ThreadSpotter™

Manual

Version 2012.1.1
2016-10-04

ThreadSpotter™: Manual

ParaTools, Inc.
2836 Kincaid St.
97405 Eugene
OR

info@paratools.com

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

ThreadSpotter™ is a powerful tool that gives developers unprecedented insight into memory related performance problems in an application.

ThreadSpotter™ analyzes an application and provides advice to the programmer on how to go about correcting memory performance problems, and offers the programmer insight into where and why performance problems occur. The programmer, using ThreadSpotter™'s advice and feedback, can modify the source code to optimize the data layout and code efficiency, and obtain performance improvements.

ThreadSpotter™ will teach a novice programmer about good design choices and help him understand what constructs are detrimental to performance by pinpointing problem areas in his program. ThreadSpotter™ will explain the performance problem, point to the source code and offer guidance on how to improve the code.

For an optimization expert ThreadSpotter™ will increase productivity, for instance, by quickly identifying the problem areas in the application and by allowing him to analyze the cache behavior of the application on multiple architectures based on a single sampling.

1.1. Overview

The process of optimizing an application with ThreadSpotter™ starts with sampling the application. The application is sampled to capture a fingerprint of its memory access behaviour and collect information about its structure. The user can sample an application from start to end, or attach to the application while it is running, sample it for a while and detach.

Once the sampling is done the captured data is analyzed to discover performance problems. A report is created, listing the performance problems found in the code and mapping them back to the source code. Each performance problem is classified and the programmer can find directions on how to fix each class of problems in the online documentation. The performance problems include issues such as:

- Inefficient data layout
- Inefficient data access patterns
- Unexploited data reuse opportunities
- Prefetching problems
- Thread interaction problems

ThreadSpotter™ also calculates cache metrics, such as cache miss ratios, cache fetch ratios, cache line utilizations and hardware prefetch probabilities. These metrics further guide the developer in identifying and understanding the causes of memory related performance problems.

Performance issues are automatically ranked in order of severity in the report. Once the programmer has fixed the most severe problems, he can sample and analyze the improved application to verify the improvements and find other performance bottlenecks.

1.2. Technology

ThreadSpotter™'s analysis focuses on the memory access behaviour of the application, and specifically on how the application's memory access patterns interact with the processor caches. Improving the application's interaction with the processor caches has two effects:

- It decreases the cache miss ratio of the application. Cache misses introduce memory access latencies that cause stalls in the execution, and decreasing the cache miss ratio directly reduces the execution time of the application.
- It decreases the memory bandwidth requirement of the application. When an application becomes memory bandwidth limited it is no longer the memory access latencies that limit the execution speed, but the bandwidth at which data can be transferred to and from memory. In a memory bandwidth limited application many performance optimizations aimed at reducing the impact of memory access latencies, for example, prefetching, become useless or even detrimental.

  Decreasing the memory bandwidth requirement of applications becomes even more important with the introduction of multicore processors. More cores sharing the same memory bandwidth means the memory bandwidth is quickly exhausted. Memory bandwidth is the most important bottleneck for scaling application performance on multicore processors.

ThreadSpotter™ uses a proprietary light-weight sampling technology when sampling the application, not hardware performance counters. The collected fingerprint is richer in information than what can be obtained from the hardware performance counters and allows metrics such as utilization of fetched cache lines to be calculated.

Since it is the behavior of the application and not the hardware that is sampled, the gathered data is independent of the hardware the sampling was done on. This allows the user to analyze how the application would behave on a processor with a different cache configuration, for example, to predict the performance on different processor models.

ThreadSpotter™ is programming language independent. It only looks at the binary code of the application. It can generate a report without any source code, but to generate source code references it needs access to the source code and the application must be compiled with debug information in a standard format.

1.3. Limitations

ThreadSpotter™ produces cache usage metrics and hints for optimizing the application. The responsibility for interpreting this information and using it to optimize the application lies with the programmer.

The sampling does in general not affect the sampled application in any other way than slowing it down, but the behavior of some applications, that rely on specific timing, may in some cases be affected by the sampling.

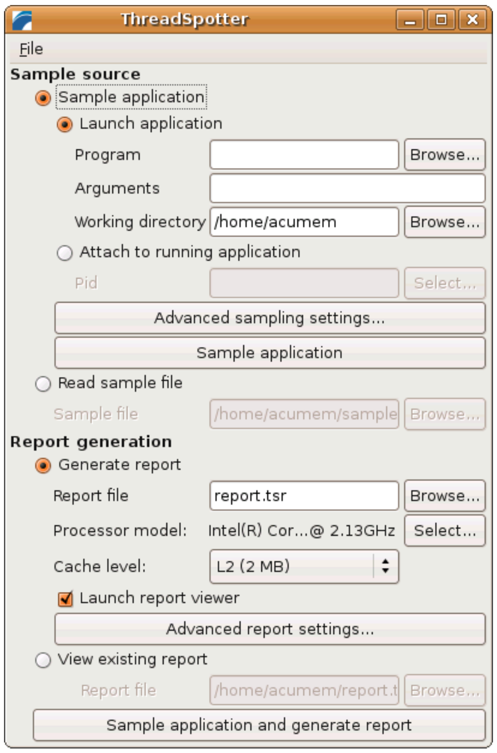
While any ELF binary can be sampled, mapping the analysis results back to the source code requires the availability of DWARF or STABS debug information.

Chapter 2. Running ThreadSpotter™

2.1. Using the Graphical User Interface

The graphical user interface provides an easy way to sample applications and generate reports. It is started from the menu of your window manager, or using the **threadspotter** command from the ThreadSpotter™ package, usually installed as **/opt/threadspotter/bin/threadspotter**:

```
$ threadspotter
```



Sample source

[✓ Sample application]

[✓ Launch application]

Program

Arguments

Working directory /home/acumem

Attach to running application

Pid

Read sample file

Sample file /home/acumem/sample

Report generation

[✓ Generate report]

Report file report.tsr

Processor model: Intel(R) Cor...@ 2.13GHz

Cache level: L2 (2 MB)

[✓ Launch report viewer]

View existing report

Report file /home/acumem/report.t

Figure 2.1. Overview of the GUI

There are three basic tasks you can perform from the graphical user interface:

- Sample an application and save the collected information in a sample file.
- Generate a report file from a sample file.
- View a report file.

You can also perform any combination of these the tasks at once, for example, launch and sample an application, generate a report and view the generated report in one go.

2.1.1. Sampling an Application

Sampling an application and saving the collected data to a sample file is useful if you want to generate multiple reports based on the same application run, for example, if you want generate reports for several different cache configurations for the same application.

First you need to select a name for the sample file. Click the *Advanced sampling settings...* button. In the dialog window that opens, check the *Save sample file as* box and enter a file name.

In this window you can also make some other choices about how to sample the application, see Section 2.1.5, “Advanced Sampling Settings” for more information.

There are two ways to sample an application:

- Launching and sampling an application.

  The application to be sampled is launched by ThreadSpotter™ and sampled. By default the application is sampled until it terminates. If you choose to stop the sampling before the application is finished the application is automatically terminated.

- Sampling a running application.

  ThreadSpotter™ attaches to a running application and samples it. When the sampling is finished ThreadSpotter™ detaches and the application continues running normally.

Note that when launching an application the sampler does not sample other processes started by the specified application. This means that if you want to sample an application that is usually started through a script or by a different application, you have to check what command the script or application uses to start the application and manually specify that command. If you specify the script, only the script itself will be sampled, not the intended application.

This also applies when you attach to a running application that in turn starts other processes.

2.1.1.1. Launching and Sampling an Application

Select *Launch application* as sample source. Fill in the application to be sampled in the *Program* field and the arguments to the application in the *Arguments* field. For example, if you want to sample the command **ls -l -a**, fill in **ls** in the *Program* field and **-l -a** in the *Arguments* field.

When you are done click the *Sample application* button. When the sampling starts an **xterm** containing the sampled application will pop up.

The application will be sampled until the it terminates, or until the sampling stop condition is met if you have specified one, see Section 2.1.5, “Advanced Sampling Settings”. When the sampling has finished the sample file will be post-processed for a short while.

The terminal has to be closed manually when the process has terminated and the post-processing of the sample file has finished. This allows you to review any output from the sampled application and the sampling process.

2.1.1.2. Sampling a Running Application

ThreadSpotter™ is also capable of sampling a running application. It will attach to it without the need to restart the application. When the sampling is done, it can detach from the application, leaving it running as before.

To sample a running application, select *Attach to running application* as sample source. Fill in the *PID* field with the process id of the process to be sampled if you know it, or click the *Select...* button to select the process from a list.

When the sampling starts an **xterm** will pop up showing the sampler prompt. Enter the **q** command to stop sampling. When the sampling has finished the sample file will be post-processed for a short while.

The terminal has to be closed manually when the process has terminated and the post-processing of the sample file has finished. This allows you to review any output from the sampling process.

Sometimes the sampler will not be able to detach from the process immediately when requested to stop sampling. This happens if a sampled thread is waiting in a system call. If this happens you can try to "activate" the application to get the thread to finish the system call, for example, if the application has a command line interface and is waiting for user input, entering a command may help.

If that does not help, you can force a detach by giving the sampler the **q** command one more time. Doing this will cause a small memory leak in the sampled process, but that will usually not cause any problems.

In very rare cases it may not be possibly to stop the sampling without corrupting the sampled application. You will get a warning if that is the case.

2.1.2. Generating a Report from a Sample File

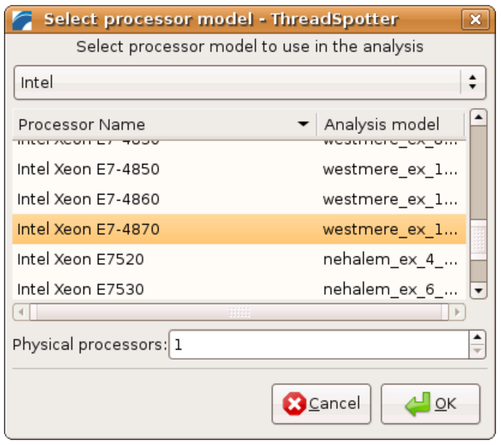
To generate a report based on an existing sample file, change the sample source setting to *Read sample file*.

You can then select the report file name and a processor model and cache level to do the analysis for in the *Report generation* part of the window.

By default the report is created in the current working directory and named `report.tsr`.

The advice in the report depends on the selected processor model and cache level. Some data structures or code sections may, for example, work well with a large cache but have problems with another processor model with a smaller cache, or they may work well in the L2 cache but have problems in the smaller L1 cache. To get relevant advice for your application and hardware, you have to specify a processor model that matches the processor you want to optimize for.

By default the processor model of the computer you are running on is selected. To change the processor model click the *Select...* button to the right of the model name.



Select processor model to use in the analysis

Intel

Processor Name	Analysis model
Intel Xeon E7-4850	westmere_ex_1...
Intel Xeon E7-4860	westmere_ex_1...
Intel Xeon E7-4870	westmere_ex_1...
Intel Xeon E7520	nehalem_ex_4_...
Intel Xeon E7530	nehalem_ex_6_...

Physical processors: 1

Figure 2.2. Processor Model Selector

The processor model contains information about several aspects of the cache hierarchy, for example cache sharing and active prefetch instructions. If the intended model is not available, you may use a similar processor from the same product family. You may need to override parts of the processor model to get something that suits the target processor.

Model parameters and additional analysis parameters may be changed if you click the *Advanced report settings...* button. See Section 2.1.6, “Advanced Report Settings”.

To start the report generation click the *Generate report from sample file* button. When the report has been generated it is automatically opened in your web browser.

