



# Rogue Wave ThreadSpotter

---

## *Optimization Tutorial – In-Memory Table Lookups*

### **Executive Summary**

*Rogue Wave ThreadSpotter is a programmer's tool that analyzes an application and then presents a list of high level advice (a.k.a SlowSpots) telling the programmer where and how to change the application to improve its performance.*

*There are many different causes for poor performance, and Rogue Wave ThreadSpotter focuses on finding how the limited resources of modern multi-core processors and their memory hierarchies are used by the program. Simple changes can make a huge difference in performance and scalability.*

*In this paper we present a series of gradual changes to a sample application. The changes were prompted by advice and statistics from the tool, leading to a 46x performance boost on a single core, and more than 83x on an 8-core system, where the final version displays almost perfect linear scaling.*

## Overview

This is a tutorial showing typical scenarios and techniques when working with Rogue Wave ThreadSpotter to improve an application's performance.

In a modern computer, the memory system components contribute to a large part of the performance characteristics of an application:

- **The memory bus** is responsible for transferring data from the memory to the CPU cores, and this bus has a limited bandwidth. The more cores there are demanding data, the higher the pressure on the bus. This ultimately puts a cap on application performance.
- Inside the CPU, there are several layers of small but fast memories, known as **caches**. The caches store often used data for quick access. Since the size of the caches are small, and the penalty for not finding your data in the cache is large, the application performance is highly limited by how well it manages to utilize the caches.
- The CPU is equipped with a unit called a **prefetcher**, which is responsible for anticipating application memory accesses and populating the caches ahead of time with the data likely to be requested by the application. It inspects the application data flow and has an easier time finding patterns in the traffic when the application performs regular accesses.

**Locality** is a central concept denoting a favorable characteristic, which in this context should be interpreted as it is cheaper to access different memory locations that are close than locations that are far apart (**spatial locality**), and that it is better to revisit the same memory location sooner rather than later (**temporal locality**).

Naïve programming will often cause these resources to be sub-optimally used, and ThreadSpotter helps to pinpoint where the program could be made to run faster, by explaining how to be leaner with respect to memory bandwidth, memory latency, and cache usage and shows where the governing principles are broken.

This is done by:

- Enhancing application memory access regularity to help the hardware prefetcher.
- Enhancing spatial locality to help minimize the amount of unused data transferred between memory and cache, and to minimize the amount of cache space occupied by unused data.
- Enhancing temporal locality by suggesting ways to reuse data while it remains in the cache.
- Hide memory access latencies by adding prefetch instructions.

## Rogue Wave ThreadSpotter

**Rogue Wave ThreadSpotter** analyzes an application's interaction with the cache and the memory subsystems. It can analyze single thread and multithread code on single- and multi-cores, as well as multi-processor machines.

**Rogue Wave ThreadSpotter** also focuses on multi-threaded issues on multi-cores arising from thread interactions and communication between the cores and caches within the processor.

It consists of a few utilities:

- **A sampler**. This component spies on the application and collects information.
- **A reporter**. The collected information is analyzed and results are written to a report.
- **A graphical user interface** to set parameters for sampling and report generation
- **Command line tools to do the same thing**

In addition, a regular web browser is used to read the reports. The reports are heavily cross-linked to allow efficient navigation between statistics, advice detail and source code.

## Optimization Workflow

Optimization is an iterative process. It is a very good idea to first establish a **repeatable test environment** where the execution time can be measured. Then it is easy to try out various changes, and see their effect.

When optimizing an application, alterations to the program's source code and inherent structure will be made, and these changes can be of local or global scope. Optimization sometimes reduces legibility, maintainability, encapsulation and coherence. It may introduce redundant code and replicate or de-normalize data. It is generally a good idea to prepare for this and to agree on an acceptable level.

The next task is to get somewhat acquainted with the code. After sampling the application for the first time, spend some time looking through ThreadSpotter's reports. Briefly **go over the top items in each advice category** to see whether they reference related code sections. Usually, resolving the advice affects structure definitions and their subsequent use throughout the application, so it is worthwhile to browse around and familiarize yourself with the code in question.

There are different approaches, but one that works fairly well is to look for signs of **irregular accesses** (*random access* issues) high up in the latency issues category, and see if those can be addressed. That will increase access regularity in your program, which will help the processor to anticipate its data accesses.

Then move to look for *cache line utilization* optimization issues and *incorrect loop order* issues, which both relate to **lack of spatial locality**. Throughout the memory system, data travels in chunks, and the minimum amount that is fetched and stored in a cache is known as a cache line, typically 64 bytes. Making use of all data in a cache line is high up on the list of optimizations.

After that, look for **long-term reuse opportunities**. Rogue Wave ThreadSpotter suggests applying common reuse techniques through the *blocking* and *loop fusion* advice. These code transformations promise rewarding returns but may require more extensive changes to loop and function structure. The resulting effect is that data is reused multiple times while it is still mapped in the cache. This reduces the pressure on the memory bus.

After fixing each SlowSpot, recompile and measure the performance to see if there was any performance increase.

Then, sample the application anew, prepare new reports and restart the process.

## Labs Part

### Setting Up Your Environment

Please load the LiveDVD into the DVD-Drive of your laptop and start the machine. If the DVD-Drive is the first item in your boot list your computer will automatically boot an Ubuntu Linux operating system from the DVD. If not you may change the boot order by either creating a temporary boot menu (usually by pressing F12 during startup) or by changing your BIOS settings.

Ubuntu will ask you to choose your language first.


The LiveDVD offers the opportunity **to try Ubuntu without installation**. Please choose this option if you want to avoid any installation on your hard disk.

As a next step you will be prompted for login.

Please login as “demouser”. Your password will be “demouser” as well.

If Linux has been loaded please plug the usb memory stick into a free port. The directory on the stick will be mounted to the file system the LiveDVD has established in memory.

Open a console window and move to the memory stick’s mount point by typing:

```
demouser @ubuntu:~$ cd / media 
```

type:

```
demouser @ubuntu:/ media$ ls 
```

You will see a subdirectory which is named as your memory stick’s volume id.

Move to that subdirectory by typing:

```
demouser @ubuntu:/ media$ cd / xxxxx.xxxxx 
```

Note: **xxxx.xxxx** has to be replaced by your memory stick’s volume id

Type:

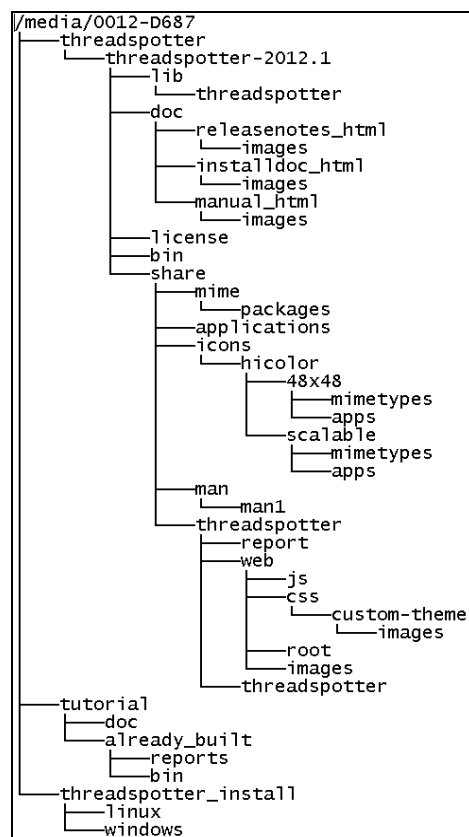
```
demouser @ubuntu:/ media/ xxxx-xxxx$ ls 
```

You will see three subdirectories and a shell script called “setup.sh”. Please source the script by typing

```
demouser @ubuntu:/ media/ xxxx-xxxx$ source / setup.sh
```

The script will set up your environment by updating the PATH variable in order to include the latest ThreadSpotter edition which is installed on the memory stick.

The memory stick's directory structure looks as follows:



- The directory called “threadspotter” contains an already installed version of the latest Rogue Wave ThreadSpotter product.
- The “threadspotter\_install” directory contains Rogue Wave ThreadSpotter installers for Linux and Windows.
- The directory “tutorial” contains source files, a makefile and some shell scripts you will need for this tutorial.

Please move to the “tutorial” directory by typing

```
demouser @ubuntu:/ media/ xxxx-xxxx$ cd tutorial
demouser @ubuntu:/ media/ xxxx-xxxx/ tutorial$
```

You have now prepared your environment for starting with the labs part.

## Example Application

The application is an example of a memory bandwidth intensive code with a host of problems in the areas outlined above.

The application models an in-memory database table, and a queue of queries against that table.

Different versions of the code implement the table using different data structures and ways to represent data and queries.

The different versions share a common part consisting of a test driver and data structures. The differences between different versions are located in the various `database_n*.hh` files.


In this tutorial, one of the purposes is to enable a detailed comparison between different source code variants. We carefully control the point where the sampler engages and disengages. This is explained in the Appendix, and the scripts in the source distribution also do this for you.

