# Automating Process and Workload Pathology Detection

Ron Kaminski

Safeway Inc.

# Introduction

- Who is this Ron Kaminski anyway?
- The CMG2003 presentation
- Glimpses of our reporting system in action
- Even more material on ramps, and why you should care about them
- Contact information slide at the end
  - ron.kaminski@safeway.com
    - pdf of the paper
    - Example website
    - Example perl and css

# Ron Kaminski

- Director of Capacity Planning and Performance Analysis at Safeway Inc.
- Responsible for modeling, problem investigation and performance studies on over 4,100 midrange *NIX and over 25,000 PCs in 1800 stores nationwide

# Ron's Problem

- No staff, existing staff shrinking
- Too many machines to even look at, much less analyze
  - Over 600 back-stage *NIX machines
  - 1800 stores, and each has
    - 2 SCO Point Of Sale controllers
    - 12-18 PC's handling all payroll, time clock, benefits and training processing
    - More coming
      - Signature Capture
      - Bio-terrorism attack, I'm not kidding!

# Ron's Solution

- Humans don't scale
- Drastically reduce reporting cycle times
- Answer the four main questions, every time
  - Do I need to buy CPU?
  - Do I need to buy Memory?
  - Are Disk Changes needed?
  - How are Workload Response Times changing?

# Ron’s Solution

- Automate Process Pathology Detection
- Provide consistent views across machines and over time.
- Always provide historical context
- Generate graphics and reports on demand

# The Problem Is Massive

- About 35% of the machines I've done studies on have undetected miscreant processes
  - Looping processes
  - Runaway SQL queries
  - Ramping up consumption as problems proliferate
  - Single threaded applications with runaway queries
- While there are many threshold monitors out there, most do not have the information or the historical context to discover process pathologies
- You can see them if you know what to look for, but
  - There are too many machines to look at!

# What Is a Process or Workload Pathology?

- Defining normal and abnormal consumption
  - Hypothesis 1: In interactive online environments resource consumption should vary with transaction load
  - Hypothesis 2: Even when batch jobs are present, a single process seldom should take an entire CPU for extended periods of time
    - If it is valid processing, shouldn't it break for IOs now and then?

# What Is a Process or Workload Pathology?

  - Hypothesis 3: It is rare for valid business data processing to consume resources (CPU, Memory) in a steady ramp upwards for extended periods
- Processes or workloads whose resource consumption breaks any of these hypotheses is judged abnormal, a potential pathology
- You have to check
  - Every machine you have
  - Every process
  - Every day

# Pathology Hunting Prerequisites

- You'll need:
  - A method to collect process resource consumption information which lets you look for:
    - Loops
    - Constrained loops
    - Memory leaks
    - Bizarre behavior
  - Deep understanding of the operating system
  - Secondary analysis and graphics production tools
    - Perl
    - Spreadsheets
    - Vendor  analysis products

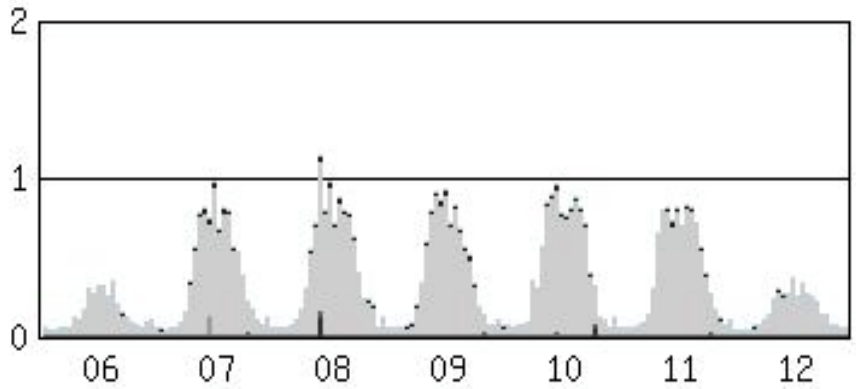
# Pathology Hunting Prerequisites

- Don’t underestimate the difficulty of creating a collector on your own
  - Mathematically valid random sampling is really tricky
    - Huge teams work for years on collectors
  - Many operating system numbers are not even close to what they claim to be
  - Often you are stuck with an OS vendor’s number which was incorrectly sampled!
- Most of us will buy a commercial collector
- Know your capture ratio

$$\frac{\text{What you can allocate to specific processes}}{\text{The total consumed on the system}}$$