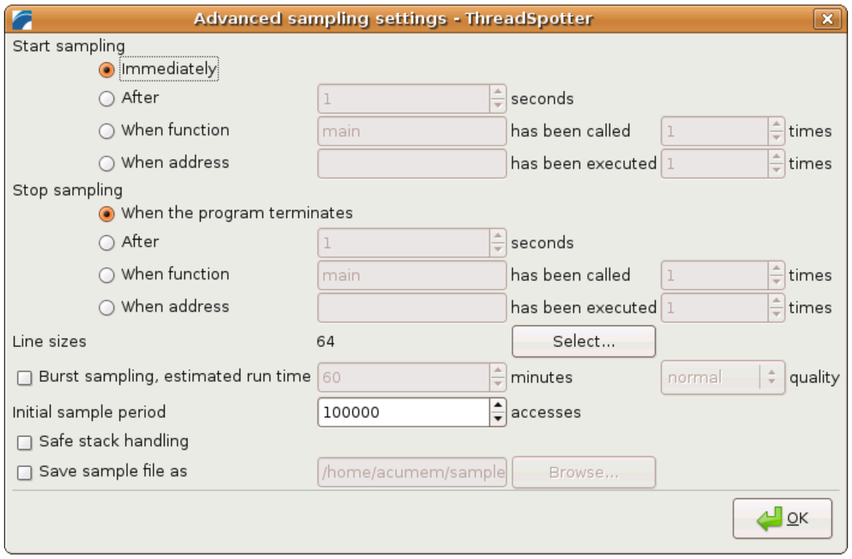
2.1.3. Sampling and Generating a Report

To sample an application and generate a report in one go, first configure the sampling as described in Section 2.1.1, "Sampling an Application". You do not need to specify a sample file name unless you want to keep the sample file. Then configure the report generation as described in Section 2.1.2, "Generating a Report from a Sample File", but without changing the sample source to *Read sample file*. Finally, click the *Sample application and generate report* button.

2.1.4. Viewing an Existing Report

To view an existing report, select the *View existing report* option in the *Report generation* section, select the report file, and then click the *View report* button.

2.1.5. Advanced Sampling Settings



Advanced sampling settings - ThreadSpotter

Start sampling
[✓ Immediately]
After 1 seconds
When function main has been called 1 times
When address has been executed 1 times

Stop sampling
[✓ When the program terminates]
After 1 seconds
When function main has been called 1 times
When address has been executed 1 times

Line sizes 64 Select...
☐ Burst sampling, estimated run time 60 minutes normal quality
Initial sample period 100000 accesses
☐ Safe stack handling
☐ Save sample file as /home/acumem/sample Browse...

OK

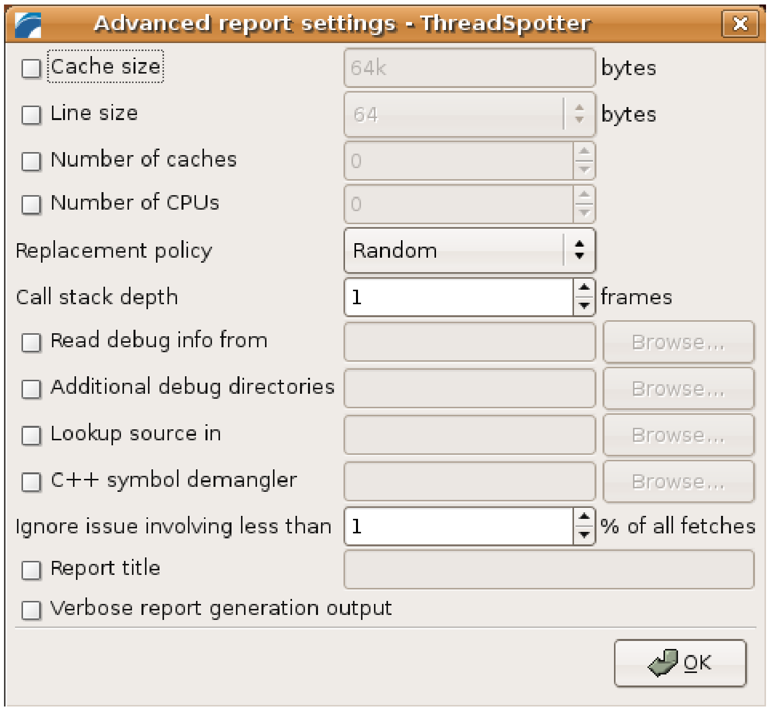
Figure 2.3. Advanced Sampling Settings

Clicking the *Advanced sampling settings...* button in the main window opens a window with some additional settings for sampling the application.

Start sampling	By default the sampler starts sampling the application immediately after it has launched or attached to it. Here you can choose to instead start sampling after a fixed delay or when a function or address has been executed a number of times. See Section 2.4.2, "Sampling Start Conditions" for more information.
Stop sampling	By default the sampler will sample the application until it terminates or the user manually stops the sampling. Here you can choose to instead stop sampling after a fixed delay or when a function or address has been executed a number of times. See Section 2.4.3, "Sampling Stop Conditions" for more information.

Line sizes	Use this option if you want to generate a report for a cache line size other than the default. The line size selected for analysis is automatically selected and can not be unselected.
Burst sampling	Use this option to enable burst sampling in order to reduce the sampling overhead for sampling runs that take at least 5 minutes. When enabled, you also need to enter an estimated execution time without sampling for the part of the application that you intend to sample. See Section 2.4.1, “Burst Sampling” for more information about burst sampling.
Initial sample period	Use this setting if you get a message that you need to decrease the sample period during post-processing of the sample file. See Section 4.13, “Sample Period” for more information.
Safe stack handling	If you experience incorrect execution or crashes in the sampled program try enabling this option, see Section 2.4.4.2, “Disabling Application Stack Use by the Sampler”.
Save sample file as	Sample files are normally not saved when sampling and generating a report in one go. Use this option if you want to save the sample file so that you can generate more reports from it later, or if you are sampling an application without generating a report.

2.1.6. Advanced Report Settings


Advanced report settings - ThreadSpotter

☐ Cache size	64k	bytes
☐ Line size	64	bytes
☐ Number of caches	0	
☐ Number of CPUs	0	
Replacement policy	Random	
Call stack depth	1	frames
☐ Read debug info from		Browse...
☐ Additional debug directories		Browse...
☐ Lookup source in		Browse...
☐ C++ symbol demangler		Browse...
Ignore issue involving less than	1	% of all fetches
☐ Report title		
☐ Verbose report generation output		

Figure 2.4. Advanced Report Settings

Clicking the *Advanced report settings...* button in the main window opens a window with some additional settings for the report generation.

Cache size	Overrides the cache size selected by the processor model. You can use the suffixes **k**, **m** and **g** for kilobytes, megabytes and gigabytes, respectively, when specifying the cache size. For example, **64k** for a 64 kilobyte cache.
Line size	Overrides the cache line size selected by the processor model. The selected line size will automatically be added to the list of line sizes to sample.
Number of caches	Overrides the number of caches on the selected cache level. Observed threads will be assumed to be mapped onto this number of caches.
Number of CPUs	Overrides the number of CPU sockets. This sets the number of assumed caches on the selected cache level in accordance with the current CPU model. Observed threads will be assumed to be mapped onto this number of caches. This is an alternative to the option of specifying the number of caches.
Replacement policy	Selects random replacement or LRU replacement as cache replacement policy for the analysis. Random replacement is the default.
Call stack depth	This tells the analysis how many call stack levels to consider when separating issues in functions called from multiple places based on where the function was called from. See Section 4.12, “Call Stack”.
Read debug information from	Use this option to specify the location of the sampled binary if it has been moved from the location it resided in during the sampling.
Look up source in	Look up source code in a user defined directory. Specify the new location of the source code if it has been moved since the application was compiled.
C++ symbol demangler	C++ symbol names normally appear mangled, particularly when using the stabs debug format. The compiler is usually accompanied with a program to render symbols in a human readable format. The default name for this program is **c++filt**. Specify the name of the program to use as a filter, or check the box and leave empty for the default name.
Ignore issues	Do not report issues involving less than this percentage of the total fetches, upgrades or write-backs.
Report title	Set a custom report title.
Verbose report generation output	If selected, a window showing the output from the report generator will be opened. Useful when looking for problems with report generation, for example, if the report generator fails to find source code.

2.1.7. Using a Different Browser

The **threadspotter** tool will try to start the default browser on the system. If you wish to view the report in another browser, or if the tool fails to find a browser, you can set the `BROWSER` environment variable to the browser that should be used. For example, if you are using the **bash** shell and want to use the **google-chrome** browser, you can use this command before starting the tool:

$ export BROWSER=google-chrome

ThreadSpotter™ has been verified to work with Internet Explorer, Firefox and Google Chrome. If you are using a different browser and experience problems viewing the report, try using one of the supported browsers instead.

2.1.8. Using Firefox on Multiple Computers

The Firefox browser, and possibly other Gecko-based browsers, have a default behaviour when used on multiple computers in an X environment that is confusing, and that does not work well with the **threadspotter** tool.

When you start a Firefox process and specify a URL to open, the process will by default look for other Firefox processes running in the same X server, and if one is found open the URL in that process's window even if that process is running on a different computer.

This does not work with the **threadspotter** tool for a couple of reasons. The URL used to connect to the report web server (see Section 2.2.3, “Viewing a Report” for more information about the web server) uses the address *localhost* to refer to the web server, so if the report is opened on a another computer the URL will refer to the wrong computer. Also, since the report web server only accepts connections from the computer it is running on, Firefox would not be able to open the report from another computer even if it managed to resolve the URL to the correct computer.

The easiest way to avoid this problem is to set the `MOZ_NO_REMOTE` environment variable to 1 before you run the **threadspotter** tool. For example, if you are using the **bash** shell, you can use this command:

$ export MOZ_NO_REMOTE=1

This forces Firefox to start a new process on the computer where the command is run, even if there are already Firefox processes running on other computers. However, it will also prevent you from launching multiple browsers viewing the same report on the same computer, so you will have to close the report you are viewing before you can open another report on the same computer.

2.2. Using the Command Line Tools

The command line tools provide a way to sample applications and generate reports when the graphical user interface can not be used, for example, when you are not working in a graphical environment, or in a scripted environment. Or if you simply prefer to work from the command line.

Note

The ThreadSpotter™ command line tools use the short and generic names **sample**, **report** and **view**. These names are known to also be used by operating system tools that are in the default `PATH` on some Linux distributions. We therefore recommend that you add the directory containing the ThreadSpotter™ tools, `/opt/threadspotter/bin`, first in your `PATH`, or refer to the tools using absolute paths.

For example, if you are using the **bash** shell and want to add the ThreadSpotter™ tools first in your `PATH`, you can use this command:

$ export PATH="/opt/threadspotter/bin:${PATH}"

To automatically do this every time you log in, you can add the command at the end of the `.bash_profile` file in your home directory.

2.2.1. Sampling an Application

Applications are sampled using the **sample** command in the ThreadSpotter™ package, usually installed as **/opt/threadspotter/bin/sample**. The only required arguments are the mode of operation and the information needed to start or identify the application:

- **$ sample -r** ***`application and arguments`***

  Run ***`application and arguments`*** and sample it, for example, **$ sample -r ./myapp arg1** to sample the command **myapp arg1**. The application is sampled from the start until it terminates.

  The **-r** option must be the last option to the **sample** command on the command line. All options after the **-r** option are interpreted as options to the application.

  Note that the sampler will only sample the process started by the command you specify, it will not sample any processes that the process starts in turn. This means that if you, for example, specify a script that starts an application, only the script will be sampled and not the application.

- **$ sample -p** ***`PID`***

  Sample the already running process with process id ***`PID`***. The process is sampled until it terminates or until the user stops the sampling.

  This mode of operation is useful to sample a process without having to restart it or to sample only a part of its execution, for example, if an application has a long start up time but you are not interested in the start up code.

You can delay the start of the sampling by a specified number of seconds using the **-d** ***`delay`*** option to the **sample** command. The start of the sampling will then be delayed by ***`delay`*** seconds. This may be useful, for example, if you want to start the application using **sample -r** but do not want to sample the application start up.

You can also start the sampling when a specified function is executed with the **--start-at-function** ***`function`*** option, or at a specified address with the **--start-at-address** ***`address`*** option. You can also start the sampling after the function or address has been executed a number of times using the **--start-at-ignore** ***`count`*** option. See Section 2.4.2, “Sampling Start Conditions” for more information.

2.2.1.1. Burst Sampling

If the part of the application run you intend to sample takes at least 5 minutes to execute, you can enable burst sampling to reduce the sampling overhead. This is done with the **-b** ***`execution time`*** option.