## Building the Example Application

In order to discover opportunities for performance optimization Rogue Wave ThreadSpotter samples binaries of an application. For looking up related lines in the source code ThreadSpotter needs to make use of references included in the binary's debug information. Therefore it is recommended to prepare debug builds by using the `-g` compiler option.

Building the different versions of the example application is straight forward because an already prepared makefile invokes the gcc compiler by using the correct compiler flags (`-g -O3`).

Please type

```
demouser@ubuntu:/media/0012-D687/tutorial$ make all 
```

The following binaries will be built:

**test1 test1b test1c test2 test3 test4 test4b**

(In case you encounter problems in building the examples please find already built binaries in the directory `/media/xxxx-xxxx/tutorial/already_built/bin`)

## Lab 1 – Baseline: Standard Doubly-Linked List of Records

The baseline code uses a standard C++ list template, `std::list`, to store database records.

The vital part of the original version looks like this:

```
class database_1_linked_list_t : public single_question_database_t {
public:
    virtual void ask_one_question(query_t &query) const;
private:
    typedef std::list<car_t> cars_t;
    cars_t cars;
};

void database_1_linked_list_t::ask_one_question(query_t &query) const {
    cars_t::const_iterator i = cars.begin(), e = cars.end();
    for (; i != e; i++) {
        switch (query.query_type) {
            case 0: // count matching colors
                if (i->color == query.car.color)
                    query.result++;
                break;
            case 1: // count same model but heavier than minimum weight
                if (i->model == query.car.model &&
                    i->weight > query.car.weight)
                    query.result++;
                break;
        }
    }
}
```

Please execute this version by typing

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ ./test1
```

The program will run as many complete loops as possible in 60 seconds and will print the median execution time.

**Question 1:** What is the median execution time of the program version “test1”?

In order to analyze “test1” we now will sample “test1” with ThreadSpotter.

Please type

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ sample --start-at-function start_sampling
--stop-at-function stop_sampling -o test1.smp -r /test1
```

If you look at the sampler’s output messages you will see a warning that the number of samples would not be enough for reliable results. The warning includes a recommendation for an adjusted sample period.

**Question 2:** How many samples are necessary to get a reliable report?

Please follow the sampler’s advice and start a new sample run by providing the sampler with the new sample period:

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ sample --start-at-function start_sampling  
--stop-at-function stop_sampling -s <new sample period> -o test1.smp -r /test1
```

You will now find a ready to use fingerprint file called “test1.smp” in your working directory.

**Question 3:** What is the reason for starting/ stopping sampling at functions start\_sampling/ stop\_sampling?

In order to encounter opportunities for optimizing the program with regard to the target architecture you will need to generate a report based on the sample file.

Please type

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ report -c 2m -i /test1.smp -o test1-r.tsr
```

ThreadSpotter’s report generator will create a report named “test1-r.tsr” in your working directory.

Note: By adding the `-c 2m` option to the command line we are forcing the generator to generate a report targeting a last level cache with a cache size of 2Mb. This tutorial is set up to have a footprint of that general size. If you had not added this cache size override then your system’s actual parameters would be used instead. Depending on your system’s cache size you might not get the anticipated result for this programmed tutorial.

**Question 4:** Why should you always start optimizing your application related to the highest level cache?

In order to open the report in your browser you will need to start a webserver application called “view”.

Please type

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ view -i /test1-r.tsr
```

The command above will open the report in your standard browser (in this case Firefox).

**Note:** Firefox will sometimes tend to switch to its offline mode. In case the report won’t be displayed please change the mode to “online” in the “File” menu.

The report’s first page will show that “test1” is suffering from limited bandwidth as well as latency and locality issues. This page tells you that the program has potential for some improvements.

Please open the main part of the report and open the “Issue” tab of the “Summary” window.

**Question 5:** What is the dominant issue listed under the “Latency Issues” tab?

As a next step please expand the “Statistics for instructions of this issue” as shown in the “Issue” window.

**Question 6:** What is the meaning of “Access randomness” which is very high in this case?



**Question 7:** What fundamental data structure is the cause of this problem?

Please click on the issue in the “Summary” window and have a look where it occurs in the source code.

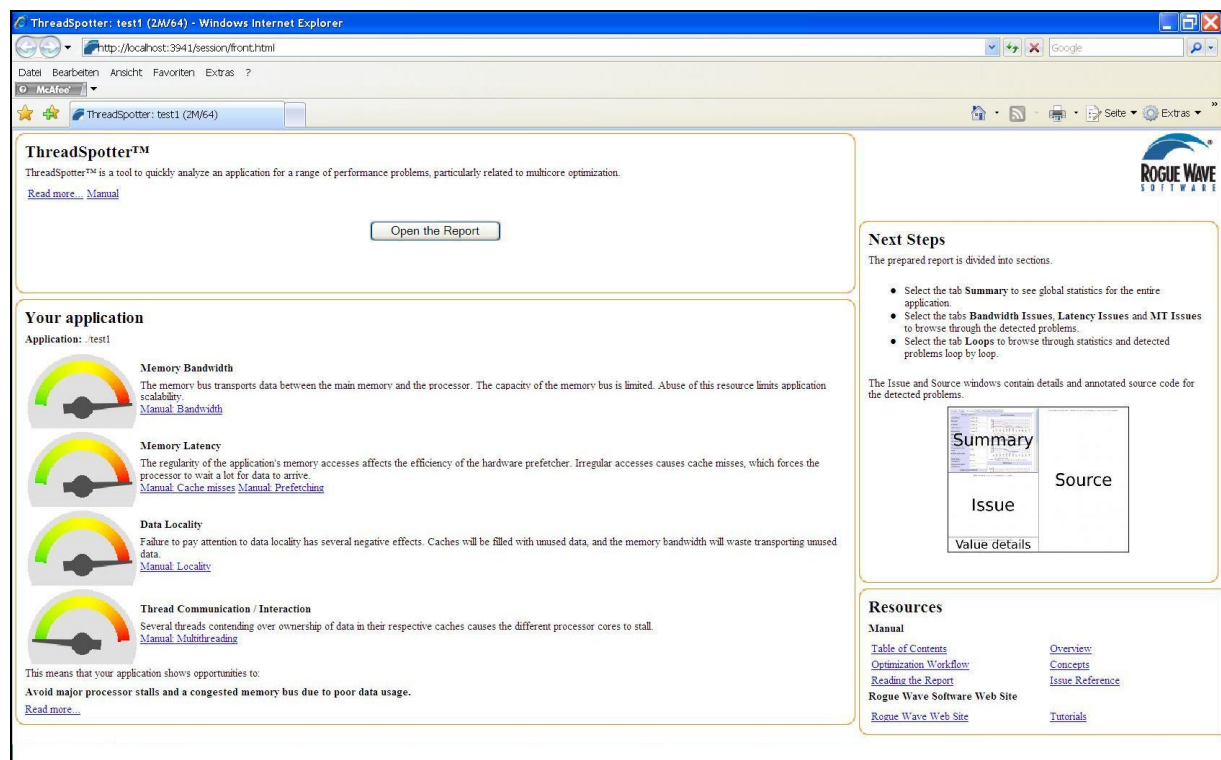
The “Miss ratio” mentioned in the “Statistics for instructions of this issue” shows a high percentage.

**Question 8:** What is the reason for the huge amount of cache misses?

**Question 9:** What is so bad about cache misses?

**Question 10:** What are the options to avoid the prefetching problem in this case?

After running and sampling the program, the following report is presented:



The first page shows that the application generally suffers from being limited by memory bandwidth, and also that it is negatively affected by memory latencies and exhibits poor data locality. This page is meant for the programmer to get an overview of the problems affecting the application, and at a glance be able to see how optimization attempts play out.

Entering the main part of the report reveals three sub-windows, which contain respectively:

- Lists of issues, loops and global information such as statistics
- Issue details, loop details
- Annotated source code

ThreadSpotter: test1 (2M/64) - Windows Internet Explorer

http://localhost:1500/session/main.html

Issues Loops Summary Files Execution About/Help

Bandwidth Issues Latency Issues Multi-Threading Issues Pollution Issues

#	Issue type	% of misses	HW-Prefetch	Randomness	Fetch utilization
8	Random access	49.9%	0.6%	Very high	11.2%
11	Spat/temp blocking	49.9%	0.6%	Very high	11.2%
7	Random access	48.3%	0.4%	Very high	11.2%
9	Spat/temp blocking	48.3%	0.4%	Very high	11.2%
4	Fetch hot-spot	1.6%	0.0%	Low	41.9%
12	Temporal blocking	1.6%	0.0%	Low	41.9%

Copyright (c) 2006-2012 Rogue Wave Software, Inc. All Rights Reserved. Patents pending.

### Issue #8: Random access

This instruction group also shows symptoms of Fetch utilization, Fetch hot-spot.

Statistics for instructions of this issue

Instructions involved in this issue

Loop statistics

Loop instructions

Copyright (c) 2006-2012 Rogue Wave Software, Inc. All Rights Reserved. Patents pending.

