

**Chapter
2**

**APPLICATION
PERFORMANCE**

*Get on the
Fast Track!*



TM

**SYS-ED/
Computer
Education
Techniques, Inc.**

TM

Objectives

You will learn:

- Why software fails.
- Improperly constrained input.
- Improperly constrained stored data.
- Improperly constrained computation.
- Improperly constrained output.
- Performance tuning.
- The tuning cycle.

1 Why Software Fails?

There are four classes of software failures:

- Improperly Constrained input.
- Improperly constrained stored data.
- Improperly constrained computation.
- Improperly constrained output.

1.1 Improperly Constrained Input

```
/*shellsort: sort v[0]...v[n-1] in order*/
void shellsort(int v[], int n)
{
    int gap, i, j, temp;
    for (gap = n/2, gap > 0; gap /= 2)
        for (i = gap; i < n; i++){
            for (j=i-gap; j>=0 && v[j]>v[j+gap]; j-=gap){
                temp = v[j];
                v[j] = v[j+gap];
                v[j+gap] = temp;
            }
        }
}
```

The function shellsort() will be called with two arguments; an array v[] of integers and a number n representing the number of elements in the array v[]. An error condition will arise if the parameters are not consistent. In order to expose the error condition, the routine will be passed a “bad” parameter. There are two choices to choose from.

The following driver code will cause the shellsort to include the array length as one of the values in the array.

```
int main(void)
{
    int in[] = {10, 20, 30, 40};
    int length = 5;
    shellsort(in, length);
}
```

The output will be {5, 10, 20, 30, 40}. Such incorrect behavior can be difficult to detect since the program does not crash.

The shellsort can also be passed an invalid array pointer; the program will fail due to a null pointer exception:

```
int main(void)
{
    int *in = main;
    int length = 20;
    shellsort(in, length);
}
```

Prevention of broken input constraints doesn't necessarily mean including large numbers of source code with "if" statements to check value ranges. This can be achieved by tweaking the user interface to properly filter user input. As an alternative, a single routine can be used which will validate inputs and thereby help the source code remain clean.

1.2 Improperly Constrained Stored Data

Keeping bad input out of a software product is only one aspect of preventing bad data. Sometimes bad data is stored as a result of internal processing; causing the software to corrupt its own stored data.

There is little testing technology for checking constraints on internally stored data and finding input sequences that violate them. To do so, models of sequences must be constructed based on the relationship of inputs and the knowledge of how data is stored in the system.

To model the relationship between stored data and input sequences, testers will create state-transition diagrams that describe how data flows through the system. The problem is that few systematic approaches to building enumerating software states and controlling the size of the state diagram exist.

1.3 Improperly Constrained Computation

The key to finding improperly constrained computation is knowledge of the problem domain.

In the absence of good domain expertise, it also helps to have the source code available. A simple search through the source looking for places where operators or built-in system functions are used can serve to expose what computation takes place. The results which can then be generated are then studied with the objective being to identify the situations that are not supported.

1.4 Improperly Constrained Output

Sometimes software developers will get all three of the above situations right and then fail to correctly display or transmit responses to users.

Locating broken output constraints is similar to finding broken input and data constraints. The task is difficult because it is not always evident from looking at a display panel where the application can have the errors. What will be needed to be done is to drive each output to its maximum and minimum value and vary the length and character sets as much as possible and then check each result.

2 Performance Tuning

Performance tuning is the main activity associated with performance management. At its most basic level, tuning consists of finding and eliminating bottlenecks.

Before starting the performance tuning cycle, it will be necessary to establish a framework for ongoing

Performance Tuning Activities	Explanation
Identifying constraints	Business case determines priorities, which in turn establish boundaries. Performance work must be focused that are not constrained.
Specifying the load	Involves determining what services the site's clients require and the level of demand for those services. The most common metrics for specifying load are the number of clients, client think time, and load distribution.
Setting performance goals	Performance goals must be explicit, which involves identifying the metrics used for tuning as well as their corresponding benchmark values. Total system throughput and response time are two common metrics used to measure performance.

Performance and capacity are closely related; logically therefore the constraints, load, and goals are also applicable to capacity planning.

Once the boundaries and expectations for performance tuning have been established, the tuning cycle can begin. The tuning cycle will be an iterative series of controlled performance experiments.

3 Tuning Cycle

The four phases of the tuning cycle are repeated until the established performance goals are met.

Phase	Explanation
Collection Phase	<p>Starting point of any tuning exercise. Performance counters will be chosen for a specific part of the system and used for gathering data. Common counters which could be used would be network, server, or the back-end database.</p> <p>A baseline measurement will be required. A pattern of system behavior will need to be established when the system is idling as well as when the system is executing specific tasks. The baseline establishes the typical counter values that are expected when the system is behaving satisfactorily.</p>
Analyzing Phase	<p>Data will need to be analyzed to determine bottlenecks. It is also common for problems in one system component to result from problems in another component.</p> <p>Common starting points for interpreting counter values and eliminating false or misleading data are:</p> <ul style="list-style-type: none"> • Monitoring processes of the same name • Monitoring several threads • Intermittent spikes in data values • Monitoring over an extended period • Excluding start-up events • Zero values or missing data
Configuration Phase	<p>After collecting the data and completing the analysis of the results, it can be determined which parts of the system are the best candidate for a configuration change and implement this change. The cardinal rule for implementing changes is implement only one configuration change at a time</p>
Testing Phase	<p>After implementing a configuration change, the appropriate level of testing should be performed to determine the impact of the change on the system that are being tuned.</p> <p>The objective will be to determine whether or not the change:</p> <ul style="list-style-type: none"> • Improved performance • Degraded performance • Had no impact on performance <p>If performance improves to the anticipated level, the tuning cycle should be stopped. If not, one must step through the tuning cycle again.</p>

When testing:

- Check the correctness and performance of the application that you are using for testing by looking for memory leaks and inordinate delays in response to client requests.
- Ensure that all tests are working correctly.
- Make sure you can repeat all tests by using the same transaction mix and the same clients generating the same load.