additional hours needed to qualify as a loop

Actual Loop Checker Exception Examples:

- process_loop_exception,AIX,DISGUISED_NAME,8
- process_loop_exception,WindowsNT,sqlservr,12
- process_loop_exception,Windows2000,sqlservr,12

That looks pretty simple, doesn't it?

3.2 How To Detect The Constrained Loop

This one was tough. We found our first test case by examining a node somewhat infamous for Simple Loops that had not triggered any lately. What we found resembled the example presented earlier in the paper (see Figure 6), a saturated node, with more loops than available processors. Since a process likely to loop might choose to do it a lot, you need to find these.

Theory: A process that would use an entire CPU for an extended period of time (if it were not prevented from doing it by competition) is often not desirable. This competition originates from both real work and often other constrained loops. Detect and report loops that exist for extended periods of time

Practice: There are at least two types of Constrained Loops with very different properties! There are different ways to check for each type.

Kibitzing: Why doesn't the Simple Loop checker find them? Why not lower the mean and widen the spread?

Answer: There are infinite special cases that wreak havoc on any attempt to find Constrained Loops with large spans around a mean, and the number of false positives will be substantial.

Exception Note: We use the exact same code to check for exceptions and to notify for all loop types.

Finding Constrained Loops When Real Work Is Present on the Node Along With Loops:

The initial idea was that on a node with significant real work, constrained loops would all consume roughly the same resources and have a negative correlation coefficient [Ding, Thornley, Newman, CMG2001] when compared to actual work and a highly positive one when compared to each other. This puts you in the unhappy position of trying to find an automatic way of deciding whether each and every process is a loop or a good one, before deciding if they are loops.

So, keep it simple, and just look for high positive correlations. This turned out to be amazingly effective. The computational load of computing correlation coefficients for every process pair was daunting, so we developed a simple crutch. There could be infinite loops, each using a tiny bit of CPU, and we had ways of detecting them at that point. So, as a first filter, reject any process record that consumed less than 10% of a CPU during an hour. Then, compute unbiased correlation coefficients on the remaining small number of process pairs and list as "probable Constrained Loop" any two processes that have a correlation or 0.66 for enough hours out of 24 (we use the Simple Loop value for that OS).

Incidentally, 0.66 is pretty conservative. Most "Constrained Loops When Real Work Is Present on the Node" pairs correlate at or above 0.9. With those simple filters in place, we find almost all Constrained Loops and suffer almost no false positives. Seems easy, doesn't it?

Finding Constrained Loops When Real Variable Work Is *<u>Not</u>* Present on the Node Along With Loops: It turns out that if you have more loops than available processors on an otherwise dormant or static machine, the correlations between process pairs is essentially a chaotic number between –1 and 1.

We were finding all the Constrained Loops on nodes where there was real processing demand, and those were likely to be the ones we were most interested in. Still, we want to find them all. Here's an example that shows our results.

Imagine an otherwise dormant two-processor machine with five Constrained Loops on it. Each of the five Constrained Loops was getting about 40% of a single CPU over time. However, when closely examined, you will see small variations as the loops wrestle control from each other.

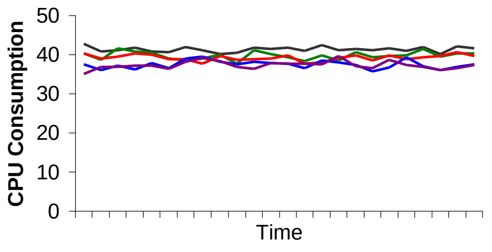

Figure 11, CPU Used By Five Constrained Loops on a Two-Processor System

Since all of the means are so close, and the variances are so small, the process of computing correlation coefficients causes these small variations to take on undue significance, and the result is that correlation coefficients aren't high and positive, they are all over the place.

We can't use those correlation coefficients to find Constrained Loops on dormant systems!

But, we can take advantage of their close means. If the process consumes more than 10% of a single CPU during an hour, and its value lies within two standard deviations (plus or minus) of the mean of another suspect, tag it as a probable. If two processes meet this test for enough hours in a day (again, use the number for that OS from the Simple Loop parameters), then send mail to the capacity planners. So far, this simple "clustering" detection method has worked exceptionally well, but try it for a while to make sure that there are no lurking unforeseen special cases. We recommend that you try new detection methodologies in a stealth mode too.