# Typical Pathology Example

- A normal, problem free week on a two processor mail system
- More mail is sent during daylight hours
- Less mail is sent on weekends
- Utilization peaks near 50%
- Response time is good
- Utilization follows transaction volume



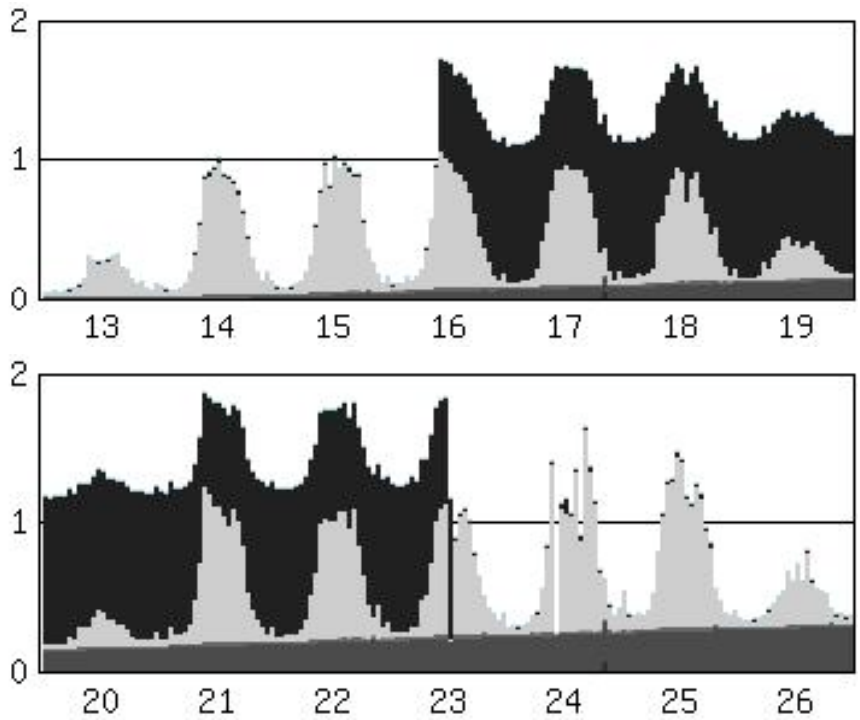

Week 1, Normal

No Pathologies

Yet…

# Typical Pathology Example

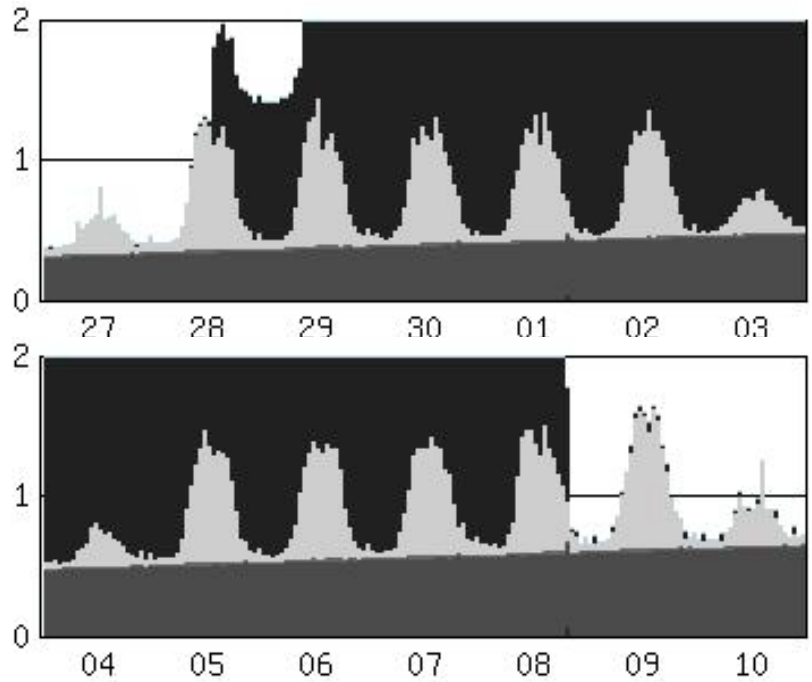
- Early in Week 2, someone "*improved*" a system utility
  - They adding a check that reads a growing log file
- Around lunch on the 16th, an operator abnormally exited a utility
  - Their process lived on, valiantly looping, trying to find them
- Daily performance complaints began arriving
- A system reboot on the 23rd seemed to help




Weeks 2 & 3, A Ramp Starts, A Loop Thrives

# Typical Pathology Example

- Around lunch on the 28th, an operator abnormally exited, leaving another loop
- The did it again the next day!
- User performance complaints rose dramatically
- A reboot on the 8th helped, but…
  - performance still seems worse than last month!




