1550
Accelerating Simulations
Using Efficient Modeling Techniques
Lab 1: Simulation Kernel Performance

Overview
How fast can a simulation run? Let’s establish the simulation “speed limit” for your machine. See how fast the OPNET Simulation Kernel can process very simple events, establishing the best-case simulation performance for the machine at hand.

In this lab, you will compare the run time performance of the OPNET Development and Optimized simulation kernels. You will also learn how to configure the simulation environment for efficient execution.

Objectives
• Run simulations with different configurations
• Compare the running times
• Learn the configuration settings for the best performance
• Run multiple simulations concurrently
• Investigate model compilation preferences

Instructions
Improving Simulation Performance
Let’s establish the simulation “speed limit” using different compilation and simulation configurations. Vary the optimization setting and kernel type. See the overhead of function stack trace recording and simulation status monitoring.

Minimal_self is a simple model that processes self-scheduled events. It is about the simplest thing a simulation can do. This establishes the underlying “speed limit” of the simulation kernel.

Base-Line Simulation with Development Kernel
1. Start OPNET Modeler®.
2. Open project efficiency_techniques:
   a. Choose File / Open or press keys Ctrl-O.
   b. Click folder op_models on the left.
   c. Select efficiency_techniques.
   d. Click Open.
   The minimal_self scenario is active.
3. Run simulation:
   a. Choose **DES / Run Discrete Event Simulation** or press keys **Ctrl-Shift-R**.

4. Observe run-time performance:
   a. Observe the simulation speed graph on the **Simulation Speed** page.

   b. When the simulation completes, record **Average Speed** in the Performance Summary table at the end of this lab on page 5 for the Development Kernel.

5. Click **Close**.

**Extending the Speed Limit**

6. Duplicate the scenario:
   a. Choose **Scenario / Duplicate Scenario**… or press keys **Ctrl-Shift-D**.
   b. Name the new scenario **minimal_self_optimized**.

7. Configure simulation kernel:
   a. Click the **Configure/Run DES** toolbar button or press keys **Ctrl-R**.
   b. Set **Simulation Kernel** to **Optimized**.
8. Reduce frequency of status updates for optimized use:
   a. Set **Update interval** to **10000000** events (ten million).

9. Set to recompile models without function stack trace information:
   a. Expand the tree view to **Execution / Advanced / Compilation**.
   b. Check ☑️ **Force model recompilation**.
   c. Select **Kernel-based** for **Include function stack trace information in recompiled models**.
      This will let the Development kernel include function stack trace information and the opposite for the Optimized kernel.

10. Minimize simulation status updates:
    a. Expand the tree view to **Runtime Displays / Memory Usage**.
    b. Uncheck ☐ **Graph total memory usage during simulation**.
    c. Uncheck ☐ **Show memory statistics panel**.
    d. Expand the tree view to **Runtime Displays / Other Displays**.
    e. Uncheck ☐ **Graph event processing speed**.
    f. Uncheck ☐ **Display live statistics panel**.
    g. Other options may be left checked.

11. Run simulation:
    a. Click **Run**.

12. Observe run-time performance:
    a. When the simulation completes, record **Average Speed** in the Performance Summary table at the end of this lab below.

13. Clean up:
    a. Click **Close** to finish.

---

**Getting the Most Out of Dual-Cores**

Let’s execute multiple simulations to take advantage of the dual-core CPU that we have.

14. Configure simulation environment for concurrent runs:
    a. Choose menu **Edit / Preferences** or press keys Ctrl-Alt-P.
    b. Expand tree to **All / Discrete Event Simulation / Configuration**.
    c. Enable preference **Allow Simulation Runs on Multiple Hosts** if not already so.
d. Click on the value of preference **Distributed Simulation Hosts**. Enter **localhost::2**. This preference lists the hosts on which to execute simulation runs. Since we only want to maximize the CPU, we just specify the local host.

When changing the preference, be sure to select the row labeled `<empty>` and then click on **Insert** before typing.

e. Click **OK** to save the preference changes.

15. Execute Development and Optimized simulations at the same time:
   a. Choose menu **Scenarios / Manage Scenarios**.
   b. For scenarios **minimal_self** and **minimal_self_optimized**, click in the **Results** column.
   c. Select `<recollect>`.
d. Click **OK** to start the simulation runs. 
Watch both simulations launch and execute concurrently. If you take a look at the current CPU utilization in Windows Task Manager, you’ll notice that the CPU is 100% busy. You may need to scroll the window to the right. 
Do the simulation speeds match the prior results? Can you explain why?

16. Clean up:
   a. Click **Close** to close the **DES Execution Manager**.
   b. Leave OPNET Modeler open.

**Review of Simulation Kernel Performance**

Fill in the following table with the simulation performance from the results above.

<table>
<thead>
<tr>
<th>Kernel Type</th>
<th>Average Speed (events/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td></td>
</tr>
<tr>
<td>Optimized</td>
<td></td>
</tr>
</tbody>
</table>

Using the optimized kernel and minimizing the debugging information improves simulation performance. The optimized kernel is much faster than the development kernel. If you recompile the models with the optimized kernel setting, the compiler uses a different set of compiler flags compared to the development kernel settings. This typically includes compiler optimizations.

Removing the function stack trace information (**FIN/FOUT/FRET** overhead) no longer includes the function stack in error reports for enter/exit execs of process models and user-defined functions. You need to recompile the models for this to take effect.

Simulation progress graphs also incur some overhead that is not necessary for production-grade simulations. Reduce the frequency of basic progress updates as well.

OPNET Modeler can execute simulations concurrently on any number of hosts and processors. Do so to maximize the hardware at hand for parametric simulations.
[OPTIONAL EXPLORATION]

OPNET Compilation Preferences

If you have time, explore the OPNET compilation environment preferences.

Viewing Compilation Preferences

17. Start OPNET Modeler.
18. Choose Edit / Preferences.
19. In the tree view, expand to preference group All / Discrete Event Simulation / Code Generation. Review the preferences listed below.
20. Click Cancel when finished reviewing.

The following preferences affect the OPNET compilation environment:

- Compiler selection: C Compilation Script and C++ Compilation Script
  - Windows
    - comp_msvc: Microsoft Visual C/C++
  - Linux
    - comp_unix, comp_cc, or comp_gcc: GCC C compiler
    - comp_unix_cpp, comp_CC, or comp_g++: GCC C++ compiler
- Compiler target selection
  - Compile for {32,64}-Bit {Development, Optimized} {Sequential, Parallel}
- Compiler flags
  - Compilation Flags for 32-Bit Code
  - Compilation Flags for 64-Bit Code
  - Compilation Flags for All Code
  - Compilation Flags for C Code
  - Compilation Flags for C++ Code
  - Compilation Flags for Development Code
  - Compilation Flags for Optimized Code
  - Compilation Flags for Parallel Code
- Linker selection: Network Repositories Linking Script and Static Simulation Linking Script
  - Windows
- bind_so_msvc
- bind_msvc
- Linux
  - bind_so_unix, bind_so_gcc
  - bind_unix, bind_gcc
- Linker options
  - {32, 64}-Bit {Network,Static} Repositories Flags
  - {32, 64}-Bit {Network,Static} Repository Libraries
  - Common {Network,Static} Repositories Flags
  - Common {Network,Static} Repository Libraries
  - {Development, Optimized} {Network,Static} Repositories Flags
  - {Development, Optimized} {Network,Static} Repository Libraries

**Conclusion**

Improving the performance of a simulation involves several factors:

- Configure the simulation environment
- Eliminate debugging support
- Compile models with optimizations

OPNET Modeler 16.0 makes this configuration easy to do from the GUI. For further speed improvement, you can avoid the overhead of the OPNET GUI by running the simulation from the OPNET Console (shell command-line).