Loop Wrap Up: Hopefully this demonstrates that via relatively simple formulas acting on minimal metrics, you can find almost all of the CPU loops plaguing your systems.

3.3 How We Detect The Ramp, at least so far…

There is one process present on most of our systems that has a history of slow "Bumpy Ramp" behavior. Left to its own devices and given enough time, it will take over an entire machine. That process is the model for the ramp in our example, but it is a lot bumpier.

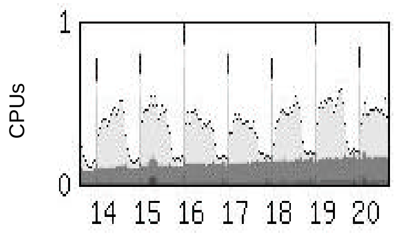

Figure 12, The Lurking Ramp

All formulaic attempts that we've tried so far are mired in complexity. Depending on the ratio of "bumpy ramp behavior", slope, and number of periods easily available to examine, different formulas work or fail. High slope ramps are easy to find, subtle slope ramps are really tough. How do you distinguish between a ramp's slope and another valid workload whose slope is mathematically exactly the same during some periods?

Ramps do have one "easy to see" quality. Eventually, they grow past any threshold you want to set. You must identify and set thresholds for key workloads or processes. That said, automatically detecting a problem on 50 nodes a day and sending daily alerts can sure focus attention on a widespread problem. Also, it is a trivial matter to decide on a maximum value that a given workload should be allowed to consume on a machine and alert when it exceeds it.

For example, if all Tools (monitors, collectors, anti-virus, backup and restore activity, disk defraggers, etc.) are in one group, it is easy to make a statement like "The Tools workload should never take more than 10% of any box". It is recommended that you run this one FYEO or in stealth mode (ours still is), notifying only select individuals. You may be amazed at how much your Tools infrastructure is eating your total infrastructure.

4 Summary – Join The Hunt!

You should now be convinced that automatically finding process pathologies is relatively easy to try, and has the potential to seriously improve the end user's experience and reduce your investigation workload. While we've given examples of finding the "usual suspects", with both supremely accurate mathematical precision and kludge methods, we are convinced that many more pathologies are out there waiting for some bright individual to discover their simple detection algorithms.

A good ramp detector would be particularly useful for detecting memory leaks and slow ramps that sneak up on you. We are devoting serious energy to this one and will happily cooperate in testing algorithms that look promising.

If multiple people contribute working ideas, we are willing to post the underlying code/algorithms to some public place, for everyone's use. This paper is a first step in that process, and I eagerly expect further research and discussion in this area. Please feel free to contact the author regarding coauthoring new process pathology detection papers, possibly helping test your ideas against our mountain of process data, or even just telling you about the disaster that we created when we tried "that" idea. Let's get started!

5 Appendix: Formulas! Pseudo-Code!

Perhaps picking the precise way we did it from the text is not your favorite method. Try this:

Where:

d^{os} =	allowed deviation for that operating system
$loop^{threshold}$ =	number of hours in your review period required to qualify as a loop (we use 8)
m^{os} =	loop mean for that operating system
m^{low} =	$m^{os} - d^{os}$
m^{high} =	$m^{os} + d^{os}$
p^{cpu} =	CPU consumed by a unique process
p_n^{cpu} =	CPU consumed by a unique process *n*
p^{hours} =	number of hours in your review period that p^{cpu} looped
$p^{exception\_hours}$ =	the additional hours needed for a p^{cpu} on the exception list to qualify as a loop. Note that when $loop^{threshold} + p^{exception\_hours}$ >= total periods, *this process will never trigger a loop!*
σ_{pncpu}=	standard deviation of CPU consumed by a unique process *n* during the hours where both processes existed.
m_n^{cpu}=	computed mean of the CPU consumed by a unique process *n* during the hours where both processes existed.

Note: We usually examine 24 one-hour periods each day.