Weeks 4 & 5, A Loop, Two Constrained Loops and a Ramp gets bigger

# Typical Pathology Signatures

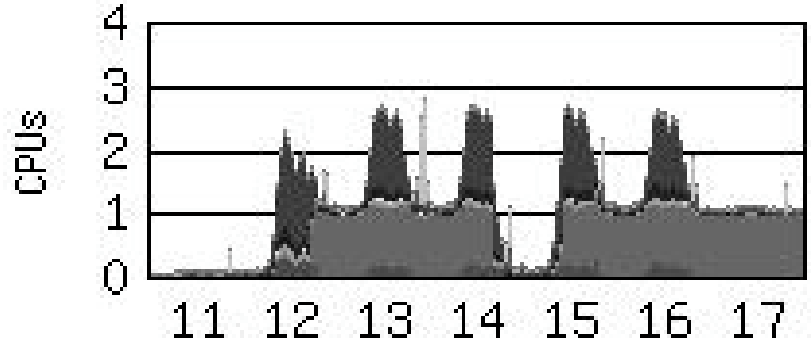
- A node that “needs a reboot” regularly
  - You are just clearing out built up pathologies!
  - Until you find them, you will be rebooting forever
    - Should you keep bailing or fix the leaky boat?
- Tip-off phrases
  - “We don’t have time to fix every little…”
  - “The performance problems on the server are not in my project’s scope”
- User complaints come in waves

# Typical Pathology Signatures

- Pathological processes and workloads consume resources in amounts either negatively correlated with business usage or in a pattern all their own
- Lets see some pathology patterns!

# The Simple Loop

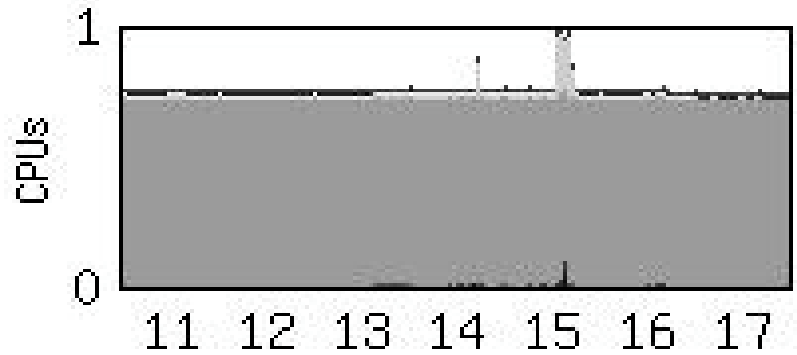
- Signature:
  - A single CPU high “fence” of consumption
  - Often IOs are minimal
  - Ends with process termination
    - Resolution often inadvertent, like reboots
  - Often repeat
  - Will switch processors on most modern operating systems
  - Easy to see if graphed versus number of CPUs




The Simple Loop with natural repeats

# The Hum

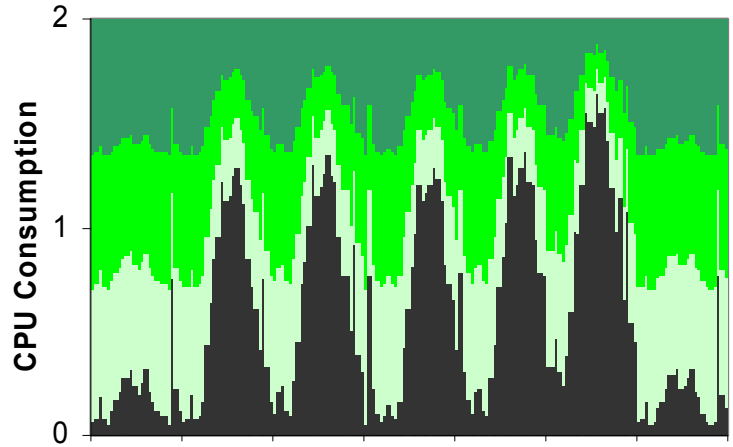
- Signature:
  - Consistently consumes a lot of resources
    - Even when there is no work to do
  - Often “web” code or any with “look for work” loops
    - If we look more often, it’s better, right?
  - Severe “Hums” are called “Shrieks”