END OF LAB 1
Lab 2: Improving Code Efficiency

Overview

Your task is to improve the simulation run time efficiency of a simple router model. Where is your simulation spending time? Use the OPNET Profiling tool to evaluate simulation run time performance and pinpoint the performance bottlenecks. Explore different ways to improve the running time.

In this lab, you will evaluate the baseline performance of a routing protocol model and will improve the code efficiency by using different algorithms. You will also learn how to use the OPNET Profiler to investigate performance details of a simulation.

Objectives

- Learn how to use the OPNET Profiler user interface.
- Understand the profiler output.
- Iterate through the profiling/optimization process.
- Tune model code for efficiency.

Instructions

Router Overview

The node model consists of a central queue with sets of receivers and transmitters. The process model reads periodic “hello” packets from other nodes. The “hello” messages contain routing databases containing neighbor information. The nodes compute a Bellman-Ford-like shortest path algorithm to obtain the global routing picture.

Initial Model Performance

Let’s first evaluate the current simulation’s efficiency to verify optimizations.

1. Start OPNET Modeler if it is not already open.
2. Open project efficiency_techniques if not already open:
   a. Choose File / Open or press keys Ctrl-O.
   b. Click folder op_models on the left.
   c. Select efficiency_techniques.
   d. Click Open.
3. Switch to random_mesh scenario:
   a. Press keys Ctrl-2 or choose Scenarios / Switch to Scenario / random_mesh.
4. Configure efficient simulation environment:
   a. Click on the **Configure/Run DES** toolbar button or press keys **Ctrl-R**.
   b. Set **Duration** to 3000 seconds.
   c. Set **Simulation Kernel** to **Optimized**.
   d. Set **Update interval** to 500000 events.
   e. Expand the tree view to **Execution / Advanced / Compilation**.
   f. Check **Force model recompilation**.
   g. Select **Kernel-based** for Include function stack trace information in recompiled models. This will let the Development kernel include function stack trace information and the opposite for the Optimized kernel.
   h. Expand the tree view to **Execution / Profiling**.
   i. Make sure **Collect profiling information** is not checked.
   j. Expand the tree view to **Runtime Displays / Memory Usage**.
   k. Uncheck **Graph total memory usage during simulation**.
   l. Uncheck **Show memory statistics panel**.
   m. Expand the tree view to **Runtime Displays / Other Displays**.
   n. Uncheck **Graph event processing speed**.
   o. Uncheck **Display live statistics panel**.

5. Run simulation:
   a. Click **Run**.

6. Observe run-time performance:
   a. When the simulation completes, record **Average Speed** and **Elapsed Time** in the table at the end of this lab on page 23.

7. Click **Close** to finish.

**Reference Simulation Results**

Let’s look at statistics collected during simulation to have a reference for comparison later after changing the model.

8. Configure statistics results:
   a. Choose **DES / Results / Find Top Statistics…**
   b. Select **Link Statistics / point-to-point / throughput (bits/sec)**.
   c. Click **Find Top Results**.
9. View graph:
   a. Set the graph filter under Stacked Statistics from As Is to adder
   b. Click Graph.

![Graph showing top objects and throughput]

10. Clean up, without closing the graph displayed at the previous step:
    a. Close Top Objects: point-to-point.throughput (bits/sec).
    b. Click Close in Select Statistic for Top Results.

This graph shows the total aggregate throughput on the top 10 links in the network. This graph will serve as a reference for valid results. As we make changes to the model, compare the results against this reference.

**Initial Profile of Model Performance**

Let’s see if we can improve the performance. First, find out where we are spending time in the model.

**Configuring Simulations with Profiling**

11. Configure simulation duration and kernel for shorter simulation with profiling:
    a. Press keys Ctrl-R. Bring focus to project editor window, if necessary.
    b. Set Duration to 10 seconds for shorter runs.
       As we have discovered, the Development kernel is slower than the Optimized kernel.
    c. Set Update interval to 1000 events.
d. Set Simulation Kernel to Development. Profiling of Kernel Procedures is only available with the Development Kernel.

12. Enable profiling in models:
   a. Expand the tree view to Execution / Profiling.
   b. Check ☑ Collect profiling information.
   c. Check ☑ Force model recompilation. We have to recompile all the models in the simulation so that the models use the profiling version of the FIN macro.
   d. Check ☑ Include profiling information in recompiled models.
   e. Check ☑ Enable section profiling of process models.
   f. Select Output to: Simulation progress window.

13. Run simulation:
   a. Click Run.

**Analyzing Profiler Data**

14. View profiler output:
   a. When the simulation completes, activate the Profiling page.

   ![Profiler Output](image)

   b. If necessary, resize the window to view profiling results.
c. If not already sorted as such, click on the **Total Time** column heading to sort in descending order. The actual times will vary depending on the performance of the host machine.