Detecting Simple Loops:

```
If ($p_n^{cpu}$ >= 10% of a CPU on that machine)) {
    If ((the machine is not saturated) {
        For each $p^{cpu}$ {
            $p^{hours} = 0$
            For each hour {
              If (($p^{cpu} >= m^{low}$) and ($p^{cpu} <= m^{high}$))
              then $p^{hours} = p^{hours} + 1$
            }
            If ($p^{cpu}$'s process name is on that
              operating system's exception list) {
              $loop^{threshold} = loop^{threshold} + p^{exception\_hours}$
            } else {
              $loop^{threshold}$ = normal $loop^{threshold}$
            }
            if ($p^{hours} >= loop^{threshold}$) then it's a loop!
        }
    }
}
```

Detecting Constrained Loops:

For All Constrained Loop Types:

If ((p_n^{cpu} >= 10% of a CPU on that machine)
 and (the machine is saturated)) {

… p_n^{cpu} is put on the review list for that hour.

With real variable work present (Correlation Coefficient):

$$C(p_1^{cpu}, p_2^{cpu}) = \frac{COV\ (p_1^{cpu}, p_2^{cpu})}{\sigma_{p1cpu}\ \sigma_{p2cpu}}$$

…which yields a number between –1 and 1.

If ($C(p_1^{cpu}, p_2^{cpu})$ is >= 0.66), it's a constrained loop!

For a much better description of computing correlation coefficients, see [Ding, Thornley, Newman, CMG2001], which is what I used.

With no real variable work present (Mean and spread):

If the previous formula didn't find any, you either don't have constrained loops or the competing work is non-existent or extremely consistent, like a hum or a shriek. Try this:

If (p_1^{cpu} >= ($m_2^{cpu} - 2\sigma_{p2cpu}$) and
 (p_1^{cpu} <= ($m_2^{cpu} + 2\sigma_{p2cpu}$) {

…you probably have a suspect.

This is simply a test to see if your suspect process's mean consumption is within two standard deviations of another suspect's mean.

Detecting Ramps:

```
If ($p^{cpu}$ >= threshold) {
    …notify someone!
}
```

Errata:

If ((p_n^{cpu} >= 10% of a CPU on that machine) is just a simple culling technique to reduce computing correlations for insignificant processes.

One really tricky bit is that you must remember that you have to re-compute m_1^{cpu}, m_2^{cpu}, σ_{p1cpu}

and σ_{p2cpu} each time for each pair, including only the hours where both members of the pair qualified as loops. Suspect this when your computed correlation coefficients exceed +-1.

Also, one hole in this method occurs during spans of time when the number of loops keeps changing. Imagine a node where a new constrained loop joins in frequencies shorter than your aggregation interval (we use an hour), and randomly someone kills some. This will wreak havoc on your correlations. We do have three issues in our favor here, 1) this is extremely rare, 2) a node adding loops this fast will saturate and you'll quickly notice it for other reasons, and 3) we don't have to be perfect, we just have to try!

I only said it looks simple, remember?

6 References

[Smith and Williams, 2001] C. U. Smith and L. G. Williams, "Software Performance Antipatterns: Common Performance Problems and Their Solutions" CMG 2001 Proceedings, Vol 2, pp 797-806, Anaheim CA, December 2001

[Smith and Williams, 2002] C. U. Smith and L. G. Williams, "New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot" CMG 2002 Proceedings, Vol 2, pp 667-674, Reno NV, December 2002

[Brewer] Denny Brewer, fellow capacity planner, master of automation, whose sage wisdom, implacable attention to detail and good-humored patience makes all this work possible.

[Ding, Thornley, Newman, CMG2001] Yiping Ding, Chris Thornley and Kenneth Newman, *On Correlating Performance Metrics*, CMG 2001 Proceedings, Vol 1, pp 467-477, Anaheim CA, December 2001

Yiping, Chris and Kevin's papers are great resources for formulas and precise and concise descriptions of how to apply them.

A special thank-you to Denise Kalm, the best CMG paper mentor and editor you could hope for. This paper is 200% better due to her efforts.

7 Legalese

Work safe, and have a good time!