The Hum

# Multiple Constrained Loops With Real Work Present

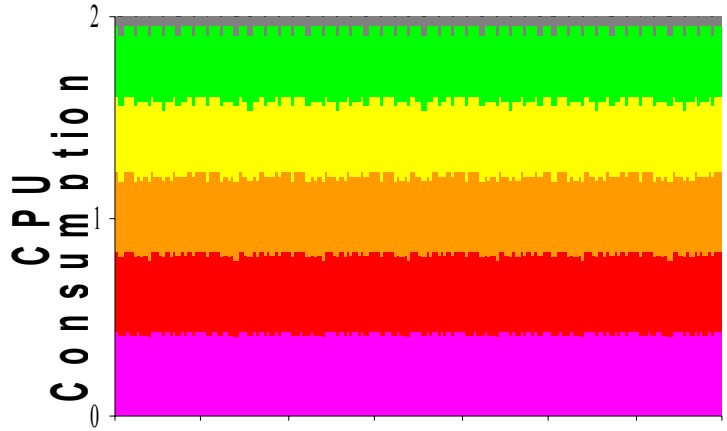
- Signature:
  - A loop that can't get a whole CPU due to contention
  - Node is often saturated
  - Often occur in multiples that have high CPU usage correlations
  - Undetected "Simple Loops" build up and become Constrained Loops if given time




Three Constrained Loops with Real work Present

# The Constrained Loop With No Real Work Present

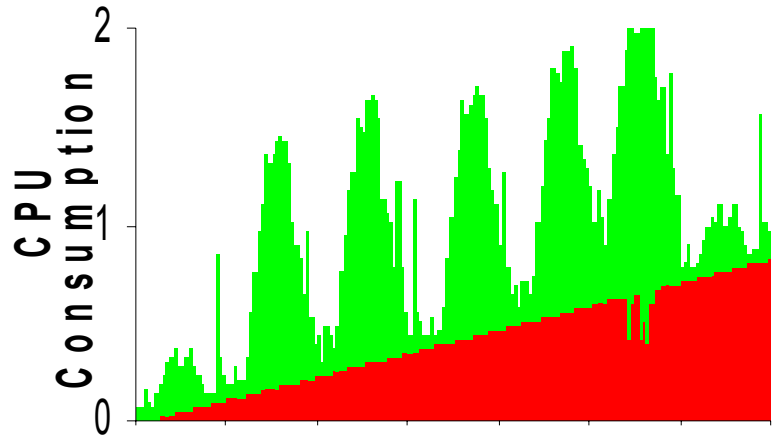
- Signature:
  - All the Constrained Loop with Real Work Present issues except…
    - CPU Usage Correlations are usually awful




Five Constrained Loops with No Real work Present

# The Simple Ramp

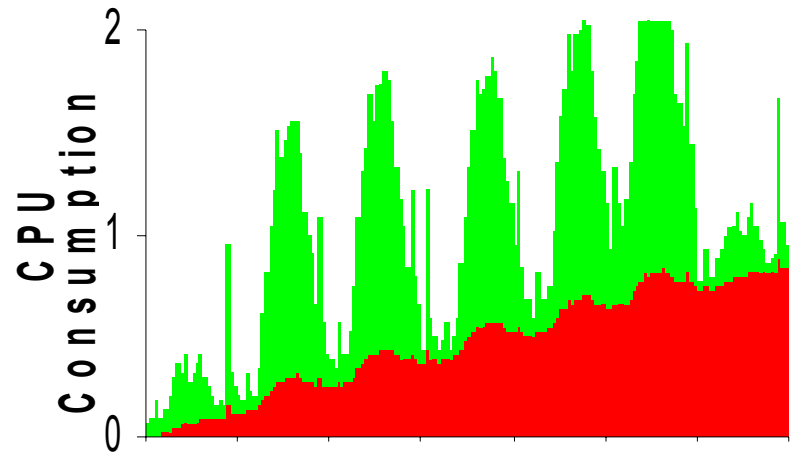
- Signature:
  - Almost continuous increases in resource consumption when unconstrained
  - Common causes
    - Reading growing logs
    - Queue entries not being deleted when completed, building up ever increasing work for the queue manager
    - Simple memory leaks