The ***`execution time`*** argument should be the estimated execution time without sampling in minutes of the part of the application that you intend to sample. For example, to sample an application that takes 30 minutes to run without sampling cases:

$ sample -b 30 -r ***`application and arguments`***

See Section 2.4.1, “Burst Sampling” for more information about burst sampling.

2.2.1.2. Stopping the Sampler

When you start the sampler with the **-r** option to the **sample** command the application is started in the same terminal as the sampler, and there is no way to stop sampling without terminating the application.

When you start the sampler with the **-p** option you get a sampler command prompt. The sampler only supports one command, **q**, which stops the sampling but leaves the application running.

Sometimes the sampler will not be able to stop the sampling immediately. This happens if a sampled thread is waiting in a system call. If this happens you can try to "activate" the application to get the thread to finish the system call, for example, if the application has a command line interface and is waiting for user input, entering a command may help.

If that does not help, you can force a detach by giving the sampler the **q** command one more time. Doing this will cause a small memory leak in the sampled process, but that will usually not cause any problems.

In very rare cases it may not be possibly to stop the sampling without corrupting the sampled application. You will get a warning if that is the case.

You can also tell the sampler to automatically stop the sampling after a preset number of seconds by specifying the **-t** ***`duration`*** option to the **sample** command. The sampling will then automatically be stopped after ***`duration`*** seconds. If the process was started using the **-r** option, the process will also be terminated.

In the same way as when starting the sampling at a function or address, you can stop the sampling at a function or address with the **--stop-at-function** ***`function`***, **--stop-at-address** ***`address`*** and **--stop-at-ignore** ***`count`*** options. See Section 2.4.3, “Sampling Stop Conditions” for more information.

2.2.1.3. Changing the Sample File Name

By default the **sample** command will generate a sample file named `sample.smp` in the current directory. If you would like a different file name you can specify the **-o** ***`filename`*** option to the **sample** command. The **sample** command will overwrite any existing sample file with the same name without asking.

2.2.1.4. Selecting Sampled Line Sizes

The sampler will by default collect information for analyzing the application's behavior with 64-byte cache lines, since this is the most common line size. If the cache you are optimizing your application for uses a different line size, you can specify the line size with the **-l** ***`line sizes`*** option to the **sample** command.

For example, if you want to produce a sample file that allows analysis for both 64-byte and 128-byte cache lines:

```
$ sample -l 64,128 -r application
```

You must specify all line sizes for which you will later generate reports based on this sampling.

2.2.1.5. Troubleshooting

If you experience incorrect execution or crashes in the sampled program, try passing the **--safe-stack** option to the **sample** command, see Section 2.4.4.2, “Disabling Application Stack Use by the Sampler”.

2.2.2. Creating a Report

After sampling your application the sampler will have created a sample file, by default named `sample.smp` in the current working directory.

To generate a report from this sample file you run it through the **report** command found in the ThreadSpotter™ package, usually installed as **/opt/threadspotter/bin/report**. The name of the sample file has to be specified with the **-i** ***`filename`*** option:

```
$ report -i sample.smp
```

The **report** command will by default create a report file named `report.tsr` in the current working directory.

2.2.2.1. Specifying the Processor Model

The advice in the report depends on the cache configuration. Some data structures or code sections may, for example, work well with a processor model with a large cache but have problems with another processor model with a smaller cache, or they may work well in the L2 cache but have problems in the smaller L1 cache. To get relevant advice for your application and hardware, you have to specify which processor and cache level you want to optimize for.

The processor model may be specified with the **--cpu *`model`*** option. By default the processor model of computer you run the **report** tool on is selected. A list of available processor models can be printed by specifying **help** as the model name.

The cache level to analyze is selected using the **$ report --level *`level`*** option. ThreadSpotter™ will analyze the highest cache level by default.

It is possible to override parts of a CPU model using options to explicitly set cache size and line size. Refer to report(1) for a description of all available options.

Note

Always specify a processor model and cache level that matches the intended target as closely as possible. Not all model parameters can be overridden at the command line and some of these parameters depend on the cache level.

For example, to analyze the application's behavior with respect to the first level data cache in an Intel Core 2 Quad with 12 MB L2 cache you could use the following command line:

```
$ report --cpu intel/yorkfield_4_12288 --level 1 -i sample.smp
```

Note

You can only use line sizes that the sampler was told to sample when it generated the sample file, by default 64 bytes.

Note

Detecting the current cpu and cache hierarchy does not work reliably in virtualized environments. In this case you need to manual supply the number of caches that ThreadSpotter™ shall assume to be present.

2.2.2.2. Renaming the Report

If you want a different name than `report.tsr` for the report file you can specify a different name with the **-o *`name`*** option to the **report** command.

For example, to generate a report file named `after-opt.tsr`:

```
$ report -i sample.smp -o after-opt.tsr
```

2.2.2.3. Specifying the Location of the Source Code and Binaries

The report generator will try to look up source code files in the location specified in the debug information of the application. However, if the source code has been moved since the application was compiled, you will have to specify the new location of the source with the **-s** ***`source directory`*** option to get a source code view in the report. If multiple parts of the source code have been moved you can specify the **-s** option multiple times.

If the application binaries have been moved from where they were during sampling, the report generator will need to be told the new location to be able to find source code references as described above. The **-b** ***`filename`*** option can be used to name the moved binaries. The **-b** option can be specified multiple times if multiple files have been moved, for example, if both the binary and a shared library it uses has been moved.

2.2.2.4. Selecting the Threshold for Reporting Issues

If you do not specify anything else the generated report will include all locations in the program that are responsible for at least one percent of all cache line fetches, write-backs or upgrades. For a large application this may lead to an excessively large report. To limit the report to issues that contribute to at least a certain percentage of the total cache line fetches of the application, specify the **-p** ***`percentage`*** option to the **report** command.

For example, if you are only interested in issues contributing to at least 5% of the total cache line fetches of the application you could run:

```
$ report -p 5 -i sample.smp
```

2.2.3. Viewing a Report

Once you have created a report you can use the **view** command found in the ThreadSpotter™ package to view it. It is usually installed as **/opt/threadspotter/bin/view**.

The **view** command requires the filename of the report to be specified with the **-i** option. For example, to view a report with the name `report.tsr` you would use the following command:

```
$ view -i report.tsr
```

The report viewer actually starts a web server serving the report as HTML pages on a port on the local machine, and also starts a web browser displaying the report. The web server detects when the report is closed in the browser and will then automatically quit, by default 10 seconds after the report has been closed.

If you do not want the **view** command to start a new web browser you can give it the **--no-browser** option. It will then only start the web server and print a URL that you can use to view the report. Note that the port serving the report is only accessible from the computer on which the **view** command is run, so you can not view the report from a browser on another computer.

By default the **view** command will start the web server on an arbitrary free port on the computer it is running on. If you want the web server to use a specific port you can specify it using the **--port** option.

For example, to start the web server serving the report `report.tsr` on port 2000 without starting a browser, you would run this command:

```
$ view -i report.tsr --port 2000 --no-browser
```

To protect the report from being viewed by other users on the same computer, the URL used to open the report contains a randomly generated password. Once you have opened the report in a browser the password becomes invalid, so you can not open the report again in another browser. If you need to do that simply run the **view** command again.

2.2.3.1. Using a Different Browser

The **view** command will try to start the default browser on the system. If you wish to view the report in another browser, or if the command fails to find a browser, you can set the `BROWSER` environment variable to the browser that should be used. For example, if you are using the **bash** shell and want to use the **google-chrome** browser, you can use this command:

```
$ export BROWSER=google-chrome
```

ThreadSpotter™ has been verified to work with Internet Explorer, Firefox and Google Chrome. If you are using a different browser and experience problems viewing the report, try using one of the supported browsers instead.

2.2.3.2. Using Firefox on Multiple Computers

The Firefox browser, and possibly other Gecko-based browsers, have a default behaviour when used on multiple computers in an X environment that is confusing, and that does not work well with the **view** command.

When you start a Firefox process and specify a URL to open, the process will by default look for other Firefox processes running in the same X server, and if one is found open the URL in that process's window even if that process is running on a different computer.

This does not work with the **view** command for a couple of reasons. The URL used to connect to the report web server uses the address *localhost* to refer to the web server, so if the report is opened on a another computer the URL will refer to the wrong computer. Also, since the report web server only accepts connections from the computer it is running on, Firefox would not be able to open the report from another computer even if it managed to resolve the URL to the correct computer.

The easiest way to avoid this problem is to set the `MOZ_NO_REMOTE` environment variable to 1 before you run the **view** command. For example, if you are using the **bash** shell, you can use this command:

```
$ export MOZ_NO_REMOTE=1
```

This forces Firefox to start a new process on the computer where the command is run, even if there are already Firefox processes running on other computers. However, it will also prevent you from launching multiple browsers viewing the same report on the same computer, so you will have to close the report you are viewing before you can open another report on the same computer.

2.3. Common Report Settings

Some of the performance problems that ThreadSpotter™ looks for are more visible on some cache levels than others, depending on topology, number of caches, and what processor features exist on different cache levels.

For reporting purposes, ThreadSpotter™ considers one cache level at a time. A consequence is that it will focus on those problems that are visible on the selected cache level, and disregard such suggestions that are inapplicable on that level.

Tip

To get the full picture, it may be necessary to prepare multiple reports, one for each cache level.

This section outline some common analysis scenarios, and explain the corresponding settings.

2.3.1. Cache Performance

From a performance optimization perspective, it makes sense to start optimizing with respect to the highest level cache. There are at least two reasons.

A cache miss in the last cache level is much more expensive that a cache miss in the first cache level, since they have to all the way out to the memory to satisfy the memory access. Conversely, if such misses can be avoided, the benefit is most noticeable when optimizing on the highest level.

The highest cache level is also the largest cache. It will be less difficult to squeeze in the data set to fit in the highest cache, than it will be to make it fit in the smaller lower level caches.

This is the default analysis mode of ThreadSpotter™. No extra parameters are required.

2.3.2. Analysis of software prefetch instructions

Depending on the processor, the software prefetch instructions may fetch data to the lowest level cache, or to some outer level.

AMD processors have typically fetched data into the L1 cache level, while Intel processors normally target L2. Generation of prefetch related advice is only active when preparing a report for the corresponding cache level.

Select the target cache level using **--level** ***`cache-level`***. If that cache level is not affected by prefetch instructions, the output from ThreadSpotter™ will include a warning to that effect.

2.3.3. Threading Advice - Inter-Cache Communication

ThreadSpotter™ offers advice and statistics in relation to how well multithreaded programs communicate among their caches. The analysis requires that there are several caches for the analyzed cache level, since if there was only one cache, there would be no other cache to exchange data with.

So, ThreadSpotter™ only presents communication related advice if:

- The application has multiple threads.
- The application threads use memory to share data with each other (as opposed to using operating system communication channels to exchange data). That is, one thread writes and another one reads data from a particular place in memory.
- ThreadSpotter™ deduces or is told that there are more than one cache at the specified cache-level.

The latter point is a function of what processor model is being used, what cache-level is the target cache-level, and whether the user overrides the number of caches to be considered.

Note

ThreadSpotter™ assumes one processor unless you tell it otherwise. If this is not the case, use the setting **--number-of-caches** ***`number`*** to inform ThreadSpotter™ on the total number of caches to assume.

Example: If you are interested in finding problems related to the communication traffic between L1 caches for a dual socket, quad core Intel system (Yorkfield), where each socket has four private L1 caches, use the following parameters:

$ report --cpu intel/yorkfield_4_12288 --number-of-caches 8 --level 1 ...

2.3.4. Threading Advice - Inter-Socket Communication

In a similar way, looking at inter-socket communication requires providing ThreadSpotter™ with the total number of top-level caches in the system.

Example: AMD Opteron 2427, codename Istanbul, has four cores and a shared L3 cache. To analyze the communication between sockets in a system consisting of two such processors, use the following command:

```
$ report --cpu amd/istanbul --number-of-caches 2 --level 3 ...
```

This would cause ThreadSpotter™ to distribute the threads of the application onto two L3 caches. The resulting traffic would correspond to the communication between the two processors.