<table>
<thead>
<tr>
<th>Name</th>
<th>Model</th>
<th>Total Time (s)</th>
<th># Calls</th>
<th>Function Time (s)</th>
<th>Child Time/Model</th>
<th>Child Time/Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eff_router</code></td>
<td>✓</td>
<td>25.477</td>
<td>3.526</td>
<td>5.586</td>
<td>0.032</td>
<td>1.7458</td>
</tr>
<tr>
<td><code>eff_router[Process_Hello]</code></td>
<td>✓</td>
<td>21.981</td>
<td>2.473</td>
<td>8.586</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td><code>eff_router_hello_packet_process</code></td>
<td>✓</td>
<td>21.981</td>
<td>2.473</td>
<td>8.586</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td><code>eff_router_hello_data_process</code></td>
<td>✓</td>
<td>21.981</td>
<td>2.473</td>
<td>8.586</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td><code>eff_router_db_send</code></td>
<td>✓</td>
<td>10.466</td>
<td>135.536</td>
<td>136</td>
<td>0.062</td>
<td>18.908</td>
</tr>
<tr>
<td><code>pro_log_column_add</code></td>
<td>✓</td>
<td>10.131</td>
<td>5.548</td>
<td>5.548</td>
<td>2</td>
<td>10.131</td>
</tr>
<tr>
<td><code>pro_log_column_add</code></td>
<td>✓</td>
<td>9.039</td>
<td>5.555</td>
<td>5.555</td>
<td>2</td>
<td>9.039</td>
</tr>
<tr>
<td><code>eff_router_hello_packet_process</code></td>
<td>✓</td>
<td>1.447</td>
<td>135.536</td>
<td>136</td>
<td>0.059</td>
<td>1.952</td>
</tr>
<tr>
<td><code>eff_router_hello_packet_process</code></td>
<td>✓</td>
<td>1.447</td>
<td>135.536</td>
<td>136</td>
<td>0.059</td>
<td>1.952</td>
</tr>
<tr>
<td><code>eff_router_hello_packet_process</code></td>
<td>✓</td>
<td>1.261</td>
<td>2.473</td>
<td>2.473</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td><code>eff_router_hello_packet_process</code></td>
<td>✓</td>
<td>1.261</td>
<td>2.473</td>
<td>2.473</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td><code>eff_router_hello_data_process</code></td>
<td>✓</td>
<td>1.173</td>
<td>2.510</td>
<td>2.510</td>
<td>0.055</td>
<td>0.055</td>
</tr>
<tr>
<td><code>eff_router_hello_db_send</code></td>
<td>✓</td>
<td>0.633</td>
<td>2.573</td>
<td>3.039</td>
<td>0.032</td>
<td>0.032</td>
</tr>
<tr>
<td><code>pro_log_column_add</code></td>
<td>✓</td>
<td>0.453</td>
<td>9.000</td>
<td>5.000</td>
<td>2</td>
<td>0.000</td>
</tr>
<tr>
<td><code>eff_hello_entry_create</code></td>
<td>✓</td>
<td>0.302</td>
<td>150.436</td>
<td>2</td>
<td>0.302</td>
<td></td>
</tr>
<tr>
<td><code>eff_hello_entry_create</code></td>
<td>✓</td>
<td>0.302</td>
<td>150.436</td>
<td>2</td>
<td>0.302</td>
<td></td>
</tr>
<tr>
<td><code>eff_hello_data_entry_add</code></td>
<td>✓</td>
<td>0.261</td>
<td>141.339</td>
<td>2</td>
<td>0.251</td>
<td>0.251</td>
</tr>
<tr>
<td><code>eff_hello_entry_destroy</code></td>
<td>✓</td>
<td>0.261</td>
<td>141.339</td>
<td>2</td>
<td>0.251</td>
<td>0.251</td>
</tr>
<tr>
<td><code>eff_hello_data_entry_exit</code></td>
<td>✓</td>
<td>0.261</td>
<td>141.339</td>
<td>2</td>
<td>0.251</td>
<td>0.251</td>
</tr>
<tr>
<td><code>pro_log_column_add</code></td>
<td>✓</td>
<td>0.261</td>
<td>147.194</td>
<td>2</td>
<td>0.260</td>
<td>0.260</td>
</tr>
<tr>
<td><code>pro_log_column_add</code></td>
<td>✓</td>
<td>0.261</td>
<td>147.194</td>
<td>2</td>
<td>0.260</td>
<td></td>
</tr>
<tr>
<td><code>eff_hello_entry_create</code></td>
<td>✓</td>
<td>0.261</td>
<td>147.194</td>
<td>2</td>
<td>0.260</td>
<td></td>
</tr>
<tr>
<td><code>eff_hello_data_entry_add</code></td>
<td>✓</td>
<td>0.261</td>
<td>147.194</td>
<td>2</td>
<td>0.260</td>
<td></td>
</tr>
<tr>
<td><code>eff_hello_entry_destroy</code></td>
<td>✓</td>
<td>0.261</td>
<td>147.194</td>
<td>2</td>
<td>0.260</td>
<td></td>
</tr>
<tr>
<td><code>eff_hello_data_entry_exit</code></td>
<td>✓</td>
<td>0.261</td>
<td>147.194</td>
<td>2</td>
<td>0.260</td>
<td></td>
</tr>
<tr>
<td><code>pro_log_column_add</code></td>
<td>✓</td>
<td>0.055</td>
<td>5.850</td>
<td>0.017</td>
<td>0.0121</td>
<td>0.000</td>
</tr>
<tr>
<td><code>pro_log_column_add</code></td>
<td>✓</td>
<td>0.055</td>
<td>5.850</td>
<td>0.017</td>
<td>0.0121</td>
<td></td>
</tr>
</tbody>
</table>

- **i.** Knowledge of call-graph from code is helpful here.
- **ii.** Process `eff_router` consumes the bulk of the running time.
- **iii.** Most of that time is spent in its `Process_Hello` enter executives.
- **iv.** A lot of time is spent in `eff_router_hello_data_process()`, which is called from `eff_router_hello_packet_process()`.
- **v.** What is the breakdown of `eff_router_hello_data_process()`? How much time is in the function itself compared to children calls?

Let’s try to find where in `eff_router_hello_data_process()` we are spending time.
Profiling Detailed Sections of Functions

Normally, you would want to save the output of the profile run for later comparison. You can do so by clicking on the Export… button. We’ll skip that step here.

15. Click Close.

16. Open the eff_router process model:
   a. In the Project Editor, double-click on any node. (They all have the same node model.)
   b. Double-click on the eff_queue module in the center.

17. Locate offending code:
   a. Click on the Edit Function Block toolbar button or press key F7.
   b. Press keys Ctrl-F and search for hello_data_process to locate function eff_router_hello_data_process(), or press keys Ctrl-, (control and comma) and go to line 221.
c. Note the `for()` loop, a likely source of inefficiencies.

Macros `PROFILE_SECTION_BEGIN` and `PROFILE_SECTION_END` define a section of code to be profiled separately from the function. This helps to identify areas of an expensive function to narrow the optimization candidate.

Profile this function in three areas: top, middle, and bottom. To save time, we have already added the necessary code to profile the three sections. Remove the surrounding comments as directed below.

18. Enable top profile section:
   a. Remove the comment delimiters for the profile section after the `for()` statement (line 237) and after the `op_ima_obj_attr_get()` call (line 250). Note that the BEGIN and END arguments match exactly.

   ```
   HELLO_SIZE = HELLO_DATA_TOTAL_GET (HELLO_DATA_PTR);
   for (hello_index = 0; hello_index < hello_size; hello_index++)
       PROFILE_SECTION_BEGIN (eff_router_hello_data_process_top);
   hello_entry_ptr = eff_hello_data_entry_access (hello_data_ptr, hello_index);
   /* This hello entry tells is that it costs a certain cost to reach a */
   /* certain destination via src_node_id. So we must first find the */
   /* destination in our internal database and see if the current cost */
   /* to reach that destination is better than his cost. Remember to */
   /* add in the cost of getting to src_node in the first place. */
   /* Ignore destination entries which are "myself" */
   op_ima_obj_attr_get (op.topo.parent.src_id.self (), "my_id", &my_id);
   PROFILE_SECTION_END (eff_router_hello_data_process_top);
   ```

19. Enable middle profile section:
   a. Remove the comments for another profile section around the call to `eff_router_neighbor_cost_get()` (line 259).