The Simple Ramp

# The Bumpy Ramp

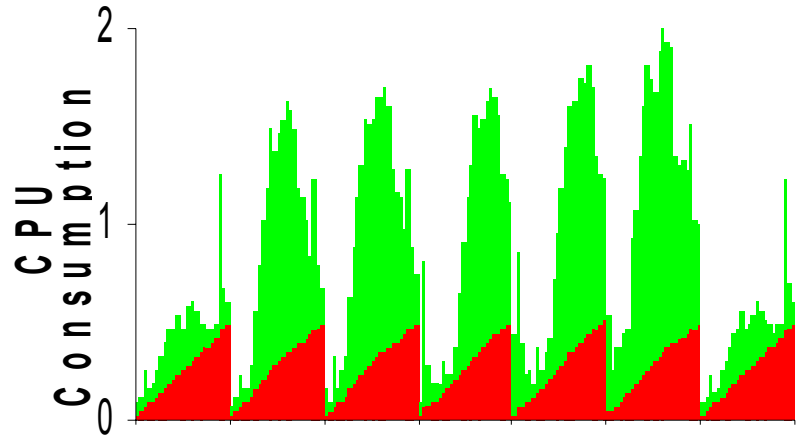
- Signature:
  - A lot like a simple ramp, but with (usually regular) periods of decreasing use
  - General trend over long periods of time is always up
  - Typical of slow memory leaks
  - Very difficult to describe mathematically!




The Bumpy Ramp

# The Saw Tooth

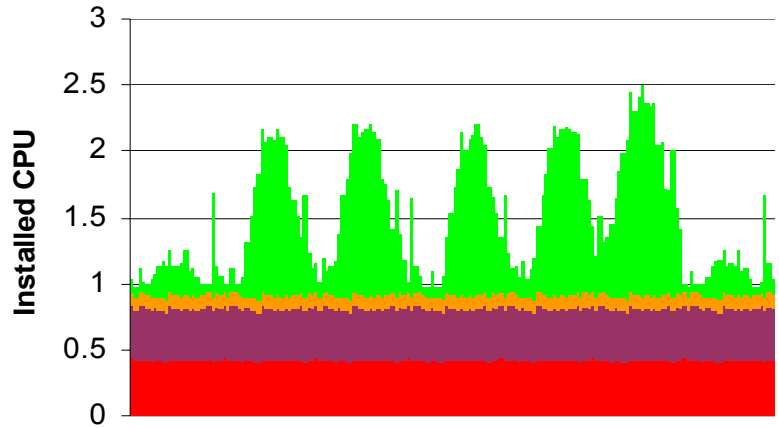
- Signature
  - A series of resetting ramps
    - Only the ramp trigger gets reset to zero periodically
  - Commonly seen in:
    - Monitors reading daily logs
    - Disks that fill and are cleaned up




The Saw Tooth

# Session Bonus: The Single Threaded Multi-process Application Loop

- Signature
  - A stuck query in software (often java) that calls middleware and then a database as fast as it can
  - Single threaded application using multiple processes
  - Seldom eats an entire CPU because of inter process communication delays
  - Usually run until killed