```

20 void database_1_linked_list_t::add_one(const car_t &c)
21 {
22     cars.push_back(c);
23 }
24 void database_1_linked_list_t::finalize_adding()
25 {
26     cars.sort();
27 }
28 void database_1_linked_list_t::ask_one_question(query_t &query) const
29 {
30     cars_t::const_iterator i = cars.begin(), e = cars.end();
31     for (; i!=e; i++) {
32         switch (query.query_type) {
33             case 0: // count matching colors
34                 if (i->color == query.car.color)
35                     query.result++;
36                 break;
37             case 1: // count same model but heavier
38                 if (i->model == query.car.model && i->weight > query.car
39                     query.result++;
40                 break;
41             }
42         }
43     }
44 }
45 #endif

```

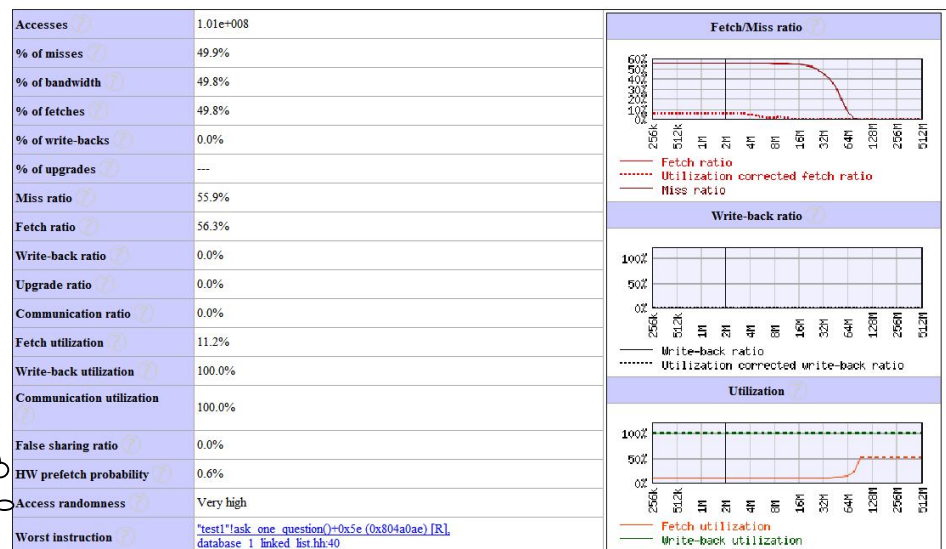
Copyright (c) 2006-2012 Rogue Wave Software, Inc. All Rights Reserved. Patents pending.

In the first version of the example program, we find that **random access patterns dominate the latency issue tab**. Clicking on the top one will focus the source code frame around one of the two query sections, and the issue (shown below) details fill up the lower left panel.

### Issue #8: Random access

This instruction group also shows symptoms of Fetch utilization, Fetch hot-spot.

#### Statistics for instructions of this issue



If the program was changed as to reach 100% fetch utilization, fetches in this instruction group would be reduced with 89.2%, and total number of fetches would be reduced with 44.4%.

#### Instructions involved in this issue

Instruction	% of misses	% of fetches	Fetch ratio	Fetch utilization	W-B Utilization
"test1"ask_one_question()+0x58 (0x804a0a8) [R], std_list.h:231	5.6%	5.6%	12.7%	11.2%	100.0%
"test1"ask_one_question()+0x5e (0x804a0ae) [R], database_1_linked_list.h:40	44.3%	44.2%	99.9%	11.2%	100.0%

In this case, the top advice in the Latency section tells you about irregularity among the memory accesses. Advice of this kind points to the accesses to the fields of the elements in the main data structure, the *std::list* members, and these items are apparently accessed in a non-contiguous, irregular way.

This is common for linked lists and other dynamic data structures. Elements are allocated dynamically and as the heap warms up it becomes fragmented. New allocations reuse free slots and this tends to spread out these elements in an unpredictable way throughout memory.

A cache is often accompanied by a unit called a hardware prefetcher. Its job is to look at the application's memory accesses and try to detect a pattern. If it finds a pattern, it will fetch data to the cache from memory just ahead of the time when that data is needed. If successful, this hides much of the memory latency.

More than anything else, it is access randomness that affects how well the hardware prefetcher will work. For random access patterns, the prefetcher will not detect any prefetchable patterns, and the core will stall while waiting for data to arrive.

Execution time, (Intel Xeon E5345): 14 seconds
--

Consequently, the statistics for the issue and for the instructions show low hardware prefetch ratios for this advice.

Optional task: The example version “test1b” uses an intrusive singly linked list instead of the *std::list* which is doubly linked. Test1b is further modified in the next section. Sample this version, generate a report and explore the difference to “test1”.

## Lab 2 – Adding Prefetch Hints

As the ThreadSpotter online help will tell you, there are two ways to deal with this situation. Either you can arrange data in memory in such a way that the hardware prefetcher can anticipate the access pattern, or add explicit prefetch instructions yourself. We will explore both ways, but first we focus on the latter suggestion.

Since the data accesses will be irregular, the hardware prefetcher will be inactive. The idea is to manually add special prefetch instructions to bring data into the cache well ahead of when it is needed. How much in advance of its use to issue this instruction is a function of how busy the processor is between the prefetch and the subsequent usage. Assuming that the latency of a memory access is 100 times slower than the CPU cycle time, one should prefetch data at least 100 cycles before it is needed.

If all you do in your loop is to traverse the list and look at a field or two, it is not enough to prefetch the next element. You need to be further ahead with prefetching, but this introduces a problem: It would seem that you need to see the preceding node to be able to find the address to the next node, and that this appears to preclude prefetching anything but the next node in the list.

The solution to this is adding an auxiliary field whose only purpose is to point to a node several steps ahead. Whenever traversing the list, in addition to operating on the data, one should prefetch the address pointed to by this field.

It turns out that adding an auxiliary field which is set up to point to a node at a proper distance isn't usually such a tricky thing. The extra space it occupies is compensated by less time spent waiting for data.

```
#define PREFETCH_DISTANCE 8

// Replacing std::list with homegrown linked list
struct node {
    struct node *next;
    struct node *prefetch_hint;
    ... // rest of fields
};

struct node *head;

// Traverse a list and populate prefetch hints to point
// 'PREFETCH_DISTANCE' steps ahead.
void prepare_prefetch_hint()
{
    struct node *q, *p;
    int distance = PREFETCH_DISTANCE;
    for (p = head; p; p = p->next)
        if (0 == distance--) break;
    for (q = head; p && q; p = p->next, q = q->next)
        q->prefetch_hint = p;
}