2.4. Advanced Use

2.4.1. Burst Sampling

By default ThreadSpotter™ will sample the application continuously during the run, collecting information about its memory access pattern. The sampling adds some overhead to the application, making it run slower than usual.

However, for long application runs ThreadSpotter™ can collect enough information without continuous sampling. Instead it can engage periodically during the execution. This is called *burst sampling*. Since a smaller part of the execution run is sampled the overhead of the sampling becomes much lower.

Burst sampling can be used to sample executions that take at least 5 minutes. With runs shorter than that the sampler needs to engage so frequently that the execution becomes more or less continuously sampled, and no reduction in overhead is gained.

With burst sampling the overhead for a sampling run becomes independent of the length of the execution. Sampling a 1-hour application run may, for example, cause a 20 minute overhead, taking 1 hour and 20 minutes. Sampling a 4 hour run of the application would then also cause approximately a 20 minute overhead, taking 4 hours and 20 minutes.

When you enable burst sampling you also need to specify an estimated (normal) execution time for the part of the application run that you intend to sample. ThreadSpotter™ needs this information to calculate how densely the bursts need to be placed to collect enough information for analysis. If ThreadSpotter™ does not capture enough information you will get this warning when the sample file is post-processed:

```
Post-processing sample file, please wait...
Warning: Not enough samples for reliable results.
Got 6252 samples of the largest line size.
Required number of samples: 10000.
Consider decreasing the estimated execution time by 87%.
Sample file post-processing finished.
```

If you get the warning above it does not necessarily mean your estimate of the execution time was inaccurate. You may also get this warning if the overhead of sampling this particular application is unusually low. Adjust the estimated execution time as suggested and sample the application again.

It is possible to tweak the overhead of the burst sampling by adjusting the quality parameter. Switching this parameter to 'fast' yields a lower overhead but may have a negative impact on the data quality. Conversely using 'detailed' quality increases the sampler overhead and will also increase the data quality.

The effects from changes in burst quality are usually limited to applications with low fetch ratios when analyzing large caches. Some issues are more sensitive to the quality setting than others, specifically 'Loop Fusion', 'Blocking' and 'Inefficient Loop Nesting'. The locations affected by these issues will still be flagged as issues, even when the burst quality is too low, but with less specific issue types.

It is usually not necessary to change the burst quality parameter, the default quality should provide an acceptable overhead and good data quality for most common cache sizes.

See Section 2.1.5, “Advanced Sampling Settings” and Section 2.2.1.1, “Burst Sampling” for instructions on how to enable burst sampling in the graphical interface and from the command line, respectively.

2.4.2. Sampling Start Conditions

By default ThreadSpotter™ will start to sample the application as soon as you have launched it or attached to it. However, you may sometimes want to delay the start of the sampling, for example, to avoid sampling the application start up. There are a few ways to do that.

The simplest way is to specify a delay in seconds before the sampling is started. This can be done in the *Advanced sampling settings* dialog if you are using the GUI, or with the **-d *`seconds`*** option if you are using the **sample** command.

You can also specify the sampling to start when a specific function is called. This can again be done in the *Advanced sampling settings* dialog if you are using the GUI, or with the **--start-at-function *`function`*** option if you are using the **sample** command. The sampling is started the first time the specified function is called. Note that when specifying a function to start the sampling at, the function must be in the application binary. The function may not be in a shared library loaded by the application.

If you want to start the sampling at a function in a shared library you can instead determine the address where the function will be loaded and specify that address. You find the address of the function by starting the program and looking up the function in a debugger.

The address where the sampling should be started is specified in the *Advanced sampling settings* dialog if you are using the GUI, or with the **--start-at-address *`address`*** option if you are using the **sample** command. The sampling is started the first time the instruction at the address is executed.

It is also possible to ignore the first few times the function or address is executed. The number of executions to ignore is specified in the *Advanced sampling settings* dialog if you are using the GUI, or with the **--start-at-ignore *`count`*** option if you are using the **sample** command. For example, to start the sampling the fourth time the function `mult` is called you could add the options **--start-at-function mult --start-at-ignore 3** to the **sample** command.

2.4.3. Sampling Stop Conditions

By default the sampling will be ended when the application terminates or when you manually stop the sampling at the sampler prompt. However, you can also stop the sampling after a fixed time or when a specific function or address is executed, just like when controlling when to start the sampling in Section 2.4.2, “Sampling Start Conditions”.

If you are using the GUI you can find the controls for stopping the sampling in the *Advanced sampling settings* dialog. They work just like the controls for starting the sampling.

To stop the sampling after a fixed time using the **sample** command, use the **-t *`seconds`*** option. The options for stopping the sampling when a function or address is executed are **--stop-at-function *`function`***, **--stop-at-address *`address`*** and **--stop-at-ignore *`count`***. They work just like the corresponding options for controlling the start of the sampling.

The same limitation as when specifying a function to start the sampling at still applies, the function must be in the application binary.

If you specify that the sampling should be stopped after a fixed time, that time is counted from when the sampling is started. Similarly, if you specify that the sampling should be stopped after a function or address has been executed a number of times, those executions are counted from when the sampling is started.

2.4.4. Sample Files

A sample file contains the fingerprint of the application memory access behavior collected during sampling. If you have a sample file you can generate a report for that sampling of the application for any cache configuration.

The amount of information captured in a sample file is measured in the number of samples. Too few samples will cause unreliable results, capturing too many samples will cause the sampling of the program to run slower and the report generation to take longer and use more memory. 10000 samples is enough to get a reliable result, and by default the sampler will try to capture 50000 samples.

The report will contain a clearly visible warning if the sample file contained too few samples for a reliable result.

2.4.4.1. Tuning the Sample Period

The sampler automatically adapts to the running program and attempts to capture 50000 samples during the sampling run. There is therefore usually no need to adjust the sample rate.

However, if the program runs for a very short time, a few seconds or less, the sampler may fail to capture enough samples. You will then get a warning like this when the sample file is being post processed:

```
Warning: Not enough samples for reliable results.
Got 3011 samples of the largest line size.
Required number of samples: 10000.
Consider decreasing the sample period to at least 301.
```

To collect enough samples you then have to manually tell the sampler to use a lower sample period than the default of 100000, as suggested in the warning. If you are using the graphical user interface you can specify the sample period in the Advanced sampling options dialog. If you run the **sample** command from the command line you use the **-s** ***`sample period`*** option, for example, to sample **ls -l** with a sample period of 100:

$ sample -s 100 -r ls -l

If you are sampling a program for a long time you may get a small sampling speed up if you increase the sample period from the default. There is no simple rule of thumb for how to set the sample period in this case. Try increasing sample periods, and when you get the warning above you know you have exceeded that maximum sample period that can be used.

2.4.4.2. Disabling Application Stack Use by the Sampler

By default the sampler will use the application stack for temporary storage, since that is faster than alternative storage options. This is safe to do with the vast majority of applications, since the sampler obeys the standard conventions for how the stack should be managed.

However, with a few applications that use the stack in a non-standard way this may lead to incorrect execution or crashes. If you experience such problems you may want to try disabling the sampler's use of the application stack. This will result in slower sampling, but avoids the potential problems.

To disable the sampler's use of the application stack add the **--safe-stack** option to the **sample** command, or check the *Safe stack handling* check box in the *Advanced sampling settings* dialog if you are using the GUI.

2.5. Running ThreadSpotter™ in a Virtualized Environment

When doing its analysis, ThreadSpotter™ will by default try to detect the processor architecture of the machine the analysis is run on and use that for the analysis. This ensures that the analysis results are accurate with respect to the number of cores, threads and cache levels, cache sizes, cache sharing between threads, and so on.

However, when running in a virtualized environment it may not be possible to find a consistent hardware model for the virtual machine. The virtual machine may, for example, only have three virtual processors even though the physical machine it is running on has two quad-core processors. The mapping from virtual processors that virtualized software sees to actual cores or threads in the physical hardware may also change from one moment to the next.

In these circumstances ThreadSpotter™ may not be able to accurately identify the processor architecture. When this happens ThreadSpotter™ will print a warning during the analysis and assume that each virtual processor is a separate single-core, single-threaded processor:

```
$ report -i sample.smp
Info: Failed to auto-detect processors, the number of threads differs
between packages, 1 vs 3. Assuming 4 single-core, single-threaded
processors.
Info: Processor auto-detection failure is often caused by running in
a virtualized environment. It is recommended that you manually
specify the processor model and number of processors you want to do
the analysis for.
...
```

A warning banner will also be included at the top of the generated report.

If this happens you should manually specify the processor model and number of processors you want to perform the analysis for. See Section 2.1.2, “Generating a Report from a Sample File” for how to set the processor model from the graphical user interface, or use the **--cpu** and **--number-of-cpus** options to the **report** command if you are using the command line tools, see Section 2.2.2.1, “Specifying the Processor Model”.

Chapter 3. Introduction to Caches

To be able to optimize a program for efficient processor cache usage some knowledge of how caches work is required. This chapter will give you a basic introduction to caches.

Processors may use un-cached memory regions for low-level I/O. Some processors can map fast private memory banks into parts of the memory space instead of using a cache. There may also be special memory access instructions that do not use the cache that can be used in some situations. This chapter only considers regular memory accesses, and ignores these special cases.

3.1. Motivation for Caches

Modern processors can run at clock speeds of several GHz and are able to execute several instructions per clock cycle. This means that a processor may have a peak execution speed of several instructions per nano-second. For example, a 3 GHz processor capable of executing 3 instructions per cycle has a peak execution speed of 9 instructions per ns.

Modern RAM memories on the other hand are quite slow. An access to RAM memory takes 50 ns or more, causing the processor to stall waiting for the data to arrive. This makes RAM accesses one of the slowest operations a processor can perform. For example, a processor capable of executing 9 instructions per ns could have executed up to 450 instructions in the time it takes to perform a single RAM access with a latency of 50 ns.

The time it takes to load the data from memory is called the *latency* of the memory operation. It is usually measured in processor clock cycles or ns.

Since memory accesses are very common in programs and can account for more than 25% of the instructions, memory access latencies would have a devastating impact on processor execution speed if they could not be avoided in some way.

To solve this problem computer designers have introduced cache memories, which are small, but extremely fast, memories between the processor and the slow main memory. Frequently used data is automatically copied to the cache memories. This allows well written programs to make most of their memory accesses to the fast cache memory and only rarely make accesses to the slow main memory.

Often a computer does not just use a single cache, but a hierarchy of caches of increasing size and decreasing speed. For example, it may have a 64 kilobyte cache with a latency of 3 cycles for the most frequently accessed data, and a 1 megabyte cache with a latency of 15 cycles for less frequently accessed data. The caches in such a configuration are called the first level cache (the 64 kilobyte cache) and the second level cache (the 1 megabyte cache), or shorter the L1 and L2 caches. Some computers may also also have an additional third cache level, the L3 cache.

3.2. Cache Lines and Cache Size

It could be left up to the programmer or compiler to determine what data should be placed in the cache memories, but this would be complicated since different processors have different numbers of caches and different cache sizes. It would also be hard to determine how much of the cache memory to allocate to each program when several programs are running on the same processor.

Instead the allocation of space in the cache is managed by the processor. When the processor accesses a part of memory that is not already in the cache it loads a chunk of the memory around the accessed address into the cache, hoping that it will soon be used again.

The chunks of memory handled by the cache are called cache lines. The size of these chunks is called the cache line size. Common cache line sizes are 32, 64 and 128 bytes.

A cache can only hold a limited number of lines, determined by the cache size. For example, a 64 kilobyte cache with 64-byte lines has 1024 cache lines.

3.3. Replacement Policies

If all the cache lines in the cache are in use when the processor accesses a new line, one of the lines currently in the cache must be evicted to make room for the new line. The policy for selecting which line to evict is called the *replacement policy*.

The most common replacement policy in modern processors is LRU, for *least recently used*. This replacement policy simply evicts the cache line that was least recently used, assuming that the more recently used cache lines are more likely to soon be used again.

Another replacement policy is random replacement, meaning that a random cache line is selected for eviction.

3.4. Cache Misses

When a program accesses a memory location that is not in the cache, it is called a *cache miss*. Since the processor then has to wait for the data to be fetched from the next cache level or from main memory before it can continue to execute, cache misses directly influence the performance of the application.