   ```
   PROFILE_SECTION_BEGIN (eff_router_hello_data_process_middle);
   total_cost = hello_entry_ptr->cost + eff_router_neighbor_cost_get (src_node_id);
   PROFILE_SECTION_END (eff_router_hello_data_process_middle);
   ```
20. Enable bottom profile section:
   a. Remove the comments for the bottom profile section before
      `eff_router_db_find()` (line 265) and just before the end of the `for()` loop block (line 288).

```
PROFILE_SECTION_BEGIN (eff_router_hello_data_process_bottom);

  eff_database_entry_ptr = eff_router_db_find (hello_entry_ptr->destination_id);
  if (eff_database_entry_ptr)
  {
    if (total_cost < eff_database_entry_ptr->cost)
      /* If we discovered a cheaper way of getting to */
      /* the destination, we need to replace the */
      /* information with updated data. */
      eff_database_entry_ptr->cost = total_cost;
      eff_database_entry_ptr->next_hop_node_id = src_node_id;
      eff_database_entry_ptr->destination_id = hello_entry_ptr->destination_id;
      eff_database_entry_ptr->update_time = sp_sim_time();
      eff_database_entry_ptr->sender_timestamp = timestamp;
  }
  else
    /* We had not known how to get to the destination before, */
    /* so add a new entry now. */
    eff_router_db_add (hello_entry_ptr->destination_id, src_node_id, total_cost,
                       timestamp);

PROFILE_SECTION_END (eff_router_hello_data_process_bottom);
```

21. Save the model:
   a. Press keys Ctrl-S to save the function block code changes. Then close the editor.
   b. Press keys Ctrl-S again to save the process model in the Process Editor.

22. Check for errors by compiling code:
   b. Select Yes for Include function call stack support.
   c. Check ✔ ... with profiling information.
   d. Check ✔ ... and section profiling.
   e. Click Compile to make sure there are no errors in the code.
   Let’s ignore any warnings for now. You will know there are only warnings and no errors if the
   Compilation of process model eff_router box shows the done with warnings status.
23. Go back to the Project Editor:
   a. Choose **Windows / Project / efficiency_techniques**.

24. Configure simulation kernel:
   a. Press keys **Ctrl-R**.
   b. Set **Duration** to 5 seconds to make the run finish more quickly.
   c. Expand the tree view to **Execution / Advanced / Compilation**.
   d. Uncheck **Force model recompilation** to avoid unnecessary model compilation since we already compiled the changed process model above.

25. Run simulation:
   a. Click **Run**.

26. View the profiler output by clicking on the **Profiling** tab.
The largest component of the time spent in `eff_router_hello_data_process()` is in the bottom section. (How do we know that?) Another round of PROFILE_SECTION can isolate the problem within the bottom section. However, `eff_router_db_find()` consumes a lot of run time, so it is the likely culprit. Note also that it is called frequently.

27. Analyze `eff_router_db_find()`:

a. Do not close the profiler output window yet.

b. Go back to the `eff_router` Process Editor.

c. Click the Edit Function Block toolbar button or press key F7.

d. Press keys Ctrl-F and search for “db_find” to locate function `eff_router_db_find()` (line 354).

   i. This is a simple function with a for() loop and calls to `op_prg_list_access()` for searching the destination node ID.

   ii. Though `op_prg_list_access()` is simple, frequent execution contributes to the large Total Time spent in the `eff_router_db_find()` function.
iii. What is the Big-O of the function?

e. Scroll up to review the code for eff_router_db_add().
   
The use of data structures there matches the linked list used in the search function.

28. See profile of list accesses:
   
a. Press keys Ctrl-W to close the code editor.

b. Go back to the Project Editor by choosing Windows / Project / efficiency_techniques.

c. In the profiler output window that is still open, note the Child Time / Kernel for eff_router_db_find().

d. Also note the Number of Calls to op_prg_list_access() and op_prg_list_size(), the majority of which are probably called from eff_router_db_find(). Be aware that op_prg_list_access() and op_prg_list_size() will probably be displayed without the op_ prefix because they are actually macros defined to be prg_list_access() and prg_list_size(), respectively.

Linear searching through a linked list is O(n). Possible performance improvements include:

- Trees – O(log n) searching.
- Hash tables – O(1) searching.

Let’s try to improve the performance by using another searching algorithm.
Tuning the Code

Searching through a list for a particular matching element is O(n). Searching items in a hash table is O(1). Since searching is a frequent operation here, let’s try to use a hash table implementation for the destination node IDs to see if there is any performance improvement. Hash tables are appropriate in this context because we want an algorithm and data structure that is efficient at searching but is good at insertions as well.

Using a Better Searching Algorithm

29. Locate code to modify:
   a. Close the profiler output window if it is still open.
   b. Choose Windows / Process Model / eff_router.
   c. Go back to eff_router_hello_data_process() in the function block of the eff_router process model (line 221).

We have started to define a new function to replace eff_router_db_find(). This new function uses hash tables. It is a good idea to leave the original code around in case the new code does not yield any performance improvements.

30. Rework the searching design:
   a. Replace the call to eff_router_db_find() in the bottom section with eff_router_db_hash_find()(line 265).
   b. Also replace the call to eff_router_db_add() with eff_router_db_hash_add() in the same profiling section (line 284).
   c. Press keys Ctrl-F and search for other instances of calls to eff_router_db_add() or eff_router_db_find(). Except for the function definitions themselves, replace the calls with eff_router_db_hash_add() or eff_router_db_hash_find().
31. Define hash table searching code:
   a. At the very bottom of the function block, review the code for functions
      `eff_router_db_hash_add()` and `eff_router_db_hash_find()`.
   
      ```c
      eff_router_db_hash_add() and eff_router_db_hash_find().
      ```
   
   b. In `eff_router_db_hash_add()`, remove the #if guards to revive the dead code.
      ```c
      In eff_router_db_hash_add(), we have replaced op_prg_list_insert() with
      prg_bin_hash_table_item_insert(). Note that the first argument is now
      SV_database_hash for the new state variable data structure. Binary hash tables require a
      key in the form of a memory address. We’ve directly passed the address of
      destination_id for the key.
      ```
   
   c. Also in `eff_router_db_hash_find()`, remove the #if guards to revive the dead code.
      ```c
      We’ve replaced the for() loop entirely with a call to
      prg_bin_hash_table_item_get(). This replaces a sequential search with a hash table
      search.
      ```

32. Enable other hash table searching-related changes:
   a. To save time, we have already changed other parts of the model that depend on the router
      database data structure. Uncomment the #define at the beginning of the function block to
      enable these changes.

      ```c
      #define EFF_USE_HASH_DB
      ```

   b. Press Ctrl-S to save the function block code changes.
   
   c. Press Ctrl-W to close the function block editor.

33. Review binary hash table initialization:
   a. Open the Enter Executives of the Init state, by clicking on the top half of the Init state.
   
   b. Note the arguments to `prg_bin_hash_table_create()`. Why did we not have to
      pass the length of the integer key to `prg_bin_hash_table_item_insert()` or
      `prg_bin_hash_table_item_get()`?
   
   c. Close the Enter Executives window.

34. Save the model:
   a. In the Process Editor, press Ctrl-S to save the process model.

35. Check for syntax errors by compiling code:
a. Click the **Compile Process Model** toolbar button to make sure there are no errors in the code.

b. Click **Close**.

c. Press **Ctrl-W** to close the process model.