struct node *p;
for (p = head; p != 0; p = p->next) {
    __builtin_prefetch(p->prefetch_hint);           // gcc specific
    ... // use p-> fields
}
```

Replacing the linked list implementation in our example code with something similar to this causes the execution time for our example to drop, because more fetches can be in flight at the same time, and data is already being fetched when the instruction to consume data is encountered.

Execution time:  
6.5 seconds

Please run the third version by typing

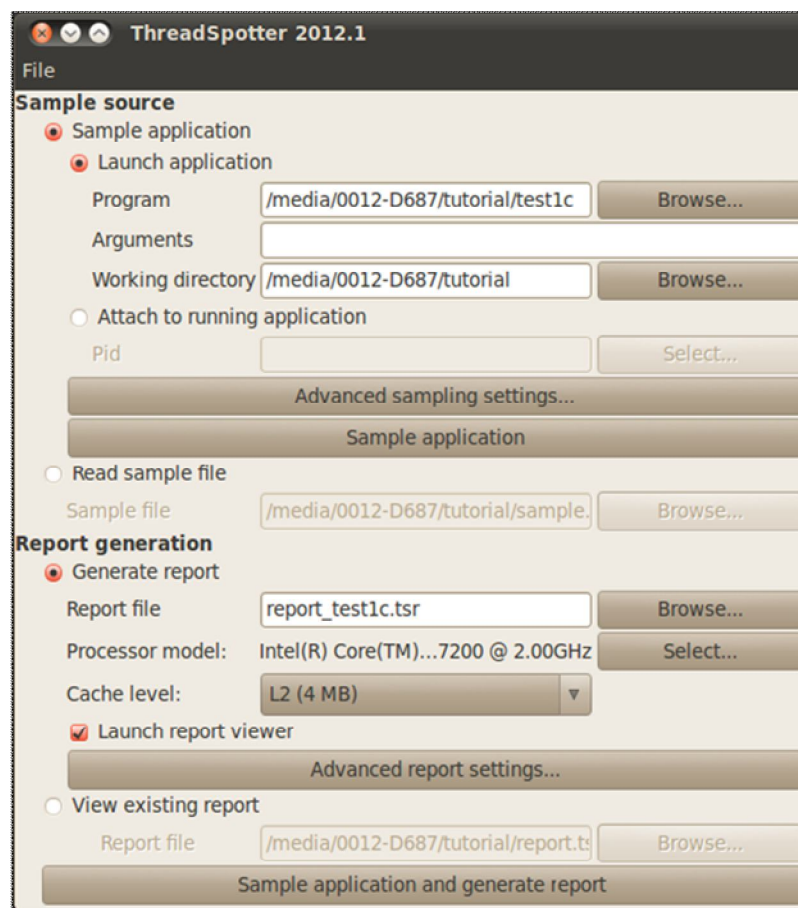
```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ ./test1c
```

**Question 1:** What is the median execution time of the program version “test1c”?

In order to analyze this program version we will invoke ThreadSpotter’s sampler and report generator via its graphical user interface (but the same results can be achieved using command line tools just like in the first lab).

Please open ThreadSpotter’s GUI by typing:

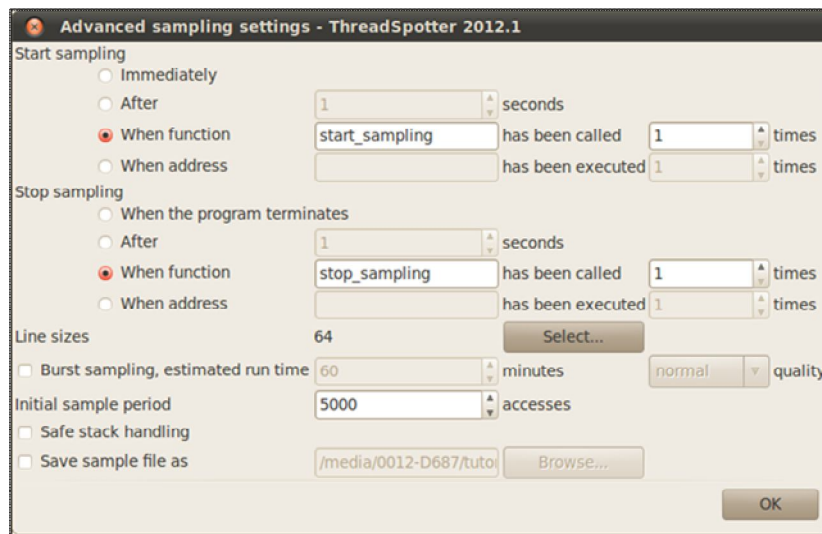
```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ threadspotter
```



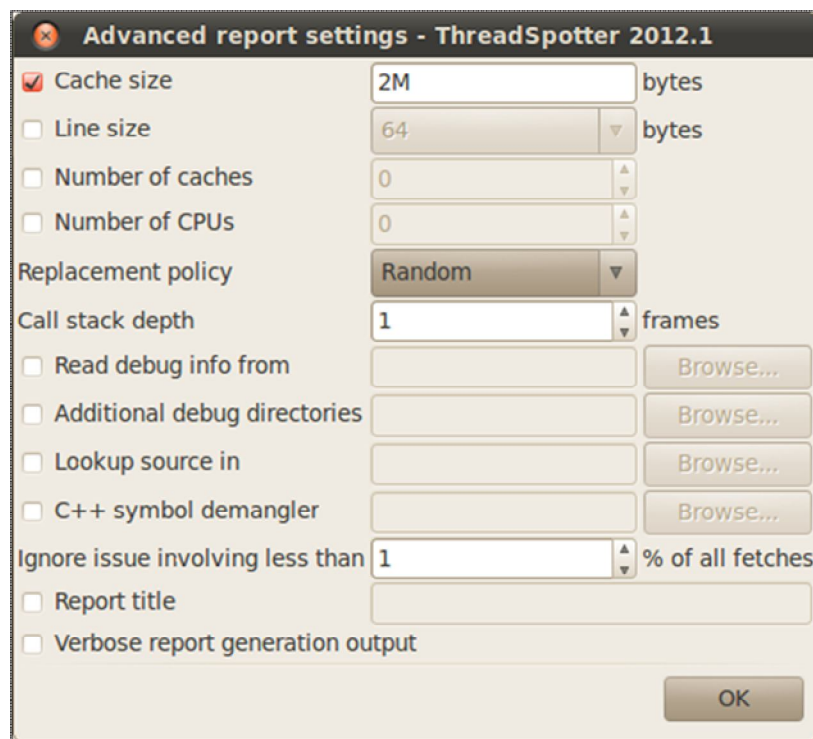
Your settings should look similar to the settings shown in the screenshot above. You will need to tell the sampler that it has to start at function “start\_sampling” and to stop at function “stop\_sampling”. Additionally it will be necessary to decrease the initial sample period as we did



when sampling “test1”. Please open the “Advanced sampling settings” window and add the parameters as shown in the screenshot below.



The last step before starting the sampler and report generator is to set the target cache size to 2 Mb as the fictitious target system’s highest level cache is limited to that size.



Pushing the button “Sample application and generate report” will do the sampling, report generation and will automatically open the report in the standard browser.

Please open the “Latency issues” tab and explore the “Random access” issue again.

**Question 2:** Is the prefetcher working efficiently now?

**Question 3:** What can be done in order to avoid the “Random access” issue?

### Lab 3 – Vector of Records

Rerunning the sampling and report generation has not changed much. The access pattern is still irregular and the hardware prefetcher is still not working well, which can be seen from the low hardware prefetch probability for the top issue. Adding these software prefetch instructions removes some of the CPU stalls (the miss rate is lower than before), but this still does not help us with fully using each cache line (the cache line utilization is still poor).

To get further we need to try a different approach. The other possible remedy suggested by the ThreadSpotter online help is to revise the data structure, possibly replace it with something denser.

Two drawbacks with linked lists are that they require extra fields to maintain the list structure, and that the spatial locality is poor. The former means that less percentage of cache space is devoted to storing useful data. The latter means that it is improbable that more than one record is used from each cache line before that cache line is evicted, since nodes are scattered throughout a large part of the memory, and that two items close in the sequence are located in the same cache line is improbable. While this can be addressed by implementing a custom dynamic memory allocator and regularly sort nodes physically in memory, this is a fragile solution.

The alternative straightforward solution is to use a contiguous storage data type, such as a plain old array or `std::vector`.

Replacing the linked list with a vector has other noteworthy implications for some cases. It is no longer as cheap to remove or insert elements in the middle of the sequence, but random access is cheap, and maybe more importantly, a linear traversal trivially engages the hardware prefetcher. Such a traversal causes cache misses on adjacent cache lines, and this is a simple pattern for the hardware prefetcher to train on. Elements are also placed consecutively and no extra housekeeping pointers are needed.

If the freedom of a linked list is required, but the dataset is traversed a lot during some phases of the execution, it is sometimes worthwhile to make a temporary copy of the elements that you are interested in, stored in a contiguous way. Then use the temporary copy to traverse the data. Throw away the temporary copy when you are done traversing.

For our case, we see a dramatic reduction in running time when replacing the linked list with a `std::vector`.

Execution time:

1.0 seconds.

Please run the version “test2” by typing

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ ./test2
```

**Question 1:** What is the median execution time of the program version “test2”?

Please let ThreadSpotter sample “test2” and generate a report. It is up to you whether you prefer to control ThreadSpotter via gui or by using the command line interface.

**Question 2:** Has the “Random access issue” been solved by substituting the linked list by a `std::vector`?

Please have a look at the issues listed under the “Bandwidth issues” tab.