It is hard to tell from just the number of misses if cache misses are causing performance problems in an application. The same number of misses will cause a much greater relative slowdown in a short-running application than in a long-running one.

A more useful metric is the cache *miss ratio*, that is, the ratio of memory accesses that cause a cache miss. From the miss ratio you can usually tell whether cache misses may be a performance problem in an application.

The cache miss ratio of an application depends on the size of the cache. A larger cache can hold more cache lines and is therefore expected to get fewer misses.

The performance impact of a cache miss depends on the latency of fetching the data from the next cache level or main memory. For example, assume that you have a processor with two cache levels. A miss in the L1 cache then causes data to be fetched from the L2 cache which has a relatively low latency, so a quite high L1 miss ratio can be acceptable. A miss in the L2 cache on the other hand will cause a long stall while fetching data from main memory, so only a much lower L2 miss ratio is acceptable.

A special case is cache misses caused by prefetch instructions, see Section 3.6.1, “Software Prefetching”. Unlike other cache misses these do not cause any stalls, but will instead trigger a fetch of the requested data so that later accesses will not experience a cache miss. In fact, a prefetch instruction should ideally have a high miss ratio, since that means the prefetch instruction is doing useful work.

ThreadSpotter™ therefore does not include misses caused by prefetch instructions when calculating statistics for instruction groups, loops or the entire application.

3.5. Data Locality

As described above, caches work on the assumption that data that is accessed once will usually be accessed soon again. This kind of behaviour is known as *data locality*. There are two kinds of locality that are sometimes distinguished:

- *Temporal locality* means that the program reuses the same data that it recently used, and that therefore is likely to be in the cache.
- *Spatial locality* means that the program uses data close to recently accessed locations. Since the processor loads a chunk of memory around an accessed location into the cache, locations close to recently accessed locations are also likely to be in the cache.

It is of course possible for a program exhibit both types of locality at the same time.

Good data locality is essential for good application performance. Applications with poor data locality reduce the effectiveness of the cache, causing long stall times waiting for memory accesses.

3.6. Prefetching

Even programs with good data locality will now and then have to access a cache line that is not in the cache, and will then stall until the data has been fetched from main memory. It would of course be better if there was a way to load the data into the cache before it is needed so the stall could be avoided. This is called *prefetching* and there are two ways to achieve it, software prefetching and hardware prefetching.

3.6.1. Software Prefetching

With software prefetching the programmer or compiler inserts prefetch instructions into the program. These are instructions that initiate a load of a cache line into the cache, but do not stall waiting for the data to arrive.

A critical property of prefetch instructions is the time from when the prefetch is executed to when the data is used. If the prefetch is too close to the instruction using the prefetched data, the cache line will not have had time to arrive from main memory or the next cache level and the instruction will stall. This reduces the effectiveness of the prefetch.

If the prefetch is too far ahead of the instruction using the prefetched data, the prefetched cache line will instead already have been evicted again before the data is actually used. The instruction using the data will then cause another fetch of the cache line and have to stall. This not only eliminates the benefit of the prefetch instruction, but introduces additional costs since the cache line is now fetched twice from main memory or the next cache level. This increases the memory bandwidth requirement of the program.

Processors that have multiple levels of caches often have different prefetch instructions for prefetching data into different cache levels. This can be used, for example, to prefetch data from main memory to the L2 cache far ahead of the use with an L2 prefetch instruction, and then prefetch data from the L2 cache to the L1 cache just before the use with a L1 prefetch instruction.

There is a cost for executing a prefetch instruction. The instruction has to be decoded and it uses some execution resources. A prefetch instruction that always prefetches cache lines that are already in the cache will consume execution resources without providing any benefit. It is therefore important to verify that prefetch instructions really prefetch data that is not already in the cache.

The cache miss ratio needed by a prefetch instruction to be useful depends on its purpose. A prefetch instruction that fetches data from main memory only needs a very low miss ratio to be useful because of the high main memory access latency. A prefetch instruction that fetches cache lines from a cache further from the processor to a cache closer to the processor may need a miss ratio of a few percent to do any good.

It is common that software prefetching fetches slightly more data than is actually used. For example, when iterating over a large array it is common to prefetch data some distance ahead of the loop, for example, 1 kilobyte ahead of the loop. When the loop is approaching the end of the array the software prefetching should ideally stop. However, it is often cheaper to continue to prefetch data beyond the end of the array than to insert additional code to check when the end of the array is reached. This means that 1 kilobyte of data beyond the end of the array that isn't needed is fetched.

3.6.2. Hardware Prefetching

Many modern processors implement hardware prefetching. This means that the processor monitors the memory access pattern of the running program and tries to predict what data the program will access next and prefetches that data. There are few different variants of how this can be done.

A *stream* prefetcher looks for streams where a sequence of consecutive cache lines are accessed by the program. When such a stream is found the processor starts prefetching the cache lines ahead of the program's accesses.

A *stride* prefetcher looks for instructions that make accesses with regular strides, that do not necessarily have to be to consecutive cache lines. When such an instruction is detected the processor tries to prefetch the cache lines it will access ahead of it.

An *adjacent cache line* prefetcher automatically fetches adjacent cache lines to ones being accessed by the program. This can be used to mimic behaviour of a larger cache line size in a cache level without actually having to increase the line size.

Hardware prefetchers can generally only handle very regular access patterns. The cost of prefetching data that isn't used can be high, so processor designers have to be conservative.

An advantage of hardware prefetching compared to software prefetching is that no extra instructions that use execution resources are needed in the program. If you know that an application is going to be run on processors with hardware prefetching, a combination of hardware and software prefetching can be used. The hardware prefetcher can be trusted to prefetch highly regular accesses, while software prefetching can be used for irregular accesses that the hardware prefetcher can not handle.

3.7. Multithreading and Cache Coherence

Computers with multiple threads of execution, either with multiple processors, multiple cores per processor, or both, introduce additional complexity to caches. Different threads accessing the same data can now have private copies of the data in their local caches, but writes to the data by one thread must still be seen by all other threads. Some mechanism to keep the caches synchronized is needed.

The mechanism that keeps the caches synchronized is called a *cache coherence* protocol. There are different possible coherence protocols, but most modern processors use the *MESI* protocol or some variation such as the MOESI protocol. ThreadSpotter™ therefore currently models the *MESI* protocol.

The MESI protocol is named after the four states of a cache line: modified, exclusive, shared and invalid:

Cache line states in the MESI protocol

M - Modified	The data in the cache line is modified and is guaranteed to only reside in this cache. The copy in main memory is not up to date, so when the cache line leaves the modified state the data must be written back to main memory.
E - Exclusive	The data in the cache line is unmodified, but is guaranteed to only reside in this cache.
S - Shared	The data in the cache line is unmodified, and there may also be copies of it in other caches.
I - Invalid	The cache line does not contain valid data.

This section is not intended to be a complete description of cache coherency, but only a quick introduction allowing you to understand the types of multithreading problems ThreadSpotter™ can identify. Good sources with more detailed information on cache coherency can be found by searching on the internet.

We will now go through some examples of how the MESI protocol works. For simplicity we will assume that we are running on a two-processor system where each processor has its own private cache on a single cache level:

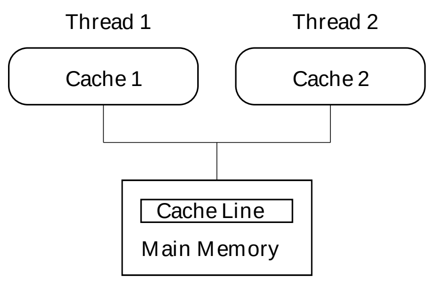


Figure 3.1. Example System

In these examples data transfers are drawn in red, while downgrade and invalidation traffic is drawn in blue.

If a thread reads data not present in any cache, it will fetch the line into its cache in exclusive state (E):

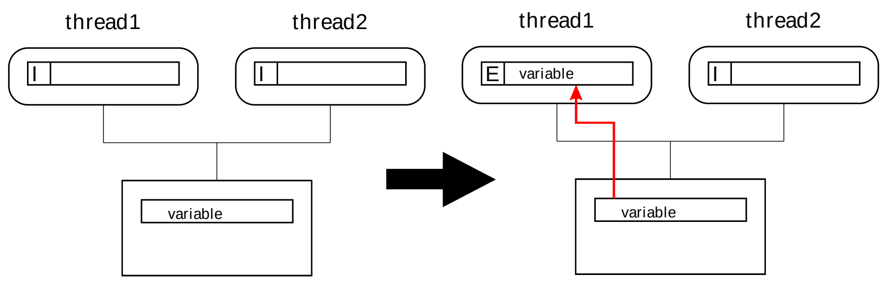


Figure 3.2. Cache Coherence, Example 1

If a thread reads from a cache line that is in shared state (S) in another thread's cache, it fetches the cache line into its cache in shared state (S):

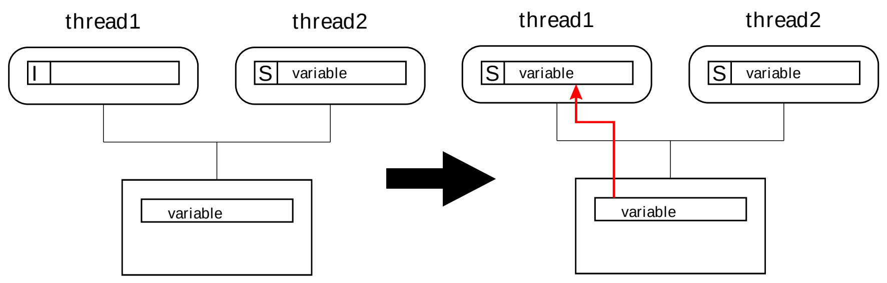


Figure 3.3. Cache Coherence, Example 2

If a thread reads from a cache line that is in exclusive state (E) in another thread's cache, it fetches the cache line to its cache in shared state (S) and downgrades the cache line to shared state (S) in the other cache:

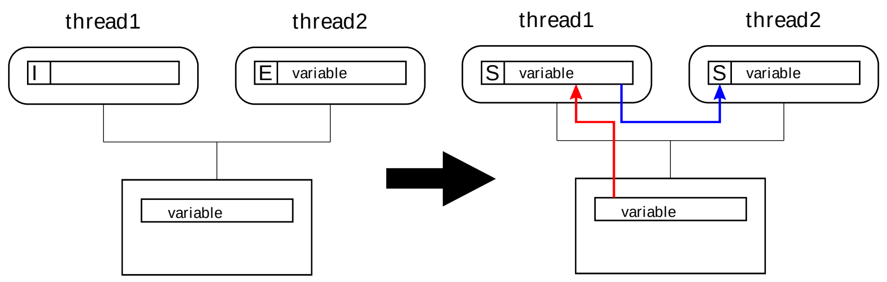


Figure 3.4. Cache Coherence, Example 3

If a thread reads from a cache line that is in modified state (M) in another thread's cache, the other cache must first write-back its modified version of the cache line and downgrade it to shared state (S). The thread doing the read can then add the cache line to its cache in shared state (S):

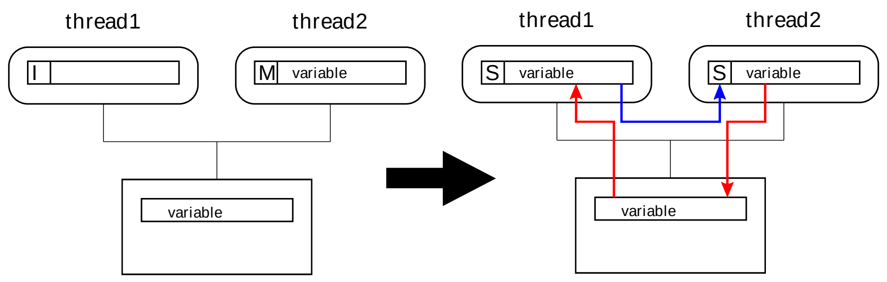


Figure 3.5. Cache Coherence, Example 4

When a thread has a cache line in exclusive (E) or modified state (M) it can write to it with very low overhead, since it knows that no other thread can have a copy of the line that needs to be invalidated. A write to an exclusive cache line makes it modified (M):