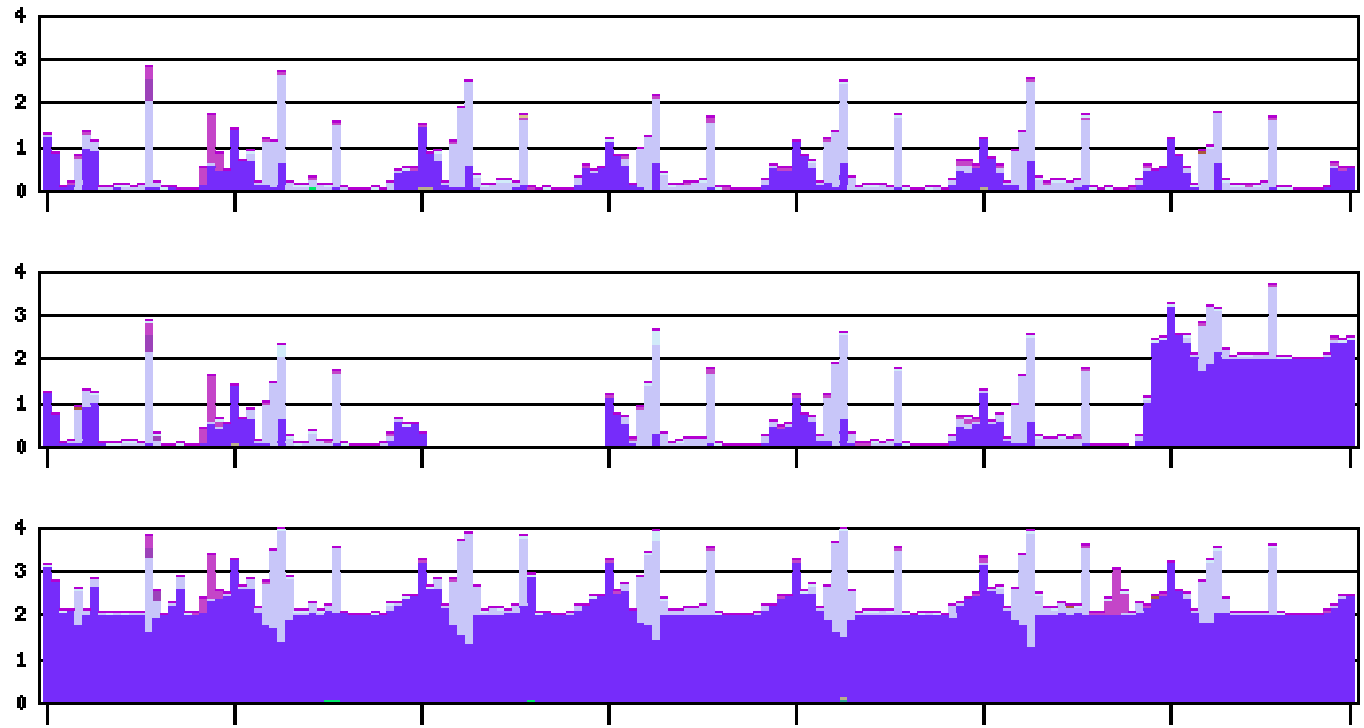

Single Threaded

Multi-process

Application Loop

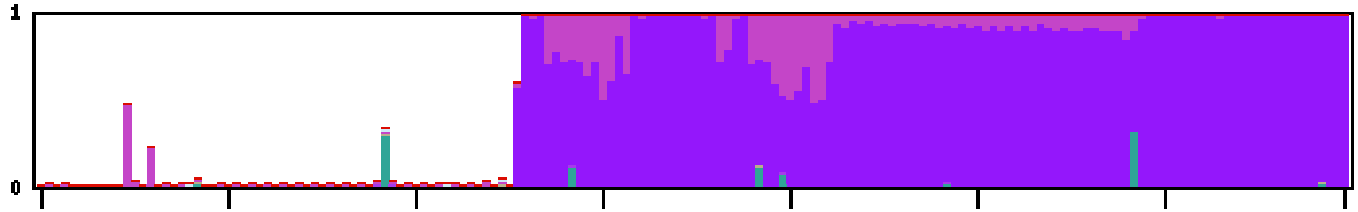
# Session Bonus: The Multi-threaded Single Process Application Loop

- A workload with a dual threaded looping process!



# Session Bonus: The Constrained Single Process Application Loop With Real Work Present

- A workload with a single looping process on a single processor machine with significant real work present




  - My current loop robots *don't see it* if total other work keeps it below thresholds
    - If (($p^{cpu}$ <=$m^{low}$) it is invisible!
    - My saturation robot sees it, but those warnings often are ignored due to high "who cares?" hits.

# And Many More…

- This is not an exhaustive list
  - The panoply of bad coding techniques…
- Any process that requires repeated human intervention to detect and correct might be a workload or process pathology that we can automatically find
- But, these are plenty to get started with!

# The Challenge

- Devise simple algorithms that efficiently detect process pathologies from data sources and publish your algorithms!
  - You don't have to be perfect!
    - Absolute perfection is very difficult
    - If your method is easy and finds a large percentage without false positives, it is a great method!
  - Start with simple issues, the complex ones will still be there waiting for you later

# Rules of the Chase

- Ideally the solutions would:
  - Follow a “single data collect, multiple use” doctrine
  - Collect process data
    - 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; which is error prone and tedious

# Rules of the Chase

- Ideally the solutions would:
  - 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
    - Perl rules!
    - Spreadsheets are ubiquitous, but macros can be tough to keep running long term
  - Combine notifications into a minimal set of messages
    - Limit numbers of email notifications

# Rules of the Chase

- Ideally the solutions would:
  - Consider a FYEO (For Your Eyes Only) class…
    - Don’t write tickets for yourself!
  - Run private (FYEO) for a while before going public
    - Nothing is worse than false positives!
    - Build guru status by mysteriously finding all this weird stuff that everyone else misses!

# Rules of the Chase

- Ideally the solutions would:
  - Follow Denny's Law
    - Never alert on something that you can't explain to someone paged at 3:00 AM!
  - Follow Ron's Law
    - Never add over a thousand nodes to your automated check system on a Friday afternoon or before you take a vacation!