**Question 3:** What dominant issue can be found on top of the issues list?

**Question 4:** Why does low fetch utilization slow down the program?

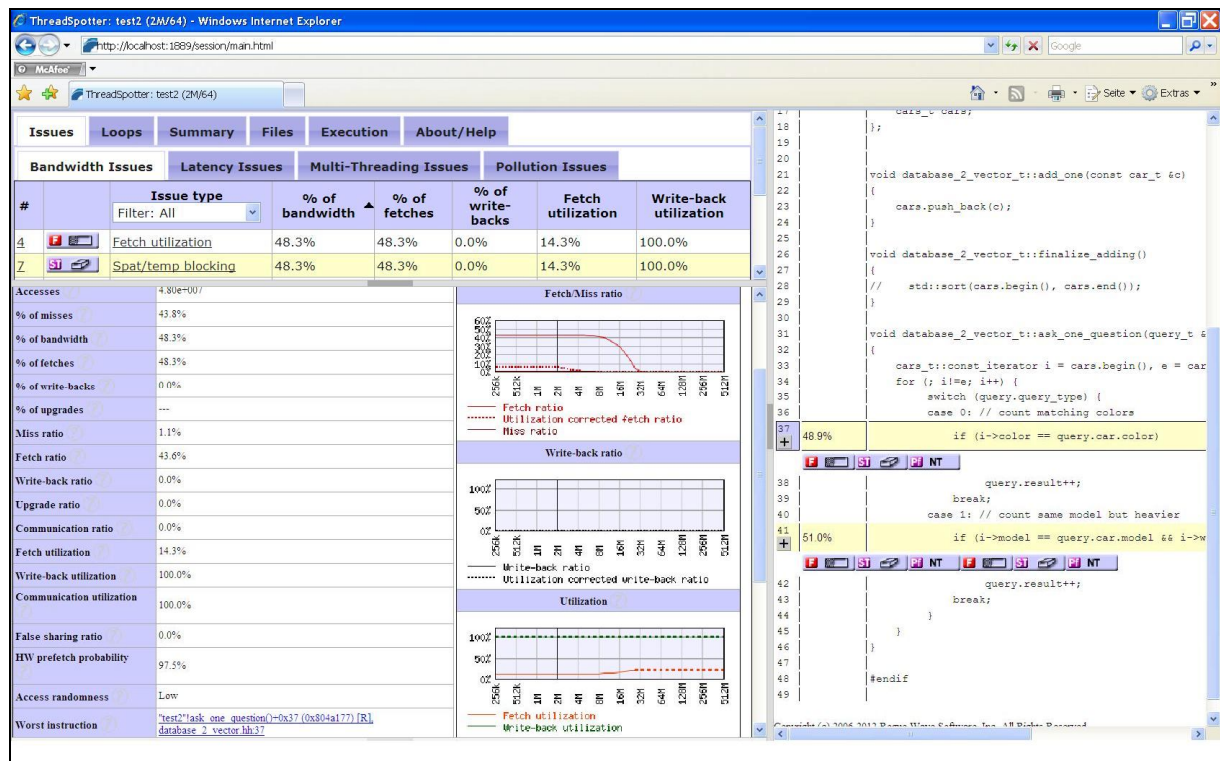
**Question 5:** What lines of the source code are responsible for the “Fetch utilization issue”?

**Question 6:** Do you have any idea how to solve this problem?



## Lab 4 – Vectors of Hot and Cold Fields

After a sampling/report generation we again look at the top level advice.



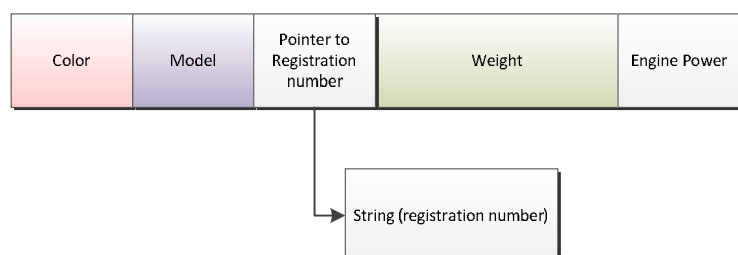
Among the top items we now find advice to address a few cases of poor spatial locality, specifically poor fetch utilization. This basically means that if the application is not using every byte in a cache line then part of the bandwidth is consumed to transfer unused data. The same unused data also occupies cache space that could otherwise be used for useful data. This makes the effective cache size smaller, and also causes less useful data to fit in each cache line.

The online help outlines the major causes of poor utilization, and offers examples of remedies.

The manual offers an enumeration of the programming patterns that cause poor fetch utilization, including code samples.

This case of poor fetch utilization complains that only a part of the `car_t` type is used in this hot loop. Several of the other fields are never used, but still use space in the cache (Reg. nr, power. String buffer is likely allocated elsewhere).

The `car_t` structure:



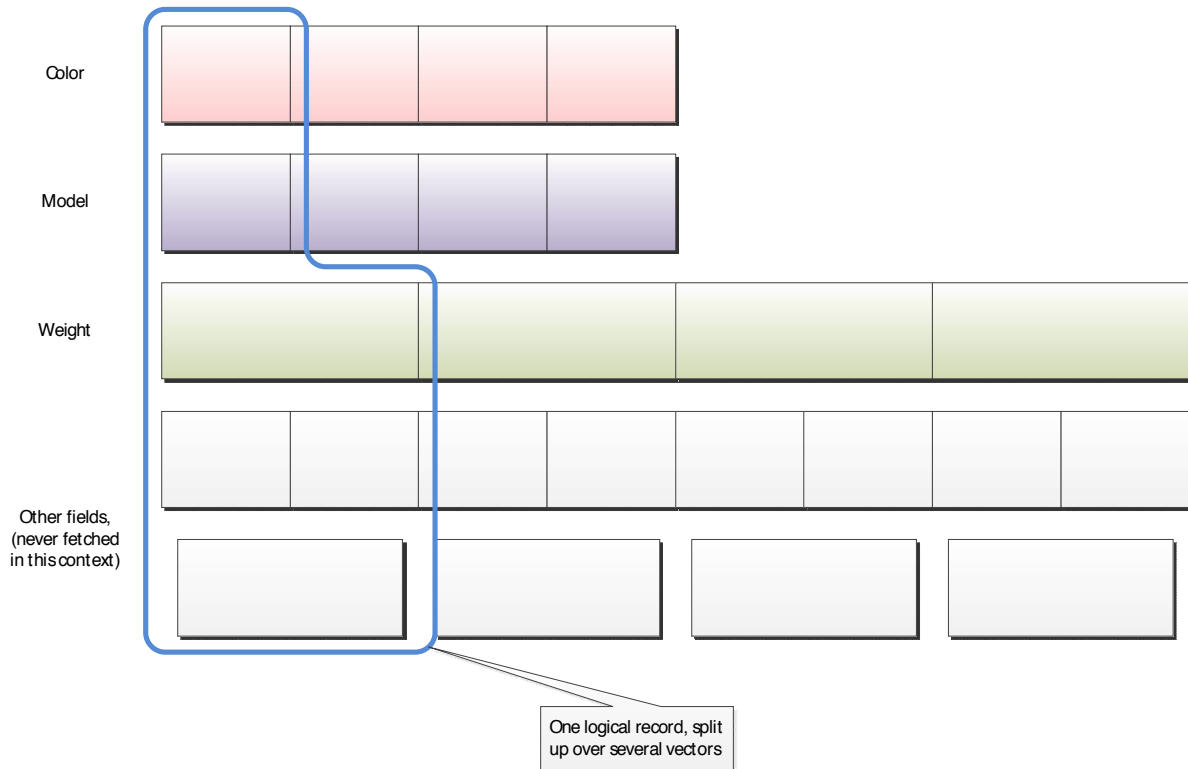
The vector:



The usual fix for this problem is to streamline the data layout. In this case, consider moving each field to its own vector. That way, only the fields being actively used will be fetched, and since the data set is traversed linearly, all adjacent entries will be used. No external alignment holes between subsequent items and no unused data.

Execution time:

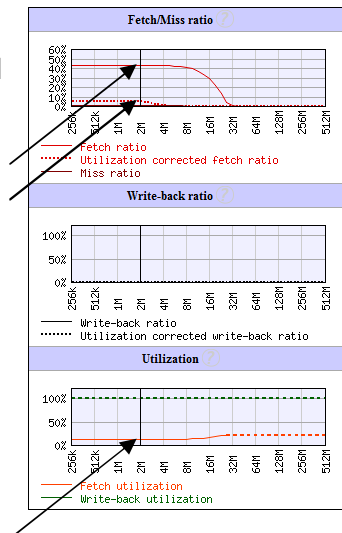
0.42 seconds.



```
class database_3_hot_cold_vector_t : public single_question_database_t {
public:
    virtual void ask_one_question(query_t &query) const;
private:
    // Primary database for the cases that infrequently fields are requested
    typedef std::vector<car_t> cars_t;
    cars_t cars;
    // cached dense copies of hot fields
    std::vector<color_t> colors;
    std::vector<model_t> models;
    std::vector<double> weights;
};

void database_3_hot_cold_vector_t::ask_one_question(query_t &query) const {
    for (int i=0; i != cars.size(); i++) {
        switch (query.query_type) {
            case 0: // count matching colors
                if (colors[i] == query.car.color)
                    query.result++;
                break;
            case 1: // count same model but heavier
                if (models[i] == query.car.model && weights[i] > query.car.weight)
                    query.result++;
                break;
        }
    }
}
```

The report does offer some additional hints of this problem. Consider the issue statistics diagram for the top “fetch utilization” issue. The red dashed line indicates how the fetch ratio would change if the fetch utilization (blue) could somehow be improved to 100%. We will revisit this graph in the next section.