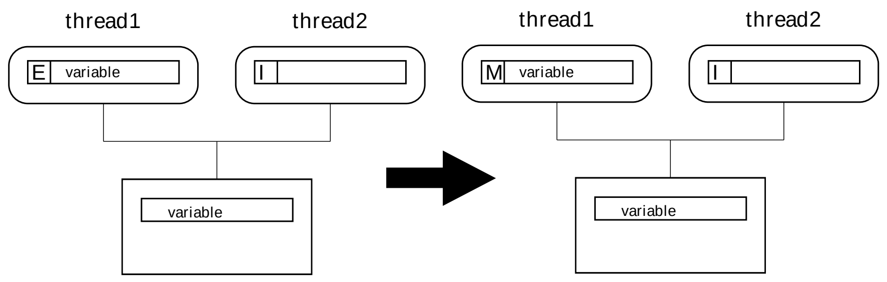


Figure 3.6. Cache Coherence, Example 5

When a thread writes to a cache line that it has in shared state (S) it must upgrade the line to modified state (M). In order to do this it must invalidate (I) any copies of the line in other caches, so that they do not retain an outdated copy:

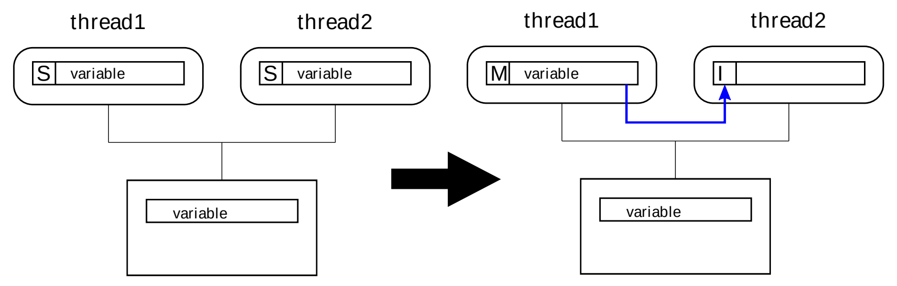


Figure 3.7. Cache Coherence, Example 6

When a thread writes to a cache line that it does not have in its cache, it must fetch the line and invalidate (I) it in all other caches. If another thread has a modified (M) copy of the cache line it must first write it back before the thread doing the write can fetch it.

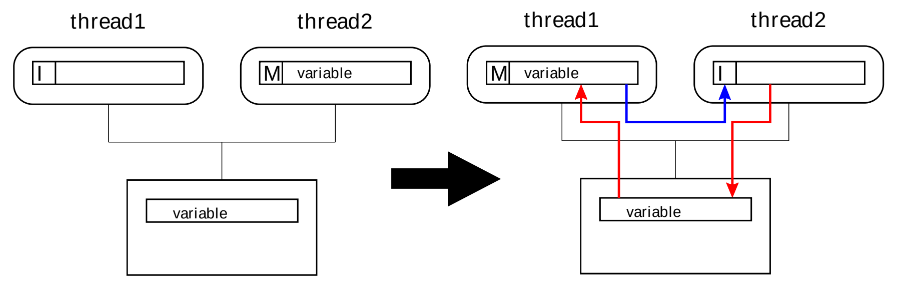


Figure 3.8. Cache Coherence, Example 7

The above is not a complete enumeration of coherence interaction, and in reality there are other access types to consider such as prefetches and cache line flushes, but it should give basic understanding of have cache coherence affects.

Based on the coherence related events, ThreadSpotter™ presents statics about the following expensive thread interactions:

Upgrades	The upgrade count is the number of memory accesses that cause a cache line to be upgraded from shared state to either exclusive or modified state. There are two scenarios where this can happen. A thread can read a cache line into its cache in shared state because it is already in the cache of another thread, and then write to it. Or, a thread has the cache line in its cache in exclusive or modified state, but it is downgraded to shared state because of a read from another thread, and the thread then writes to the line.

Coherence misses	The coherence miss count is the number of memory accesses that miss because a cache line that would otherwise be present in the thread's cache has been invalidated by a write from another thread.
Coherence write-backs	The coherence write-back count is the number of memory access that force a cache line that is in modified state in another thread's cache to be written back.

All of the above can also be reported as ratios. They are then reported as the percentage of memory accesses that suffer from one of these interactions. For example, a coherence miss ratio of 3% means that on average 3 out of every 100 memory accesses suffer from coherence misses.

3.8. Fetch Ratio

In a processor without any kind of prefetching the cache miss ratio will equal the ratio of memory accesses that cause a fetch from main memory. However, with prefetching the number of cache misses may be much lower than the number of accesses the processor does to main memory.

Prefetching therefore introduces a new concept, the fetch ratio. This is the ratio of memory access instructions that cause a fetch from main memory. Note that writes normally cause a fetch of a cache line even though the data in the line isn't actually used.

The fetch ratio directly reflects the memory bandwidth requirement of the application's read accesses, see below.

3.9. Upgrade Ratio

The ratio of memory accesses that cause an upgrade (see general description of upgrades and other coherence related concepts in Section 3.7, “Multithreading and Cache Coherence”.)

If two threads, having their own private caches, alternatingly update a piece of shared memory, the corresponding cache lines will repeatedly change ownership and their state in the caches will also change. Specifically, when a thread writes to a cache line that was owned by a different cache, the cost of a cache line upgrade is imposed on the thread performing the write operation. This is expensive, well in parity with the cost of a cache miss.

3.10. Write-Back Ratio

Similar to fetch ratio, the write-back ratio informs about the likelihood that a write instruction actually causes data to be written back to memory. This is also directly related to the bandwidth requirements of the application.

3.11. Memory Bandwidth

It is not only the latency of the memory accesses that can limit the execution speed of an application. There is also a limited ratio at which data can be read from main memory, the memory bandwidth limit.

When the memory bandwidth limit is reached it is no longer the latency of the memory accesses that limits the execution speed, but the number of accesses to main memory that the application causes. The only way to improve performance is then to reduce the fetch ratio of the application, so that the number of main memory accesses is reduced.

When a program hits the memory bandwidth limit some optimizations intended to reduce the impact of memory access latencies become ineffective, or even reduce performance. For example, prefetching does not reduce the number of main memory accesses, so it loses its effect. Instead, prefetching often leads to more main memory accesses since some data that isn't used is usually prefetched, decreasing the performance of the application.

For single-core processors it is relatively unusual that applications hit the memory bandwidth limit. Very high fetch ratios are required for this to happen. However, with the current trend towards multithreaded and multicore processors the computation power of processors is increasing very rapidly, while the memory bandwidth is increasing much more slowly. This makes the memory bandwidth limitation one of the biggest problems for scaling application performance on multicore processors.

Applications that manage to stay below the memory bandwidth limit on single-core processors and having problems that were hidden by hardware or software prefetching, may suddenly hit the memory bandwidth limit when parallelized on multicore processors. It is not unusual that applications get no speed-up at all. Some applications even get reduced performance, since threads start throwing each other's data out of the cache, further increasing the fetch ratio.

Chapter 4. ThreadSpotter™ Concepts

In addition to the concepts such as cache sizes, cache lines sizes, miss ratios and fetch ratios commonly associated with caches, ThreadSpotter™ also uses some other concepts.

4.1. Issues

When ThreadSpotter™ finds a problem area in the application it reports it as an issue in the report. The issue tells you what kind of problem has been found and points you to where in the source code it can be found.

For each issue ThreadSpotter™ points you to the instruction group related to the issue, the loop containing the issue and list some related statistics. Exactly what information is associated with the issue is varies with the issue type.

See Chapter 8, *Issue Reference* for more information about issues.

4.2. Loops

A loop in ThreadSpotter™ is just what it sounds like, a loop in the program. A *for* loop in a C program will usually translate to a loop in a ThreadSpotter™ report. However, since ThreadSpotter™ looks at the binary code of the application, the loops seen by ThreadSpotter™ may not always exactly correspond to the loops in the source code.

Since ThreadSpotter™ focuses on memory and cache performance it only lists the memory accesses when listing the instructions in loops, other types of instructions are ignored. However, some instructions that you usually would not consider to be memory accesses are included, for example, function call and return instructions may save and restore the return address on the stack and therefore be included.

Loops may be related to other loops, in the sense that one loop may be contained inside one or more outer loops, just like a for loop in C may be nested inside another for loop. This way a hierarchy of inner and outer loops is created.

Most reported issues are associated with a loop in the application.

4.3. Instruction Groups

An instruction group in ThreadSpotter™ is a number of instructions within a loop that touch the same data structure. Or more accurately, that touch the same cache lines. Instruction groups are a natural and useful unit when looking the behavior of the program.

An issue is usually related to a certain instruction group. For example, a loop may copy data from one data structure to another. It would then contain one instruction group reading from the source data structure and one instruction group writing to the destination data structure. If there is a problem in one of the data structures the issue would point you to the source code lines of the instructions making up the instruction group accessing that data structure.

4.4. Last Writer

The report generated by ThreadSpotter™ points the user to the source code lines containing the memory accesses that cause each issue, as described above. However, a single source code line may contain complex expressions that contain many memory references, and it may not be obvious which one of these it is that is causing the issue.

To make it easier for the user to determine which memory accesses or data structure in a large expression is causing an issue, the report also displays the previous location at which that particular memory location was written. Usually only one variable is written on each line, which directly tells the programmer which variable in the original expression is involved in the issue.

4.5. Fetch Utilization

The fetch utilization indicates how large fraction of the data brought into the cache that is actually read before the line is evicted. The value is proportional to the bandwidth required for loads.

A low fetch utilization means that bandwidth is wasted by loading data that is never used. A large portion of the cache is also tied up storing data never being read. When optimizing software you should strive to maximize the fetch utilization.

Remember that not only explicit data loads cause a cache line to be fetched, stores normally cause a cache line fetch as well. It is therefore possible to get a 0% fetch utilization, which would indicate that a cache line is fetched but no part of the line is read before it is evicted from the cache.

4.6. Write-Back Utilization

When an application writes data, the changes are first stored in the cache, and not until the cache line is evicted is the entire cache line written back to memory. If only part of the cache line is written to, then precious bandwidth is consumed to write-back data that has not changed.

Write-back utilization expresses how much of the cache line that has been updated. You should strive to get a 100% write-back utilization to minimize the amount of wasted bandwidth.

4.7. Communication Utilization

When two threads share data, the ownership of the cache line is moving between the two threads' respective caches as the two threads access the data. The transfer of ownership takes some time, and one should strive for arranging data and adjust algorithms so that it doesn't occur too frequently.

Similar to fetch utilization and write-back utilization, the Communication utilization measures efficiency. In this case, it measures the percentage of the cache line that is being written by the producing thread and then actually being consumed by the receiving thread. You should arrange data so that all of the cache line gets consumed before the producer writes to the cache line again.

A special case of low communication utilization is known as false sharing (see Section 5.4.1, “False Sharing”), where the two threads don't access any overlapping data.

4.8. Utilization Corrected Fetch Ratio

The utilization corrected fetch ratio is an estimation of what the fetch ratio would be if the application was changed so that 100% fetch utilization was achieved, for example, by changing a data structure to a more compact representation. It does not take the effect of other suggested changes into account, for example, blocking advice.

The utilization corrected fetch ratio can be displayed for the entire application, a loop or an instruction group. When it is displayed for a loop or an instruction group it is the utilization corrected fetch ratio of that specific part of the code if that specific part of the code was fixed.

4.9. Utilization Corrected Write-Back Ratio

The utilization corrected write-back ratio is an estimation of what the write-back ratio would be if the application was changed so that 100% write-back utilization was achieved.

The utilization corrected write-back ratio can be displayed for the entire application, a loop or an instruction group. When it is displayed for a loop or an instruction group it is the utilization corrected write-back ratio of that specific part of the code if that specific part of the code was fixed.

4.10. Hardware Prefetch Probability

Modern processors detect regular access patterns in applications, and use this information to prefetch data into the cache before it is needed by the application. This can hide the memory accesses latencies and avoid the stalls that the cache misses would otherwise have caused.

ThreadSpotter™ models a generic hardware prefetcher and estimates the percentage of the cache misses that can be avoided by the hardware prefetcher. This number assumes an idle memory bus and does not take the memory bandwidth limitation into account.