36. Go back to Project Editor:
   a. Choose **Windows / Project / efficiency_techniques**.

37. Configure simulation kernel for production to compare against initial run (since this is the true test to see if performance has improved):
   a. Press keys **Ctrl-R**. Bring focus to the project editor, if necessary.
   b. Set **Duration** to **3000** seconds.
   c. Set **Simulation Kernel** to **Optimized**.
   d. Set **Update interval** to **500000** events.
   e. Expand the tree view to **Execution / Advanced / Compilation**.
   f. Check ☑ **Force model recompilation**.
   g. Select **Kernel-based** for **Include function stack trace information in recompiled models**.
   h. Expand the tree view to **Execution / Profiling**.
   i. Uncheck ☐ **Collect profiling information**.

38. Run simulation:
   a. Click **Run**.

39. Observe run-time performance:
   a. When the simulation completes, record **Average Speed** and **Elapsed Time** in the table at the end of this lab on page 23.
   b. Click **Close** to finish.

**Validate Simulation Results**

Let’s look at statistics collected during simulation to validate that our changes did not introduce errors.

40. Configure statistics results:
   a. Choose **DES / Results / Find Top Statistics**…
   b. Select **Link Statistics / point-to-point / throughput (bits/sec)**.
   c. Click **Find Top Results**.

41. View graph:
a. Set the graph filter under Stacked Statistics from As Is to adder
b. Click Graph.
c. Compare against the reference graph obtained on page 10.
d. Seek help from a Teaching Assistant if the results differ.

42. Clean up:
   a. Close Top Objects: point-to-point.throughput (bits/sec).
   b. Click Close in Select Statistic for Top Results.

**Profiling Current Performance**

Let’s profile current performance to see if the code changes we have made resulted in actual improvements.

43. Configure simulation duration and kernel:
   a. Press keys Ctrl-R.
   b. Set Duration to 10 seconds for quicker runs.
   c. Set Update interval to 1000 events.
   d. Set Simulation Kernel to Development.

44. Enable profiling in models:
   a. Expand the tree view to Execution / Advanced / Compilation.
   b. Check Force model recompilation.
   c. Select Kernel-based for Include function stack trace information in recompiled models.
   d. Expand the tree view to Execution / Profiling.
   e. Check Collect profiling information.
   f. Check Force model recompilation.
   g. Check Include profiling information in recompiled models.
   h. Check Enable section profiling of process models.
   i. Select Output to: Simulation progress window.

45. Run simulation:
   a. Click Run.

46. View profile results.
   a. What parts of the model now consume the most amount of time?
   b. Click Close when finished reviewing results.

47. Leave OPNET Modeler open.
**Review of Code Tuning Improvements**

Fill in the following table with the event speed and the total running time of the simulations from the results above.

<table>
<thead>
<tr>
<th>Improvement</th>
<th>Events/Second</th>
<th>Elapsed Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Table</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Conclusion**

The run-time profile of a simulation using the built-in OPNET Profiler yields valuable information for improving the efficiency of a model. Profiling entails simple configuration and recompilation of the models. OPNET Modeler 16.0 makes this configuration easy to do from the GUI.

Improving the efficiency of the user code is not trivial. It requires careful analysis of the profile data using principles learned from this OPNETWORK session. Improving the run time involves repeated iterations of profiling and code tuning. If the performance does not improve after code tuning, of course, undo that change. It is important to verify the correctness of the simulation by comparing the results between the iterations. You do not want to introduce unintentional errors into the code.

Based on the profiling results, target the “expensive” parts of the code first (think 80/20). Avoid fine-tuned platform-specific optimizations unless absolutely necessary. They are generally not portable and are hard to maintain over time. Better algorithms and appropriate data structures often yield the desired performance improvements.

**END OF LAB 2**
Lab 3: Memory Statistics

Overview
Memory is often the limiting factor when it comes to performance issues. Simulations can consume quite a lot of memory. In this lab, you will learn how to profile the model’s memory usage using the OPNET memory API and memstats tools.

In a “leaky” variant of the router model, it is difficult to track the memory leak without using the OPNET Memory API. Find out where the memory leak is using the memory profiling tools in OPNET 16.0.

Objectives
• Learn how to use the memory statistics tools
• Understand the memstats output

Instructions
Tracking Memory Use in OPNET

Configuring and Running Simulations with Memory Profiling
1. Start OPNET Modeler if it is not already open.
2. Open project efficiency_techniques if not already open:
   a. Choose File / Open or press keys Ctrl-O.
   b. Click folder op_models on the left.
   c. Select efficiency_techniques.
   d. Click Open.
3. Switch to random_mesh_leaks scenario:
   a. Press keys Ctrl-3 or choose Scenarios / Switch to Scenario / random_mesh_leaks.
4. Configure memory profile output at end of simulation:
   a. Press keys Ctrl-R.
   b. Set Duration to 1000 seconds.
   c. Set Simulation Kernel to Development.
   d. Set Update interval to 10000 events.
   e. Expand the tree view to Runtime Displays / Memory Usage.
f. Check ☑ **Graph total memory usage during simulation.**

g. Check ☑ **Show memory statistics panel.**

h. Check ☑ **Generate final usage data once simulation ends.**

5. Run simulation:

   a. Review steps 6 to 7 below before starting the simulation. You will perform actions while the simulation runs.

   b. Click on **Run** when ready.

6. Immediately, check memory profile during simulation run:

   a. As soon as the simulation enters stage “Beginning Simulation” and the progress bar updates itself, click on the **Memory Stats** tab.

   b. Click on **Get Latest Data.**

   c. Observe the memory profile statistics table.

7. Observe memory leak:

   a. Click on the **Memory Usage** tab.

   b. This graph shows the real-time memory usage as the simulation proceeds.

   c. Observe the steady increase in memory usage.
8. View simulation termination memory profile:
   a. At the end of simulation, memory statistics appear on the Memory Stats page.
   b. Resize the Simulation Progress dialog box to make the memory statistics table better visible.
   c. Check **Show detailed data** for additional statistics columns.
   d. Check **Show differences** for the difference in statistics between the last sample point and the simulation end.
   e. Uncheck **Show differences**.

9. View memory profiling results:
   - **Type**: Type of OPNET Memory (Pooled, Categorized, general, or other).
   - **Name**: Name of the memory category.
   - **Model**: Whether category is owned by OPNET internally or by models.
   - **Bytes**: Number of bytes for the pooled memory object.
   - **Use (#)**: Number of memory objects in active use.
10. Observe lack of useful debugging information:
   a. Click on the **Use (KB)** column heading to sort the data by the total number of bytes.
      Note that **Shared cleared (24) bytes pool** is the largest memory category. We do not have any further breakdown of where this memory actually resides. Due to advanced memory optimizations, many different memory categories share a single entry.

11. Click **Close**.

12. Do not close the project.

It is difficult to diagnose a memory leak without model-specific information. The next section will remedy that.

**OPNET Pooled and Categorized Memory**

13. Switch to random_mesh_leaks_pmo scenario:
   a. Press keys **Ctrl-↓** or choose **Scenarios / Switch to Scenario / random_mesh_leaks_pmo**.