Please run “test3” by typing

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ ./test3
```

**Question 1:** What is the median execution time of the program version “test3”?

Please sample “test3” and generate a report.

You will notice that the “Fetch utilization” issues have been turned into “Fetch hot-spots”.

**Question 2:** What is a “Fetch hot-spot”?

**Question 4:** What opportunities does ThreadSpotter suggest for further improvements?

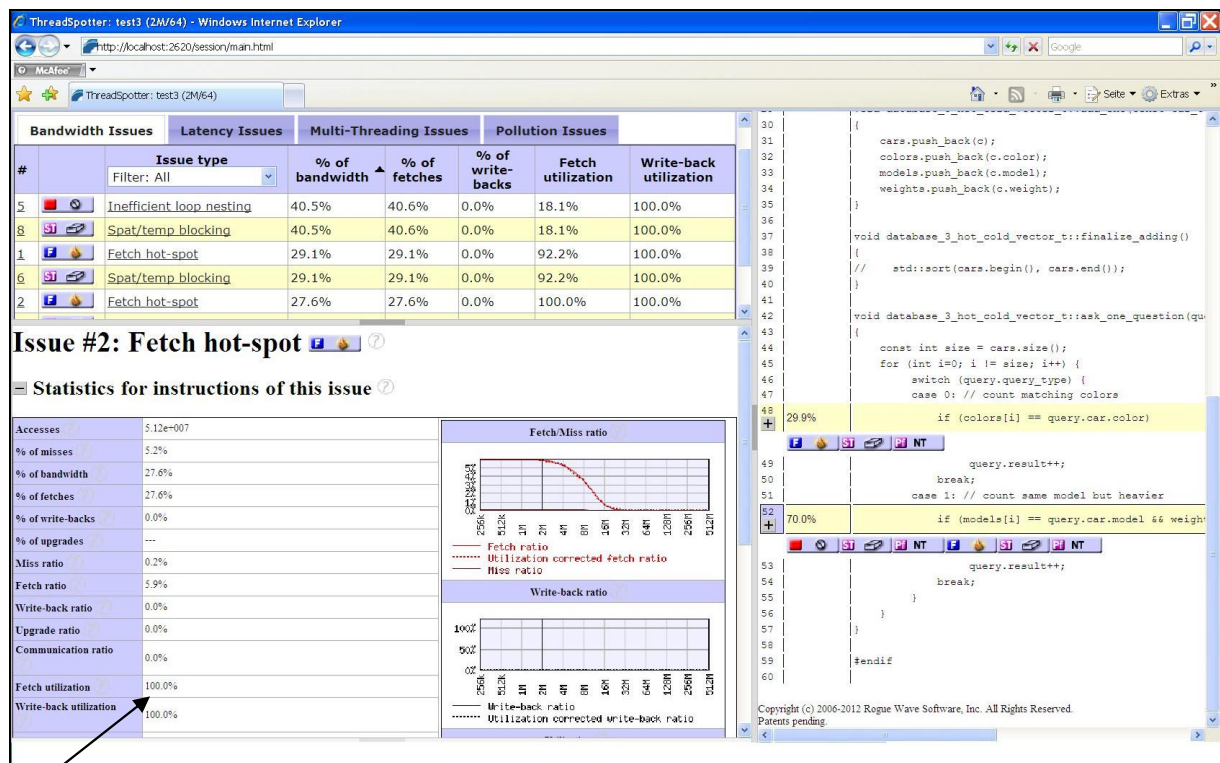
**Question 5:** What is the meaning of “spatial locality”?

**Question 6:** What is the meaning of “temporal locality”?

## Lab 5 – Blocking

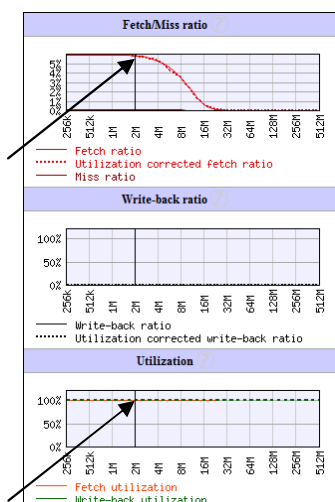
We perform another round of sampling and report generation.

Now we find that the advice to fix poor fetch utilization has been changed into hot-spots. A hot-spot is reported for instructions which use a lot of bandwidth, but have otherwise regular access patterns and most of the data is put to use at least once before the cache line is eventually evicted.



The statistics show that the fetch utilization is 100%, and comparing the fetch ratio graph we see that the fetch ratio has gone down to very close to what was hinted in the previous step. These kind of predictive capabilities are important to help the programmer judge the impact of a change before undertaking it.

Now that we have addressed all the spatial reuse opportunities, what are left in the issue list are different advice to apply blocking, both with respect to the *query* data, as well with respect to the *car* data.



ThreadSpotter: test3 (2M/64) - Windows Internet Explorer

http://localhost:2620/session/main.html

ThreadSpotter: test3 (2M/64)

Issues | Loops | Summary | Files | Execution | About/Help

Bandwidth Issues | Latency Issues | Multi-Threading Issues | Pollution Issues

#	Issue type	% of bandwidth	% of fetches	% of write-backs	Fetch utilization	Write-back utilization
5	Inefficient loop nesting	40.5%	40.6%	0.0%	18.1%	100.0%
8	Spat/temp blocking	40.5%	40.6%	0.0%	18.1%	100.0%
1	Fetch hot-spot	29.1%	29.1%	0.0%	92.2%	100.0%
6	Spat/temp blocking	29.1%	29.1%	0.0%	92.2%	100.0%
2	Fetch hot-spot	27.6%	27.6%	0.0%	100.0%	100.0%
7	Spat/temp blocking	27.6%	27.6%	0.0%	100.0%	100.0%

Copyright (c) 2006-2012 Rogue Wave Software, Inc. All Rights Reserved.  
Patents pending.

### Issue #8: Spat/temp blocking

Statistics for instructions of this issue

Instructions benefiting from blocking

Stack	Instruction	% of misses	% of fetches	Fetch ratio	Fetch utilization	W-B Utilization
+	test3!ask_one_question(-0x59 0x804a219) [R] database_3_hot_cold_vector.ll:52	78.1%	40.6%	61.7%	18.1%	100.0%

Loops and barriers related to this issue

```

29 void database_3_hot_cold_vector_t::add_one(const car_t &c)
30 {
31     cars.push_back(c);
32     colors.push_back(c.color);
33     models.push_back(c.model);
34     weights.push_back(c.weight);
35 }
36
37 void database_3_hot_cold_vector_t::finalize_adding()
38 {
39     std::sort(cars.begin(), cars.end());
40 }
41
42 void database_3_hot_cold_vector_t::ask_one_question(query_t &quer
43 {
44     const int size = cars.size();
45     for (int i=0; i != size; i++) {
46         switch (query.query_type) {
47             case 0: // count matching colors
48                 if (colors[i] == query.car.color)
49                     query.result++;
50                 break;
51             case 1: // count same model but heavier
52                 if (models[i] == query.car.model && weights[i] > que
53                     query.result++;
54                 break;
55             }
56         }
57     }
58 }
59 #endif
60

```

Blocking is a general term suggesting working on the data in smaller chunks, and using that data many times over before moving on to the next chunk. As there are many different data structures in this case, blocking can be performed in a few different ways.

The common idea is to break up data into small enough chunks that each chunk fits in the target cache. If there are multiple data sets, there are typically many different ways one or more of them can be subdivided and subject to blocking. The common idea is that the total footprint of active subsets of all data sets needs to fit in the available cache memory.

As an example of this technique, consider this nested loop structure:

```

for (int j = 0; j < size_j; j++) {
    for (int i = 0; i < size_i; i++) {                // Split this loop ...
        // Do something indexed by i, and possibly j
        sum += a[i];
    }
}

```

This will (if size\_j is large enough) repeatedly fetch the elements of the vector a. The general recipe for blocking is to split one of the inner loops to outside the outer loop:

```

for (int ii=0; ii < size_i; ii += BLOCK_FACTOR) {    // ... like this
    int limit = min(ii + BLOCK_FACTOR, size_i);

    for (int j = 0; j < size_j; j++) {
        for (int i = ii; i < limit; i++) {           // and this
            // Do something indexed by i, and possibly j
            sum += a[i];
        }
    }
}