The hardware prefetch probability can be used to judge if addressing an issue will be worthwhile, and have noticeable effect, or if the processor is likely able to handle the problem by itself. Fixing an issue with a high hardware prefetch probability may not result in a performance improvement, since the processor is likely to able to prefetch the data and therefore avoid cache misses.

However, if the application runs into the memory bandwidth limitation the prefetching will be ineffective and fixing the issue will improve performance anyway.

Another use for the hardware prefetch probability is to find data structures that are not effectively prefetched by the hardware prefetcher, and try to reorganize them so that they can be effectively prefetched.

4.11. Access Randomness

ThreadSpotter™ estimates the randomness of the memory accesses patterns of loops and instruction groups. It is reported as either *low*, *high* or *very high*.

When you optimize a program you should try to minimize the random access patterns in the program. Random access patterns are generally harmful to application performance. See Section 5.2.2, “Random Access Pattern”.

4.12. Call Stack

ThreadSpotter™ records information about how functions in the application call each other. The chain of functions leading up to a point of interest is called a call stack.

The call stack tells you which functions have called which to reach the point in the program you are looking at. This is useful to know when trying to reduce the number of calls to a function that causes large numbers of cache misses.

ThreadSpotter™ augments source code and instruction listings in the report with call stack information, and visualizes the different call stacks in the source code annotation by color coding them differently.

A function may behave differently when called from different locations. It may for example work well when called with a small data set from one location, but behave badly when called with a larger data set from another location. ThreadSpotter™ therefore optionally uses the call stack information to separate different call sites from each other and give individual advice for the function when called from each call site.

The number of levels of calling functions used to differentiate between call sites is called the *call stack depth*. The call stack depth used in the analysis can be varied to suit your application. By default ThreadSpotter™ uses a call stack depth of 1, that is, only the innermost calling function is considered in the analysis.

A lower call stack depth reduces the number of reported issues by merging different call sites. A call stack depth of 0 will completely disregard where functions are called from. A higher call stack depth generally increases the number of issues and their accuracy, but may also make the report harder to overview.

When attaching to a running process, the sampler does not have complete knowledge of the call stack at that point. This may result in several contexts being evaluated; one for each discovered version of the callstack. The same thing can happen when burst-sampling the application. The recommendation for these cases is to reduce the depth of the call stack begin part of the analysis.

4.13. Sample Period

The sample period is a measure of how dense information about the memory access patterns of the application is captured by ThreadSpotter™ when sampling an application. A lower sample period means capturing more information. ThreadSpotter™ automatically adapts the sample period to the application being run, so there is usually no need to care about the sample period.

However, for applications that run for a very short time, a few seconds or less, you may need to manually decrease the sample period. For applications that run for a long time there may also be some performance to be gained from increasing the sample period. See Section 2.4.4.1, “Tuning the Sample Period”.

Chapter 5. Memory Performance Problems and Solutions

This chapter presents common causes for cache and memory related performance problems and solutions to these problems. The problems can be divided into two categories, data layout problems and data access pattern problems. We also present some common data structures and problems that they may cause.

This chapter uses some memory layout graphs to clarify what is happening in the cache. Cache lines are drawn with solid lines. Boundaries between data fields are drawn with thinner dashed lines. Used parts of a cache line are drawn in green, and unused parts of touched cache lines in red. Completely unused cache lines are drawn in white. For example, two cache lines where the first half of the first cache line is used and the second cache line is completely unused would look like this:

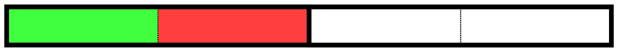


Figure 5.1. Data Layout Example

5.1. Data Layout Problems

Data layout problems is a large class of related problems that originate in how the programmer or the compiler organizes the variables and objects that the program uses in memory. This is affected by choice of data structures and data types, as well as the ordering of objects within such structures.

The common property of all data layout problems is that data that is frequently used is mixed in the same cache line with data that is rarely or never used. There are many reasons why this may happen.

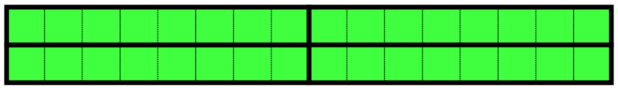


Figure 5.2. Good Utilization

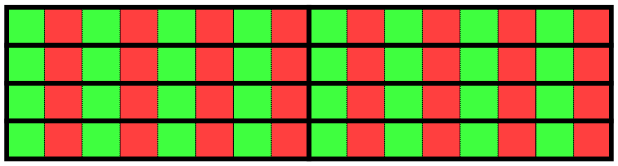


Figure 5.3. Poor Utilization

Bad data layout is penalized twice:

- Bad data layout increases the number cache lines needed to hold the frequently used data, which increases the number of cache misses and cache line fetches.

For example, assume that a part of an application frequently needs to sequentially go through a large number of 8-byte values, and that the processor uses 64-byte cache lines. If the data is optimally packed 8 of these values can be stored in each cache line, but if it is interleaved with other 8-byte values only 4 values can be stored in each cache line. See Figure 5.2, “Good Utilization” and Figure 5.3, “Poor Utilization”. With the poorly packed data twice as many lines need to be accessed when going through the values, and with the same miss ratio it causes twice as many cache misses.

- Bad data layout also increases the cache miss ratio by reducing the *effective* cache size, that is, if data is poorly packed less *useful* data will fit in a cache of a certain size. This way bad data layout increases the cache miss ratio.

  For example, in the example above twice as many of the frequently accessed values in the cache in the optimally packed case compared to the interleaved case.

Since bad data layout increases the number of cache line fetches it also increases the memory bandwidth requirement of the application. For applications that are limited by the available memory bandwidth this can have a serious performance penalty.

5.1.1. Partially Used Structures

A common cause for poor data layout is structures that are only partially used. Modern programming paradigms call for people to collect related data in structures or objects. However, this is often harmful to cache performance.

Consider the following example. Here we have a structure we four fields *a*, *b*, *c* and *d*. All the fields may be used somewhere in the application, but in the most performance critical loop only *a* and *b* are used.

```
struct record {
  int a;
  int b;
  int c;
  int d;
};

struct record r[SIZE];

for (int i = 0; i<SIZE; i++) {
  r[i].a = r[i].b;
}
```

Nevertheless, the unused fields are brought into the cache from memory, doubling the number of cache line fetches and occupying half of the cache space.

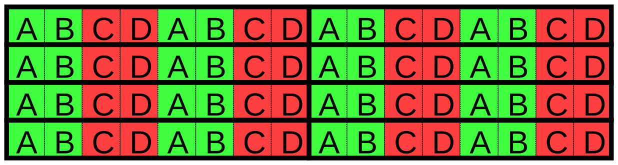

A B C D A B C D A B C D A B C D
A B C D A B C D A B C D A B C D
A B C D A B C D A B C D A B C D
A B C D A B C D A B C D A B C D

Figure 5.4. Unused Fields

If this code can be rewritten so that the rarely used fields are moved to a separate structure:

```
struct record_ab {
  int a;
  int b;
};

struct record_cd {
  int c;
  int d;
};

struct record_ab r_ab[SIZE];
struct record_cd r_cd[SIZE];

for (int i=0; i<SIZE; i++) {
  r_ab[i].a = r_ab[i].b;
}
```

The performance critical loop will now only load the fields that are actually used into the cache.

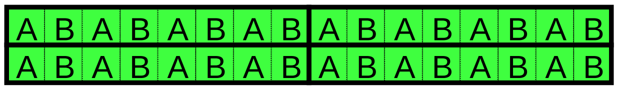

A B A B A B A B A B A B A B A B
A B A B A B A B A B A B A B A B

Figure 5.5. No Unused Fields

This optimization should only be used when it is actually needed. Most programmers will find the modified code ugly and harder to read, and the change goes against all that is taught about programming. However, when used carefully this optimization can result in very large performance improvements.

5.1.2. Too Large Data Types

A problem very similar to having unused fields is to use larger data types than necessary. The effect is the same. Fewer fields fit into each cache line, causing more cache line fetches and a higher cache miss ratio.

For example, if a 64-bit data type is used where a 16-bit data type would be sufficient, only a forth as many elements will fit into a cache line.

Unfortunately ThreadSpotter™ is not able to determine if data types are larger than necessary. The generated reports will therefore not warn about this type of problem, and you have to manually check the code for it.

5.1.3. Alignment Problems

Many modern processors require memory accesses to be made to aligned addresses. Some processors support unaligned accesses, but the performance of aligned memory accesses are usually still better.

Most processors require *natural alignment*. This means that a field has to be stored at an address that is a multiple of its size, for example, a 4-byte field has to be stored at an address that is a multiple of 4 and an 8-byte field has to be stored at an address that is a multiple of 8.

Compilers are aware of this and insert padding between variables and fields to ensure that each field starts at an address with correct alignment for its particular data type. This padding consumes cache space, and this waste can often be avoided or minimized by careful ordering of structure members. Consider this example:

```
struct record {
  char a;
  int  b;
  char c;
};
```

Assume that char is a 1-byte data type and int is a 4-byte data type. The compiler will then lay out data three bytes of padding between fields *a* and *b* to ensure that *b* is stored at an offset that is a multiple of 4.

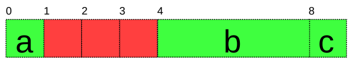


Figure 5.6. Poor Internal Alignment

If you move the field *a* after *b* the alignment requirements of all fields can be satisfied without an padding.

```
struct record {
  int  b;
  char a;
  char c;
};
```

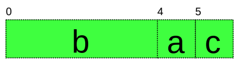


Figure 5.7. Good Internal Alignment

You can of course count offsets into the structure yourself and assure all fields get proper alignment without any padding, but a simpler way to ensure that there is no unnecessary padding is to simply sort the fields by their alignment requirements. Start with the fields with the greatest alignment requirements and then continue with the fields in declining alignment requirement order.

If your structure is so large that it uses more than one cache line you may however want make sure that the most commonly used fields are close together, so that you avoid mixing frequently used fields and rarely used fields as described in Section 5.1.1, “Partially Used Structures”.

We have now eliminated the internal alignment padding in the structure. However, there is also an alignment requirement on the structure as a whole. Consider this example where an array of structures is created:

```
struct record {
  int  b;
  char a;
  char c;
};

struct record v[1000];
```

The structure may, for example, require 8-byte alignment. Since the structure is 6 bytes large, the compiler has to insert 2 bytes of padding between each structure in the array to ensure alignment.

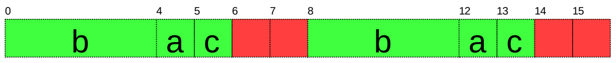


Figure 5.8. External Alignment

External alignment like this is harder to avoid than internal alignment. One way may be to break out less frequently used fields as described in Section 5.1.1, “Partially Used Structures”.

If the processor does not strictly require alignment but simply prefers it for better performance, like, for example, x86 processors, the compiler may provide pragmas that you can use to tell it to not insert any padding. The performance increase from reduced cache misses may outweigh the cost of the unaligned memory accesses.

Using unaligned structures like that may however cause other problems, for example, shared libraries compiled with such pragmas may be incompatible with code compiled with other compilers.

5.1.4. Dynamic Memory Allocation

Many implementations of standard memory allocators reserve extra housekeeping data adjacent to each piece of allocated memory. This data is typically only used when allocating and deallocating the areas, and is unused for the larger part of the application execution.

If many small memory allocations are done by the application a significant part of the cache lines can be occupied by this housekeeping data. Consider an application allocating 32-byte structures with a memory allocator that allocates 16 bytes of data for housekeeping:

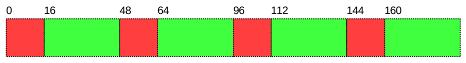


Figure 5.9. Dynamic Memory Allocation

This problem can be avoided by avoiding allocating many individual objects. If you need to allocate 1000 objects, allocate an array with 1000 objects instead of allocating 1000 individual objects.

You may even want to consider implementing your own custom memory management. It could allocate a sizable chunk of memory, and then hand out memory regions within this block. This way the overhead is kept at a minimum.