14. Disable OPNET memory optimization preferences:
   a. Choose **Edit / Preferences** or press keys **Ctrl-Alt-P**.
   b. In the tree view, expand to preference group **All / Miscellaneous**.
   c. Search for **mem_opt** preference.
   d. For the following preferences, set the values to **FALSE**.
      i. **mem_opt.compact_pools**
      ii. **mem_opt.pool_small_blocks**
   e. Click **OK**.
   f. Click **OK** for the preference change warning message.

   Since we want to disable advanced memory optimizations in simulation, which runs in a separate process, it is not necessary to restart Modeler. These preferences enable advanced
memory optimizations that make detailed analysis difficult during development and debugging. Enable them again when done with analysis.

15. Configure memory profile output:
   a. Press keys Ctrl-R.
   b. Set **Duration** to 100 seconds to prevent the memory leak from taking over the computer.
   c. Set **Update interval** to 5000 events.
   d. Set **Simulation Kernel** to Development. Memory Source tracing is only available in the Sequential Development Kernel. Memory Statistics, however, is available in the Optimized Kernel as well.
   e. Expand the tree view to **Runtime Displays / Memory Usage**.
   f. Check ✔ **Graph total memory usage during simulation**.
   g. Check ✔ **Show memory statistics panel**.
   h. Check and select ✔ **Data is generated when generating progress reports**.
   i. Check ✔ **Generate final usage data once simulation ends**.

16. Configure memory source tracking to identify source of leak:
   a. Check ✔ **Generate memory source tracing information**.
   b. Select ☑ **For all memory categories**.

17. Run simulation:
   a. Review step 18 below before starting the simulation.
   b. Click **Run**.

18. View simulation memory profile:
   a. Let the simulation run for a little bit, then click **Pause**. Memory source data is not available at the end of simulation, so let’s pause it to investigate.
   b. Click on the **Memory Stats** tab.
   c. Resize the **Simulation Progress** dialog box to make the memory statistics table better visible.
   d. Click on the **Use (KB)** column heading to sort the data by the total number of bytes used.
Note that there are two new rows for hello entry and hello data that are "Model" memory categories. It is likely that hello entry objects are leaking.

Other memory categories are not as large as they were.

The categories Sim Memory Source tracking overhead and memory source tracking cell are memory categories used internally for the Memory Sources feature. Do not worry about them.

19. View source of memory leak:
   a. Click on the Memory Sources tab.
   b. Click Update to get the latest data.
   c. Click on table column header Bytes to see the largest memory client.
   d. Check Inverse function call stack to organize the function call stack by the functions doing the allocation.
   e. Expand the largest function by allocation bytes.
   f. Select op_prg_pmo_alloc (pmh) / eff_hello_entry_create / eff_router_db_to_hello_transfer / eff_router_hello_send / eff_router_leaky_pmo [Send_Hello enter execs].
g. Click **Show Details**.

- A Memory blocks window appears, listing each block of memory allocated at this function call stack.

h. Close the Memory blocks window when finished reviewing memory results.

20. Stop simulation since we’re done with the analysis:

a. Click **Stop** in Simulation Progress.
b. Click Close.

Finding and fixing the actual cause of the leak can be done as homework.

**Using the OPNET Memory API**

Let’s review the code changes made to use the OPNET pooled memory API in our router model.

21. Open the external source code editor.
   a. In any editor, choose **File / Open** or press keys **Ctrl-O**.
   b. Set **Files of type** to **External Source (C code) Files (*.ex.c)**.
   c. Click folder **op_models** on the left.
   d. Select **eff_router_sup_pmo_ref.ex.c**, which is an External File of process model **eff_router_leaky_pmo**.

22. View OPNET Memory API use.
   a. View function **eff_hello_entry_create()** on line 174 by pressing keys **Ctrl-**.

```
  void
  eff_hello_entry_add (eff_hello_data *hello_data_pcr, eff_hello_entry *hello_entry);

  if (hello_data_pcr) {
    printf("hello_entry: %s\n", hello_entry->name);
  }
```

   o This function calls **op_prp_pmo_define()** once to set up the pooled memory handle. The memory is then registered with name “hello entry”. We can identify this memory category in the memory statistics output for debugging.
- Every time the pooled memory API needs to extend the memory pool with a new block of objects, it will do so in 1000-object increments.
- Allocations simply use the handle via `op_prm_pmo_alloc()`.

23. Close the editor:
   a. Choose **File / Close**.
   b. Leave OPNET Modeler open.

**Conclusion**

The OPNET Memory Statistics output aids debugging memory utilization issues in simulations. The Memory Stats updates in the Simulation Progress execution dialog box provide a convenient user interface for viewing the memory profiling statistics output. You can investigate the primary memory consumers by using the OPNET Memory API’s pooled and categorized memory objects. You can also obtain more detailed memory profiles using the OPNET Simulation Debugger (ODB) and the `memstats` command in a live simulation. The ODB `mtag` family of commands help track down memory leaks in the model. See Sessions 1502 and 1503 for details on using the OPNET Simulation Debugger.

Memory source tracing consumes a significant degree of memory and performance overhead. Use the feature in a targeted fashion instead of tracing all memory categories.

Using the tools learned in this lab, you can diagnose memory utilization issues in your simulations. After identifying candidates for memory optimization, use the principles from this session to reduce memory consumption of the model.

**END OF LAB 3**
Lab 4: Simulating Protocols Efficiently

Overview

In this lab you will run OSPF over a 200-node IP network. You will then exercise various efficiency methods to increase simulation efficiency. You will enable efficiency mechanisms already present in the OPNET Model Library as well as reconfiguring the models to remove any unnecessary overhead. Finally, use of pre-computed routing tables will illustrate more advanced efficiency methods.

Objectives

- Understand OSPF simulation efficiency to eliminate control traffic
- Use pre-computed routing tables from Flow Analysis
- Identify irrelevant components to reduce simulation overheads

Instructions

Baseline Analysis

200-node OSPF network

1. Start OPNET Modeler if it is not already open.
2. Open the lab project:
   a. Choose File / Open or press keys Ctrl-O.
   b. Click folder op_models on the left.
   c. Select operation_efficiency.
   d. Click Open.
   e. Observe the scenario 200_nodes_ospf_routing to see a network similar to the following:
3. Observe protocol configuration:
   a. Select View / Visualize Protocol Configuration / IP Routing Protocols / IPv4 Routing Protocols or press keys Ctrl-Shift-V to see that OSPF is configured on all interfaces of all routers
   b. Select View / Visualize Protocol Configuration / Clear Visualization or press keys Ctrl-Shift-C.

All routers are configured to start OSPF routing between 5 and 10 seconds. OSPF will start its operation, exchange keep-alive messages and routing updates, and compute routing tables until the end of the simulation. There is no other application traffic configured in this network.

4. Run simulation:
   a. Press Ctrl-Shift-R.
      The simulation takes several minutes to complete. It simulates 5 hours using the Optimized kernel.

5. Observe initial performance:
   a. After the simulation terminates, fill in the Original entry for the Performance Summary table at the end of this lab on page 40 from the Messages and Memory Usage tabs.
   b. Click Close in Simulation Progress.