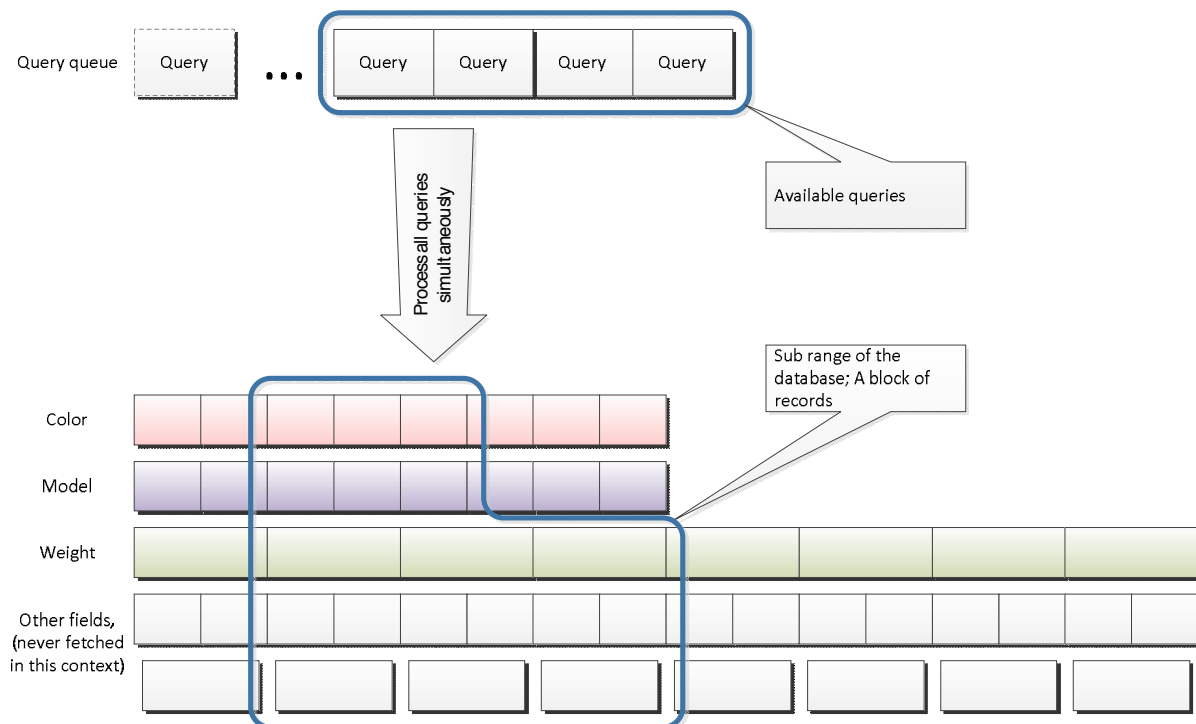
```

This will allow subsections of vector a to remain in the cache for repeated executions of the j loop.

In our example, we have a list of queries against the database. The queries may arrive on a queue from network connections or other parts of the application. Rather than processing each query by itself, we may take a block of queries and process each of them in parallel against the database. That way the total number of memory fetches will be reduced.

Execution time:  
0.31 seconds.

This idea can be implemented in different ways. Either we can make one traversal through the database, and for each record process all of our queries, or we can break up the database traversal into sub ranges, and work on one such sub range at a time, traversing it once for each query. The latter is more efficient in this case, since traversal of sequentially stored vectors is very efficient and we want to have as long stretches as possible. After all queries have had partial results recorded from the current sub range, then we advance to the next block of database records and resume processing our queries against that block.



In this case we opted to break up the database in chunks of BLOCK\_FACTOR elements, but also to group similar queries together into categories, and work on each database sub range, one category at a time. This may further have helped reduce cache pressure thanks to not involving too many record fields at the same time.

```

#define BLOCK_FACTOR 1000

void database_4_blocking2_t::ask_questions(queries_t &queries) const {
    std::vector<query_t*> query_0, query_1;
    for (int j=0; j < queries.size(); j++) {
        query_t &query = queries[j];
        switch (query.query_type) {
            case 0:
                query_0.push_back(&query);
                break;
            case 1:
                query_1.push_back(&query);
                break;
        }
    }

    for (int ii=0; ii < cars.size(); ii += BLOCK_FACTOR) {
        int limit = min(ii + BLOCK_FACTOR, cars.size());

        // query type 0
        for (int j = 0; j < query_0.size(); j++) {
            query_t &query = *query_0[j];
            for (int i = ii; i < limit; i++) {
                if (colors[i] == query.car.color)
                    query.result++;
            }
        }

        // query type 1
        for (int j = 0; j < query_1.size(); j++) {
            query_t &query = *query_1[j];
            for (int i = ii; i < limit; i++) {
                if (models[i] == query.car.model &&
                    weights[i] > query.car.weight)
                    query.result++;
            }
        }
    }
}

```

If the number of queries to be processed as a batch becomes too large, they too could outgrow the available cache space. Again, this could be dealt with in the same way, by selecting a smaller number of queries to work through the database fragments.

Please execute “test4” first. This blocking approach processes chunks of database records.

Please run “test4” by typing

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ ./test4
```

**Question 1:** What is the median execution time of the program version “test4”?

The program version “test4b” groups the database records as well as the query types.

Please run “test4b” by typing

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ ./test4b
```

**Question 2:** What is the median execution time of the improved program version “test4b”?



Please sample “test4b” and generate a report. Already the first page will indicate that there are no significant bandwidth and locality issues left. Memory latency has been improved significantly.

Opening the report you won’t find any “Slowspot Issues” any more except “Fetch hot-spot”.

There are only a few minor “Opportunity Issues” left.

**Question 3:** What does “blocking” mean?

**Question 4:** How do you determine a reasonable blocking factor?

**Question 5:** Is blocking always possible?



## Scalability

We have already measured the single instance performance improvements for each improvement step. Now, being lean has another benefit: Better scalability. The reasons are simple: the more parallel activity in a system, the easier it is to saturate the available memory bus bandwidth. If memory bandwidth is not preserved, then total throughput will suffer.

To gauge this effect, we measure the execution time when deploying more instances of the program at the same time. The table shows the wall-clock time for all parallel runs to finish, in seconds (all measurements are made on a Dual Intel Xeon E5345 Quad Core):

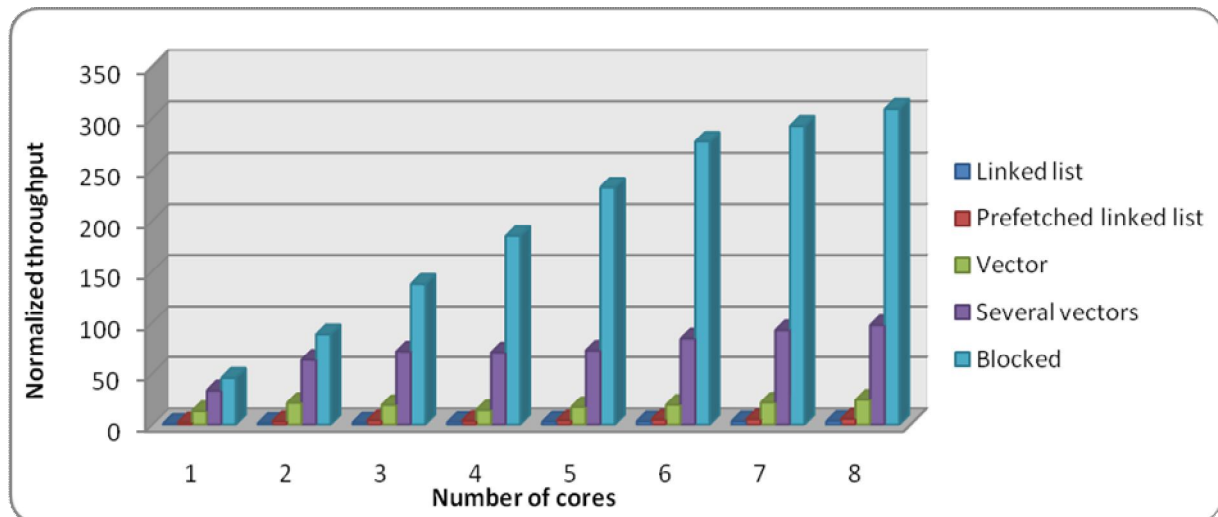
# cores:	1	2	3	4	5	6	7	8
1 – Linked list	14	16	18	20	23	24	28	30
2 – Prefetched linked list	6.5	8.0	9.7	15	16	17	18	19
3 – Vector	1.0	1.3	2.2	3.9	4.0	4.3	4.5	4.6
4 – Several vectors	0.42	0.44	0.59	0.80	0.97	1.0	1.1	1.2
5 – Blocked	0.31	0.32	0.31	0.31	0.30	0.31	0.34	0.37

And the same information expressed as normalized throughput:

# cores:	1	2	3	4	5	6	7	8
1 – Linked list	≡ 1	1.8	2.4	2.8	3.1	3.5	3.5	3.7
2 – Prefetched linked list	2.2	3.5	4.4	3.8	4.5	5.1	5.4	5.9
3 – Vector	14	22	19	15	18	20	22	25
4 – Several vectors	34	63	72	71	73	85	93	98
5 – Blocked	46	89	138	185	232	277	293	309

Notice how the first four program versions level out after just a few instances, and offer very little extra performance as more cores are used. The first two versions are mostly limited by the memory latency. The next two versions do better but are still bandwidth limited.

Only the last version scales almost linearly with number of cores. This is due to its preservation of bandwidth through reuse of data.



If your laptop has been equipped with either a multicore cpu or multiple cpus on board you may test the scalability of the different example program versions by using the shell script “run-many.sh”. It will print out the median execution times for 1 to n started instances of a program (n=number of cores/ single core cpus).

Please type

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ run-many.sh / test1
```

and compare the output with

```
demouser@ubuntu:/media/xxxx-xxxx/tutorial$ run-many.sh / test4b
```

**Question 1:** What is the reason for the improved scalability?

## Summary

Based on advice from Rogue Wave ThreadSpotter, we have implemented a series of changes to a simple table lookup mechanism. We have seen how different data representations and different data access patterns may have a large impact on application performance and scalability. And we have achieved magnitudes better performance with relatively moderate changes.