# 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
  - Elegance

# Criteria For Success

- 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!

# What We Do

- 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 (and ideally fix it) before the users even notice it!

# What We Did

- Start a new test on a subset of nodes, and widen it out when it is proven
- Run FYEO for a while
  - do the pre-training
  - warn the support staff about what is coming
- Expect to spend significant phone time explaining why the programmer/vendor 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
  - Have a way to send proof graphics outside the firewall if you discover issues in third party software

# What We Did

- People hate tickets
  - If you generate a stream of them, they won't be too fond of you either
  - Win them over with patience, good humor and graphs
- Work the management chain from the top down
  - Make sure they see the value
    - Increased productivity
    - Root causes of performance issues found
    - Decreased panic induced problems
    - We'll detect and fix it before it gets really bad
  - Make sure that their people know that they support it

# 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

# How To Detect The Simple Loop

- Parameters we use:
  - Function (process_loop)
  - Operating System (AIX, HPUX, Linux, Solaris, WindowsNT, Windows2000, Windows2003, 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

# How To Detect The Simple Loop

- 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 sqlservr
- *Greater than 1 CPU*?
  - 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

# How To Detect The Simple Loop

- Parameters we use:
  - 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
    - others

# How To Detect The Simple Loop

- Parameters we use:
  - Loop File name (if written to a file) or whatever method you use to interface with your trouble ticketing system
  - Mail Recipients
    - You will change this a lot
    - Your time savings due to this one alone is worth adding a parameter system

# How To Detect The Simple Loop

- Given:
  - $d^{os}$ = 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:
    - when $loop^{threshold} + p^{exception\_hours} >=$ total periods, *this process will never trigger a loop!*

# How To Detect The Simple Loop

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) then
            {$loop^{threshold}$ = $loop^{threshold}$ + $p^{exception\_hours}$}
        else {$loop^{threshold}$ = normal $loop^{threshold}$}
        if ($p^{hours}$ >= $loop^{threshold}$) then {$p_n^{cpu}$'s a loop!}
} } } }

# Simple Loop Exceptions

- Simple Loop Exceptions are easy!
  - Just 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
  - Sometimes you will get applications that look like loops for slightly greater than your threshold
    - In-memory “cube” databases
    - Make it more difficult, but not impossible to trigger
      - They have runaway queries too!
  - Sometimes code owners deny reality…
    - Revisit your exceptions from time to time

# Simple Loop Parameters

- 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

# Simple Loop Exception Parameters

- process_loop_exception,AIX,DISGUISED_NAME,8
  - DISGUISED_NAME must look like a loop for 8 regular plus 8 more = 16 hours out of 24 to trigger a warning
- process_loop_exception,WindowsNT,sqlservr,12
  - sqlservr must look like a loop for 8 regular plus 12 more = 20 hours out of 24 to trigger a warning
- process_loop_exception,Windows2000,sqlservr,12
  - sqlservr must look like a loop for 8 regular plus 12 more = 20 hours out of 24 to trigger a warning
- This looks pretty simple so far, doesn't it?
  - The payback from finding simple loops is huge!

# How To Detect The Constrained Loop

- **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

# How To Detect the Constrained Loop When Real Work Is Present

- **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

# How To Detect the Constrained Loop When Real Work Is Present

- We tried and failed a lot!
- The effective way turned out to be simple
  - If the machine's CPU is saturated
    - Check correlations of significant consumers
    - Significant consumers with a high positive correlations (0.66 or better) are usually constrained loops
      - 0.66 is pretty conservative, most correlations are in the high 0.9s
      - We may raise it to the 0.9 to weed out a few innocent database processes that get tagged when huge numbers of constrained loops are present

# How To Detect the Constrained Loop When Real Work Is Present

- When significant consumers have a high negative correlation…
  - one of them is real work
  - one is probably a constrained loop
  - That pair does not qualify in this round
    - If one really is a loop it will probably qualify in another process pair
  - What is a significant consumer?
    - We use (minimum consumption to qualify = 10%) just to dramatically reduce the number of correlation calculations
- Note that this also finds CPU constrained processes
  - Not what we intended, but we'll take it!