6. View OSPF messaging overhead:
a. Click the **Hide/Show Graph Panels** toolbar button.

b. Select menu **DES / Panel Operations / Panel Templates / Load with Latest Results**.

c. Right-click on each graph, select **Edit Graph Properties**, and set the **Vertical max** to **300**.

d. Click **OK**.

e. Notice that the nodes are constantly exchanging OSPF traffic throughout simulation duration.

7. **Analysis of traffic results:**

   Every pair of routers exchanges keep-alive messages. This is used to detect link and node failure/recovery.

   - If there are no failure/recovery events in a simulation, keep-alive messages can be disabled for two reasons
     
     i. They are not important for a stable network after routing tables converge.
     
     ii. They do not contribute a lot to the network traffic and hence their absence should not affect network performance.

   - However, in a mobile network, keep-alive messages should not be disabled as they are used to detect routing changes due to mobility. Based on speed of node movement, the frequency of exchange can be reduced.

8. Close the graph panel window. Do not Delete it but **Hide** it instead for later review.

**Baseline Results for Correctness**

The routing tables built during a simulation are used for forwarding data plane traffic. This network does not have any filters and policies; hence all nodes should have reachability to all other nodes in
the network. It is important to verify that any changes or optimizations to the models do not alter the correctness of results.

9. Select any node in the network; find its IP address and subnet mask:
   a. Right-click on any node.
   b. Select Edit Attributes.
   c. Type ip address in the Filter box and search for the Address attribute using Exact match mode. Expand IP/IP Routing Parameters/Interface Information. You will see a list of interfaces.
   d. Find IP Address ___________________ and subnet mask __________________ of any interface. You can do so by expanding the attribute hierarchy under that interface.
   e. Click Cancel to close all attribute dialog boxes.

10. Analyze network reachability results to have a basis for comparison after later model changes:
   a. Choose Protocols / IP / Routing / Reachability Analysis / Using DES Forwarding Tables…
   b. Select the Display tab.
   c. Enter the IP Address and mask as prefix length from step 9.d above (e.g., 192.0.215.1/24). Use prefix length here instead of subnet mask. 24 is the prefix length to represent subnet mask of 255.255.255.0.
   d. Click Display.
   e. Verify that all nodes can reach the destination network.

   Note: Please refer to the figure below. Nodes belonging to the target network are identified with a little orange and black symbol. Other nodes can reach the network if they have a green arrow pointing away to another node.
1550 Accelerating Simulations Using Efficient Modeling Techniques

Eliminating Control Traffic

**OSPF Simulation Efficiency**

11. Duplicate the scenario for new analysis:
   
   a. Choose Scenarios/Duplicate Scenario or press keys Ctrl-Shift-D.
   
   b. Name the new scenario 200_nodes_ospf_efficiency.

12. Configure simulation for OSPF efficiency mode:
   
   a. Press keys Ctrl-R.
   
   b. Expand tree view to Inputs / Global Attributes.
   
   c. Expand to Global Attribute Simulation Efficiency / OSPF Sim Efficiency.
   
   d. Set the attribute value to Enabled.
   
   e. Note the value of attribute OSPF Stop Time.

Global simulation attribute **OSPF Sim Efficiency** allows you to limit the period during which OSPF generates traffic. During the initial period OSPF behaves normally, sending packets in order to construct its routing tables. However, after **OSPF Stop Time** seconds, it assumes that the routes have stabilized and further assumes that no changes occur in the network topology. The existing routing tables at that time remain in use for the remainder of the simulation.
Validating Efficiency Improvements

**View Simulation Efficiency Improvements**

13. Run simulation:
   a. Click Run.

14. Observe optimized performance:
   a. After the simulation terminates, fill in the OSPF Efficiency entry for the Performance Summary table at the end of this lab on page 40 from the Messages and Memory Usage tabs.
   b. Click Close in Simulation Progress.

15. View OSPF traffic results:
   a. Click the Hide/Show Graph Panels toolbar button.
   b. Choose DES / Panel Operations/Reload Data Into All Panels.
   c. Right-click on the panel and select Edit Panel Properties.
   d. Set Horizontal min to 0s.
   e. Set Horizontal max to 300s.
   f. Click OK.
   g. Set the maximum vertical scale to 300 as we did earlier.
   o Note that the traffic levels goes down to zero after the OSPF stop time. (What was this value?)
   o There is no other traffic for the remaining simulation duration.
   o What are the trade-offs?
   o How to choose the protocol stop time?
   o What kind of events can be stopped?

16. Validate network reachability results to verify that the optimizations did not alter the model significantly:
Repeat steps 9 and 10 from page 36 to validate this simulation run.

Notes:

- Simulation completed successfully without change in results.
- There are improvements in simulation execution speed.
- Run simulation with one efficiency change at a time to assess effectiveness.

[OPTIONAL]

Use of Pre-Computed Data

Another method of increasing a model’s abstraction is to use pre-computed data. In this section, we will use pre-computed routing tables from IP Flow Analysis.

IP forwarding tables control the paths taken by IP data traffic. OPNET allows import of IP Flow Analysis tables into DES runs. Since IP Flow Analysis itself can import routing tables from actual routers, bringing the IP Flow Analysis results into DES permits simulation of application traffic using real-world routing tables. The resulting operational analysis does not have to worry about the accuracy of any simulated routes in the network.

Importing Flow Analysis Routing Tables

17. Duplicate the original scenario for new analysis:
   a. Go to the “baseline” scenario by pressing keys Ctrl-1 or choose Scenarios / Switch To Scenario / 200_nodes_ospf_routing.
   b. Choose Scenarios / Duplicate Scenario or press keys Ctrl-Shift-D.
   c. Name the new scenario 200_nodes_import_flan_routing.

18. Configure IP Flow Analysis:
   a. Choose Flow Analysis / Configure/Run Flow Analysis… or press keys Ctrl-Alt-F.
   b. Select the Reports tab.
   c. Click Select Reports…
   d. In the Quick Select section on the right, check IP forwarding table report.
   e. You may uncheck the other reports.
   f. Click OK to close the reports chooser.

19. Run Flow Analysis:
   a. Click Run.
   b. Click Close when Flow Analysis completes.
      The forwarding tables are now ready.
20. Configure simulation to use pre-computed routing tables:
   a. Press keys Ctrl-R.
   b. In the tree view, expand to Inputs / Global Attributes.
   c. Expand to Global Attribute IP / IP Routing Table Export/Import.
   d. Click the Value column for that attribute and select Import.
   e. Note that the IP Routing Table Source is set to Flow Analysis.

21. Run simulation:
   a. Click Run.

22. Observe optimized performance:
   a. After the simulation terminates, fill in the Routing Table Import entry for the Performance Summary table below (on page 40) from the Messages and Memory Usage tab.

23. Click Close in Simulation Progress.
   We should validate reachability results to verify that the use of Flow Analysis routing tables did not alter the simulation. However, we skip the validation to save time here.

24. Exit OPNET Modeler unless you intend to do the optional lab 5.

**Review of Model Optimizations**

Fill in the following table with the simulation performance from the results in this lab.

<table>
<thead>
<tr>
<th>Improvement</th>
<th>Events/Second</th>
<th>Running Time (seconds)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSPF Efficiency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Routing Table Import</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The running time is more than cut in half by a simple tradeoff of fidelity for efficiency, stopping the detailed modeling of OSPF after 260 seconds.