The golden rule is to ensure that all closely placed data is used and reused as much as, and as soon as possible. If there is regularity in the application’s access patterns, exploit it. Otherwise seek to change the program into one with such properties.

The studied application is memory bound and therefore responds very well to this treatment. This is also the case of many numerical applications with large datasets that are repeatedly traversed.

## Appendix I – Controlling the Point for Attach and Detach

As mentioned, for the purpose of this test we want to control exactly when the sampler is active.

We add a dummy function for each point in the code where we want to be able to attach or detach. To prevent the function from being inlined or optimized away, we use two compiler specific constructs:

```
extern "C" __attribute__((noinline))          // gcc syntax
void start_sampling()
{
    asm volatile("");                          // gcc syntax
}

extern "C" __attribute__((noinline))          // gcc syntax
void stop_sampling()
{
    asm volatile("");                          // gcc syntax
}

int main() {
    // set up data
    // ...

    start_sampling();
    // code to be sampled
    // ...
    stop_sampling();

    // clean up
    // ...
    return 0;
}
```

Now, the sampling can be started like this:

```
$ sample --start-at-function start_sampling \
        --stop-at-function stop_sampling -r ./test-binary
```

By using this technique we create an environment in which we can disregard initializations and clean-up code, to enable both completely repeatable and comparable runs to be measured.

## Appendix II – Answers and Explanations

### Lab 1

**Question 1:** What is the median execution time of the program version “test1”?

Answer: Depending on your hardware you will most likely get a medium execution time in the range of 11.5 seconds to 12.5 seconds.

**Question 2:** How many samples are necessary to get a reliable report?

Answer: 10000 samples.

**Question 3:** What is the reason for starting/ stopping sampling at functions `start_sampling/ stop_sampling`?

Answer: In order to guarantee comparable sample conditions for all our test programs we need to ensure not to sample for instance the initial parts of the programs which could differ regarding their execution times. Appendix I describes how to use dummy functions for controlling the sampler’s attach- and detach-points.

**Question 4:** Why should you always start optimizing your application related to the highest level cache?

Answer: The highest level cache is the one of most capacity. Usually the memory footprint of an application is not small enough in order to fit in the first level cache. In contrast the highest level cache will be able to store a huge amount of the application’s data. That means that optimizing regarding the highest level cache will show most significant improvements.

**Question 5:** What is the dominant issue listed under the “Latency Issues” tab?

Answer: It’s a “Random access” issue. All issues are listed ordered by severity. The “Random access” issue can be found on top of the list.

**Question 6:** What is the meaning of “Access randomness” which is very high in this case?

Answer: The access to data is very irregular. The prefetcher is not able to detect patterns in order to determine the data that will be used next. Therefore it is working inefficiently.

**Question 7:** What fundamental data structure is the cause of this problem?

Answer: In this case the linked list is responsible for the problem. In general pointer chasing, dynamically allocated chained structures like trees, graphs and lists tend to distribute data randomly in memory.

**Question 8:** What is the reason for the huge amount of cache misses?

Answer: If the prefetcher is not working efficiently the likelihood that requested data has not been cached increases. A fetch to the cache will more often end up in a miss.

**Question 9:** What is so bad about cache misses?

Answer: The cpu stalls for a long time while waiting for data to be transferred from main memory to the cache.

**Question 10:** What are the options to avoid the prefetching problem in this case?

Answer: In order to avoid the prefetching problem it would be possible either to add prefetch instructions to the code or storing the data in a way that traversals will be easier to prefetch.

## Lab 2

**Question 1:** What is the median execution time of the program version “test1c”?

Answer: Depending on your hardware you will most likely get a medium execution time in the range of 11.0 seconds to 12.0 seconds. It will probably be slightly faster than “test1”

**Question 2:** Is the prefetcher working efficiently now?

Answer: The prefetcher is definitely working slightly better because of the implemented prefetch statements but is still far from working efficiently. You will see a positive effect of a slightly lower miss-fetch ratio because of a little better prefetch probability.

**Question 3:** What can be done in order to avoid the “Random access” issue?

Answer: As we have already tried to solve the problem by including software prefetch statements the only option left is to substitute the linked list by another more prefetcher friendly data structure like a simple array or a `std::vector`.

## Lab 3

**Question 1:** What is the median execution time of the program version “test2”?

Answer: The execution time went down dramatically. Your tests will probably show results around 0.5 seconds.

**Question 2:** Has the “Random access” issue been solved by substituting the linked list with a `std::vector`?

Answer: Yes, the “Random access” issue has been solved. You will find no issue listed under the “Latency issues” tab any more.

**Question 3:** What dominant issue can be found on top of the issues list?

Answer: The most important issue to solve is a “Fetch utilization” problem now.

**Question 4:** Why does low fetch utilization slow down the program?

Answer: If the cache lines only partially consist of data used by the program a lot of valuable cache space is filled up with useless data. In addition the useless data also has to be transferred from main memory to the cache and will require bandwidth unnecessarily.

**Question 5:** What lines of the source code are responsible for the “Fetch utilization issue”?

Answer: `database_2_vector.hh`, line 37 and line 41 `if (i->color == query.car.color)` and `if (i->model == query.car.model && i->weight > query.car.weight)`

**Question 6:** Do you have any idea how to solve this problem?

Answer: In general splitting complex structure into sub-structures could solve issues like this. In this case splitting the “car\_t” structure into multiple vectors would improve the cache line utilization. Only vectors containing the “usable” data would be copied to the cache.

## Lab 4

**Question 1:** What is the median execution time of the program version “test3”?

Answer: The median execution time will be probably about 0.4 seconds.

**Question 2:** What is a “Fetch hot-spot”?

Answer: A “Fetch hot-spot” issue will be reported when ThreadSpotter has encountered a location which is responsible for an exceptionally large number of cache line fetches.

**Question 4:** What opportunities does ThreadSpotter suggest for further improvements?

Answer: Spatial/ temporal blocking.

**Question 5:** What is the meaning of “spatial locality”?

Answer: Spatial locality means that a program is using data which is located in memory close to the already used data. Because chunks of data are loaded into the cache it is most likely that data located close to recently requested data will also reside in the cache. *Spatial blocking* can help to improve spatial locality.

**Question 6:** What is the meaning of “temporal locality”?

Answer: Temporal locality means that a program is reusing recently used data again. The likelihood that the data is still in the cache is dependent on the time between the two accesses. *Temporal blocking* may improve temporal locality.

## Lab 5

**Question 1:** What is the median execution time of the program version “test4”?

Answer: The median execution time will probably about 0.5 seconds.

**Question 2:** What is the median execution time of the improved program version “test4b”?

Answer: You will probably see an improved median execution time of about 0.3 seconds.

**Question 3:** What does “blocking” mean?

Answer: Rearranging algorithms, specifically the order or nesting of loops, to focus on working on smaller subsets of the data. The idea is to partition the data set into small enough fractions that will fit in the target cache. Then change the algorithms to read and update that data a number of times before letting them be evicted from the cache.

**Question 4:** How do you determine a reasonable blocking factor?

Consider the available cache space, and look at the fetch rate curve. The place where the fetch rate curve gets close to 0 is the active footprint of the algorithm. The relationship of this point

compared to the cache size gives you a factor by which you need to reduce your current footprint.

Note that this factor applies to all data sets that are being touched. Guided by this factor and other knowledge of your program, you need to figure out how much to reduce and block each data set.

**Question 5:** Is blocking always possible?

Applying blocking invariably means that you will be altering the order of traversal of elements in your data sets. This may not always be possible. As a basic observation, if the changed code mandates a different order of say a read and a write operation to the same data, the new program will have a different meaning than the old one. So-called loop carried dependencies (or just data dependencies for short) may therefore prevent you from using blocking techniques.



















Sometimes it is possible to find non-regular or oblique spatial decompositions that allow you to succeed in finding blocking. Sometimes it is not possible, and in that case you would have to revisit your algorithm seeking alternative calculation schemes, possibly with different numerical properties.

## Scalability

**Question 1:** What is the reason for the improved scalability?

Answer: Temporal blocking means significantly reduced need to re-fetch data. Due to minimizing the bandwidth requirement of the application the cpus/ cores do not stall even if many instances share the bus system.

## Appendix III – Issues Discovered by ThreadSpotter

SlowSpot Issues	Opportunity Issues
 Fetch utilization	 Spatial blocking
 Write back utilization	 Temporal blocking
 Communication utilization	 Spat/temp blocking
 Inefficient loop nesting	 Loop fusion
 Random access	 Non-temporal data
 Prefetch: too close	 Non-temporal store possible
 Prefetch: too distant	 Fetch hot-spot
 Prefetch: unnecessary	 Write-back hot-spot
 False sharing	 Communication hot-spot





## Contact information

General Inquiries: [sales@roguewave.com](mailto:sales@roguewave.com)

German Office: +49 6103 5934 0

[sales@roguewave.de](mailto:sales@roguewave.de)

<http://www.roguewave.com>