# How To Detect the Constrained Loop When Real Work Is Present

- Given:
  - $\sigma_{pncpu}$= standard deviation of CPU consumed by a unique process *n* during the hours where both processes existed
- Remember you must re-compute $\mu_1^{cpu}$, $\mu_2^{cpu}$, $\sigma_{p1cpu}$ and $\sigma_{p2cpu}$ each time for each pair, including only the hours where both members of the pair qualified as loops

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

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

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

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

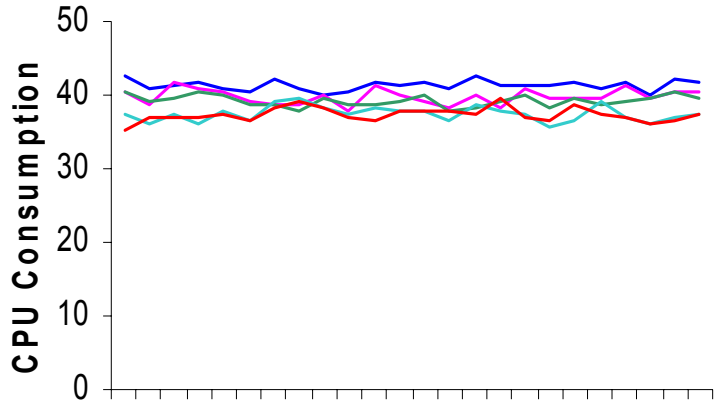
# How To Detect the Constrained Loop When No Real Work Is Present

- The good news
  - These don't really matter, do they?
- The bad news
  - They might get in the way of something that does, some day in the future
  - Correlation coefficients DO NOT WORK when you search for these
  - I want to find all Constrained Loops!
- How do we notice them?
  - If two significant processes total consumption lies within a band of two standard deviations of another process's consumption for enough hours in a day, it usually is a Constrained Loop When No Real Work Is Present

# How To Detect the Constrained Loop When No Real Work Is Present

- Given:
  - $\mu_n^{cpu}$ = computed mean of the CPU consumed by a unique process *n* during the hours where both processes existed
- Remember you must re-compute $\mu_1^{cpu}$, $\mu_2^{cpu}$, $\sigma_{p1cpu}$ and $\sigma_{p2cpu}$ each time for each pair, including only the hours where both members of the pair qualified as loops

If ($p_1^{cpu}$ >= ( $\mu_2^{cpu}$ - 2$\sigma_{p2cpu}$ ) and ($p_1^{cpu}$ <= ( $\mu_2^{cpu}$ + 2$\sigma_{p2cpu}$ ) then {you probably have a suspect}



# Loop Wrap Up

- Loops are by far the most common process or workload pathology
- Loops are easy to find automatically!
  - Check the paper for the gritty details, known issues and formulas
- Automatically finding and ticketing loops on all your system will probably cut your performance investigation requests by 2/3rds!
  - You can use all that free time to write CMG papers on how you did it!

# Finding Ramps

- We have not yet found a killer ramp finder method that works with no human involvement
  - We do have a sneaky way to detect CPU ramps that works if you give it a clue
    - Clue: This workload should never exceed X% when working properly
    - Ramps matter when they get big
    - You will trigger when they get X% big
      - If ($p^{cpu}$ >= threshold) {…notify someone!}
    - If the same application sends tickets on 100 nodes a day, someone might fix it!
  - If CPU Ramps run long enough, they become loops!

# Finding Bumpy Ramps

- I am convinced that there is a way to automatically find Bumpy Ramps, and we need it bad
  - Most real ramps are bumpy ramps
    - The periods of negative slope tend to foil most math we've tried
  - Memory leaks are everywhere
    - A memory leak in java is defined as leaks that leak faster than java's built in garbage collection can de-allocate memory
- Maybe you will find the killer formula!
  - Until you do, (and publish) try workload thresholds

# Real World Ramp Examples

- From Wednesday of this week:
  - An AIX Memory Leak
  - A CPU Ramp
  - A Nightmare Node
  - sliudv25 process memory plateau issue
- If you are not looking, you won't see them, but you will have them
- java is often another word for "memory leak"
- So is …

# Join The Hunt!

- There are plenty of pathologies out there just waiting for our attention
  - Bumpy Ramps
  - The Single Threaded Multi-process Application Loop
  - The Multi-Threaded Single Process Application Loop
  - Deadly embrace lock detectors
  - And many more…

# Join The Hunt!

- Publish your algorithms!
  - I'll help test them
  - I'll coauthor a paper with you
    - If we are lucky, this will turn into a panel!
- Note to vendors
  - Please put this functionality in your products!

# Questions?

ron.kaminski@safeway.com

for paper pdf files, example code,
example web reports, etc.