Combining Flow Analysis with DES simulation can reduce the running time of the DES part even more, since pre-built routing tables are utilized, and there is no detailed modeling of the OSPF protocol at all.

**Conclusion**

You have seen various techniques that can be used while operating simulations using OPNET standard models. Many of these techniques can be extended to your own protocol models too. Explore the implementation in the model code.

**END OF LAB 4**
Lab 5: Model Optimization

Overview
The router network of Lab 2 is still taking too long to simulate. Let’s investigate alternate model design in hopes that performance can be improved. Often, the largest performance gain comes not from line-by-line code optimization but from an overall algorithm redesign.

Objectives
• Assess routing model’s operating environment
• Understand caching algorithm
• Verify performance improvement

Instructions
To review the models from Lab 2, the router network is a mesh of routers that exchange distance-vector routing messages at regular intervals. The “Hello” messages contain routing databases. The nodes maintain a list of direct neighbors with which to exchange information. Each node keeps an internal database of <destination, cost, next hop> entries per destination node. Over time, as the databases grow larger, the “Hello” messages also grow larger. The general routing protocol algorithm generates the “Hello” message and broadcasts it to neighbors. When a “Hello” message arrives from a neighbor, the internal database gets updated with new information (the best cost to destination nodes). In Lab 2, we identified a performance bottleneck in searching for destination nodes in the internal database. Using a hash table yielded a significant performance boost over doing linear searches.

Most of the running time is spent in processing the “hello” messages (eff_router_hello_data_process). Can we reduce the number of “hellos”?

Suppose that the topology does not change frequently. (This is true for a stable network without mobility or outages.) The router’s database remains constant. The hello message data, though constructed, does not change from before. Thus, there is no need to repeatedly perform the expensive computation of processing the data. It is sufficient to “remember” the results of the previous computation by reusing the last Hello data. How can we reduce the number of hello packets received?—Do not send the hello packet for the receiver to process in the first place.

Let’s design a caching algorithm for both the sender and the receiver of the Hello messages to see if we can get any extra performance out of the routing protocol model.
Obtain baseline performance

1. Start OPNET Modeler if it is not already open.
2. Open project efficiency_techniques if not already open:
   a. Choose File / Open or press keys Ctrl-O.
   b. Click folder op_models on the left.
   c. Select efficiency_techniques.
   d. Click Open.
3. Switch to the random_mesh scenario which you optimized using hash tables in Lab 2:
   a. Press keys Ctrl-2 or chose Scenarios / Switch to Scenario / random_mesh.
   b. Double-click on any node.
   c. Right-click on queue module eff_queue.
   d. Choose Edit Attributes in the pop-up menu.
   e. Click on the Value column of attribute process model.
   f. Select eff_router_hash_ref.
   g. Click on OK.
   h. Press keys Ctrl-S to save the node model.
   i. Go back to the Project Editor.
4. Configure simulation to benchmark performance:
   a. Press keys Ctrl-R.
   b. Set Duration to 3000 seconds.
   c. Set Simulation Kernel to Optimized.
   d. Set Update interval to 500000 events.
   e. Expand the tree view to Execution / Advanced / Compilation.
   f. Check Force model recompilation.
   g. Select Kernel-based for Include function stack trace information in recompiled models.
   h. Expand the tree view to Execution / Profiling.
   i. Uncheck Collect profiling information.
5. Run simulation:
   a. Click Run.
6. Observe run-time performance:
   a. When the simulation completes, record **Average Speed** and **Elapsed Time** for Hash Table at the end of this lab.

**Cache Design**

Let’s split the cache design into two parts: the sender and receiver.

The sender can remember when its own database last changed. It can also remember when the last set of “Hello” messages was disseminated. If the database has not changed, send a signal to neighbors indicating that is the case (e.g. an empty data message). This will side step the time to package up the database information into a “Hello” message.

The receiver can keep track of the last non-trivial “Hello” message from each neighbor. If a neighbor sends an empty message (signaling that the database has not changed since the previous update), reuse the cached routing data message. The rough memory cost for this cache is [number of neighbor entries] x [size of largest “Hello” data].

Fortunately, to save time, we have implemented this for you. Let’s review the performance improvements.

**Assess Performance with Caching**

7. Change queue module implementation.
   a. Go back to the Project Editor.
   b. Double-click on any node.
   c. Right-click on queue module `eff_queue`.
   d. Choose **Edit Attributes** in the pop-up menu.
   e. Click on the Value column of attribute **process model**.
   f. Select `eff_router_hello_cache`.
   g. Click on **OK**.
   h. Press keys **Ctrl-S** to save the node model.
   i. Go back to the Project Editor.

8. Run simulation:
   a. Press keys **Ctrl-Shift-R**.
   a. When the simulation completes, record **Average Speed** and **Time Elapsed** for Hello Cache in the table at the end of this lab.
   b. Click on Close.

**Caching Implementation**

If you have time, let’s review the changes we have made for this performance improvement.

10. Open function block of process model.
    a. In the Project Editor, double-click on any node.
    b. Double-click on the `eff_queue` module in the center to open process model `eff_router_hello_cache`.
    c. View the Function Block.

11. Sender-side implementation.
    a. The sender records the last time the database was modified in `eff_router_hello_data_process()` (lines 324 & 336).
b. The sender also records the last time a "hello" was sent in `eff_router_hello_send()` (line 161).

def eff_hello_data_create():
    """We now want to fill in the hello data with ""
    """"information from our internal database."
    data_size = eff_router_db_to_hello_transfer(hello_data_ptr);
    sv_last_hello_send_time = op_sim_time();
    sv_last_hello_size = data_size;

    use_cache = (sv_last_db_update_time < sv_last_hello_send_time);
d. Send an empty packet to let the receiver know (line 148).

```c
/* If we use a cached hello data packet, then we can */
/* reduce the time spend constructing the hello packet */
/* by simply duplicating the existing cached hello */
/* data. */
if (use_cache) {
    hello_data_ptr = NULL;
    data_size = sv_last_packet_size;
} else {
```

12. Receiver-side implementation in `eff_router_hello_packet_process()` (line 237).

   a. Receiver caches the last “hello” from each neighbor.

   b. When the sender signals that the “hello” information has not changed by sending an empty packet, retrieve the cached message and process it.
13. Clean up:
   a. Close all open windows.
   b. Exit OPNET Modeler.
Review of Code Tuning Improvements

Fill in the following table with the event speed and the total running time of the simulations from the results above.

<table>
<thead>
<tr>
<th>Improvement</th>
<th>Events/Second</th>
<th>Running Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash Table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hello Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Conclusion

Rethinking the design of the routing model, we were able to spend some extra memory (in the “Hello” message cache) and gain a significant amount of performance. In addition to using better algorithms and appropriate data structures, think about the model goals and the purpose of the analysis. Make adequate assumptions about the model’s operating environment and make compromises in model abstraction or computation. Avoiding simulation of every single real world event will improve the simulation’s run time significantly with minimal cost to the analysis results.

END OF LAB 5