There are also alternative memory allocation packages available that allocate their housekeeping data in a different memory region than the allocated memory to avoid this kind of problem.

Dynamic memory allocation may also spread allocated memory regions over the heap causing random access patterns, see Section 5.2.2, “Random Access Pattern”.

5.2. Data Access Pattern Problems

Data access patterns also affect cache and memory related performance. Changing the order in which various data items are accessed can be as rewarding as optimizing the data layout. There are several variations of inefficient access patterns, each with a different set of remedies.

5.2.1. Inefficient Loop Nesting

Inefficient loop nesting is a problem specific to algorithms that work on multidimensional arrays. If the array is traversed along the wrong axis only one element will be used from each cache line before the program continues to the next cache line.

For example, consider this small snippet of C code iterating over a 2-dimensional matrix:

```
double array[SIZE][SIZE];

for (int col = 0; col < SIZE; col++)
  for (int row = 0; row < SIZE; row++)
    array[row][col] = f(row, col);
```

In one iteration of the outer loop the inner loop will access one element in each row of a specific column. For example, for a 8-by-8 matrix:

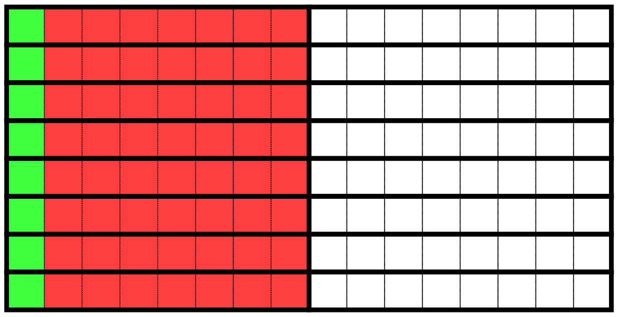


Figure 5.10. Inefficient Loop Nesting

For each element touched by the loop, a new cache line is accessed. The first level cache of a processor may not hold more than a few hundred cache lines. If the array has more than a couple of hundred rows, the rows from the top of the matrix may already have been evicted from the cache when bottom of the matrix is reached. The next iteration of the outer loop then has to reload all of the cache lines again, causing a cache miss for each element that is touched. This is completely devastating to the performance of the loops.

Since only one element from each fetched cache line is used, incorrect loop nesting also wastes a lot of memory bandwidth on fetching data that isn't used.

The problem can be fixed by changing the nesting of the loops so that they traverse the matrix along the rows instead of along the columns.

```
double array[SIZE][SIZE];

for (int row = 0; row < SIZE; row++)
  for (int col = 0; col < SIZE; col++)
    array[row][col] = f(row, col);
```

Each iteration of the outer loop now uses all of the elements in each cache line and the same cache lines will not have to be reused by later iterations, resulting in greatly reduced cache misses and cache line fetches.

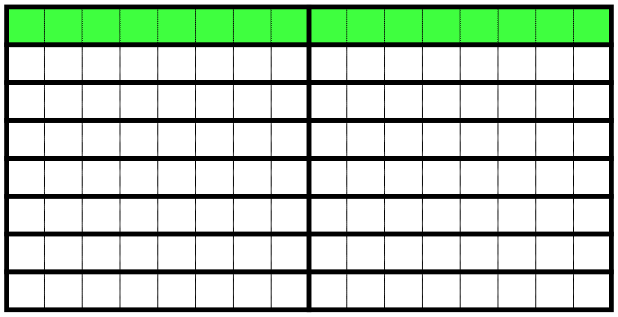


Figure 5.11. Efficient Loop Nesting

Accessing data sequentially also helps the hardware prefetcher to do its job well. This will maximize the use of available memory bandwidth.

If it is not possible to change the loop nesting without changing the calculation result an alternative is to transpose the matrix. By changing the data layout the access patterns become more regular, making better use of fetched cache lines and helping the hardware prefetcher to do a better job. The cost of the incorrect loop nesting may far outweigh the cost of transposing the matrix.

Incorrect loop nesting is not exclusive to 2-dimensional matrices. The same effect can occur when traversing any regular multi-dimensional array, for example, in a 3-dimensional matrix.

Fortran organizes its multi-dimensional arrays in the opposite direction compared to C, so if you are programming in Fortran you would want to iterate over the matrix along the columns instead of along the rows.

5.2.2. Random Access Pattern

Random memory access patterns are generally harmful to cache and memory performance. With random access patterns we do not mean truly random accesses, but generally accesses that are not to sequential addresses, and that are not to the same data or close to data that has recently been used.

As described in Chapter 3, *Introduction to Caches*, caches work on the expectation that data that has recently been touched and data close to data that has recently been touched is more likely to be accessed again. Random access patterns are bad because they generally contain little such reuse, which means there is a low probability to find the accessed data in the cache.

Random access patterns often also lead to a low cache line utilization. Individual data elements in cache lines may be accessed without touching the rest of the cache line. This increases the fetch ratio and memory bandwidth requirement of the application in relation to the amount of data it actually uses.

The hardware prefetcher present in modern processors also relies on finding regular access patterns to determine what data to prefetch, and thereby hide memory access latencies. Random access patterns therefore make the hardware prefetcher ineffective. Certain types of irregular access patterns may even trick the hardware prefetcher into prefetching data that is not useful, wasting memory bandwidth.

Even accessing main memory is faster when the access is to an address close to the last address that was accessed.

Random access patterns can originate from different sources:

- Data structures
- Dynamic memory allocation
- Algorithms

Some *data structures* inherently cause random access patterns.

For example, hash tables are often designed to spread elements randomly throughout the table to avoid collisions. Element lookups in the table therefore a cause a random access pattern.

Tree structures are another common source of random access patterns. Lookups of different keys in the tree cause traversals of different paths through the tree, causing seemingly random memory access patterns.

Replacing such data structures with more cache friendly data structures may provide performance improvements. See Section 5.5, “Common Data Structures” for more information .

Dynamic memory allocation can contribute to placing related data objects far away from each other. Memory allocators usually attempt to allocate memory regions sequentially, but as the application runs and memory on the heap is allocated, deallocated and allocated again the heap may become fragmented. Allocated memory regions may then be spread out over the heap.

Data structures that are allocated incrementally while the application runs are also likely to be spread out over the address space because of other allocations made by the application.

An example of a random access pattern caused by dynamic memory allocation could be a linked list created from newly allocated memory regions. The memory regions may be spread out more or less randomly through the address space. When the application traverses the list it would therefore cause memory accesses in a more or less random pattern. If elements are incrementally added to the list during the execution that is likely to further worsen the problem because of memory fragmentation.

If random memory access patterns because of dynamic memory allocation is a problem, it may be possible to implement a custom memory allocator that puts related data close together. If you, for example, have a data structure that is allocated incrementally during the execution, you could allocate a pool of memory for that data structure when the application starts and then use memory from that pool for the incremental allocations.

Finally some *algorithms* may inherently cause irregular access patterns.

Many graph algorithms are examples of such algorithms. If you represent a graph as an adjacency matrix the data structure itself is very regular. However, if you perform a depth first or breadth first search in the graph the algorithm will cause irregular accesses to the adjacency matrix as it follows the edges between the graph nodes.

Fixing random access patterns caused by algorithms is generally hard. There may be alternative algorithms with better cache behaviour, but they may be inferior in other ways. Sometimes it may be possibly to adapt a data structure to the algorithm, for example, it may be possible to sort an adjacency matrix so that nodes that are connected are close to each other, so that following the edges results in a more regular access pattern.

5.2.3. Unexploited Data Reuse Opportunities

To use a cache optimally an application should, once it has loaded a cache line into the cache, perform as much work as possible involving that cache line before it moves on to other data. This way the number of times the cache line has to be loaded into the cache is minimized, and thereby the number of cache line fetches and the probability of cache misses are reduced.

To achieve this there are two basic techniques. If the data set is large, try working on a small part of the data set at a time so that that part is reused as much as possible. If there are several operations that should be performed on a data set then perform them all at once on each element, instead of performing them one at a time on all elements.

ThreadSpotter™ currently detects two types of such data reuse opportunities, blocking and loop fusion.

5.2.3.1. Blocking

Blocking refers to a class of techniques that aim to break down the original problem into small chunks, so that all data needed to process each chunk fits in the cache. This way that data can be optimally reused before the next chunk is processed.

Blocking can be performed to optimize reuse of adjacent data, *spatial blocking*, or to optimize for reuse of the same data, *temporal blocking*, or both.

An example of a problem where spatial blocking can be used is creating a transposed copy of a matrix:

```
void transpose(double dst[SIZE][SIZE], double src[SIZE][SIZE]) {
  int i, j;
  for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      dst[j][i] = src[i][j];
}
```

We see that the `src` matrix is read one row at a time and that the `dst` matrix is written one column at a time. If the matrix is large enough there won't be enough cache lines for all the rows accessed in the `dst` matrix in one iteration of the outer loop, and we get a very inefficient accesses pattern as described in Section 5.2.1, “Inefficient Loop Nesting”.

We could change the nesting of the loops as suggested there:

```
void transpose(double dst[SIZE][SIZE], double src[SIZE][SIZE]) {
  int i, j;
  for (j = 0; j < SIZE; j++)
    for (i = 0; i < SIZE; i++)
```

```
      dst[j][i] = src[i][j];
}
```

However, while this fixed the problem with the `dst` matrix the `src` is instead read one column at a time, so we have only moved the problem.

What we can do is to only copy a few columns from the `src` matrix at a time, so that we know there are enough cache lines to hold the corresponding rows in the `dst` matrix. Each such set of columns becomes a block.

```
void transpose(double dst[SIZE][SIZE], double src[SIZE][SIZE]) {
  int jb, i, j;
  for (jb = 0; jb < SIZE; jb += BLOCK)
    for (i = 0; i < SIZE; i++)
      for (j = jb; j < jb + BLOCK && j < SIZE; j++)
        dst[j][i] = src[i][j];
}
```

The variable `jb` now keeps track of which columns are being copied from the source matrix in each block. In the first iteration of the outermost loop the columns `0` to `BLOCK-1` are copied, in the next iteration the columns `BLOCK` to `2*BLOCK-1` are copied, and so on.

By choosing a small enough value for the block size `BLOCK` we can make sure that the corresponding rows written to the `dst` matrix in each iteration of the middle loop fit in the cache.

Temporal blocking is sometimes possible for iterated algorithms. Data produced during one iteration is used in the next one. By performing several iterations for a sub-problem, the previous iteration values will remain in the cache when performing the next iteration.

The cookbook way of performing blocking is to focus on reuse of a target structure, and work outwards through loop levels until reaching a loop level that causes all elements of the target structure to be accessed for each iteration. Inner loops to this outer loop level are candidates to be broken up. Consider the following trivial example:

```
for (int iter = 0; iter < ITER; iter++) // "outer" loop
  for (int j = 0; j < SIZE; j++)
    func(a[j]);
```

Each iteration of the `iter` loop causes each element of `a` to be accessed. Hence we declare the `iter` loop to be the *outer loop*. Loop levels interior to this loop, namely `for j` in this case, are candidates to be blocked. Upon blocking, one new loop level is placed outside the *outer loop* and the original loop is kept in its the original location, with the restriction that it works on only a subset of the data for each iteration of the new outermost loop:

```
for (int jj = 0; jj < SIZE; jj += BLOCK)
  for (int iter = 0; iter < ITER; iter++) // "outer" loop
    for (int j = jj; j < jj + BLOCK; j++)
      func(a[j]);
```

Blocking will cause calculations to occur in a different order than in the original code, and sometimes this is fine. Many numerical algorithms can be reformulated to maintain accumulators to hold partial results between processing of different blocks. Some algorithms, however, will not lend themselves to blocking that easily.

One famous example is blocking a matrix multiplication. The unoptimized code looks like:

```
for (int i = 0; i < SIZE; i++)
  for (int j = 0; j < SIZE; j++)
    c[i][j] = 0;

for (int i = 0; i < SIZE; i++) // "outer" loop
  for (int j = 0; j < SIZE; j++)
    for (int k = 0; k < SIZE; k++)
      c[i][j] += a[i][k] * b[k][j];
```

The array `b` is traversed in the wrong way, and we should try to block with respect to it. Note that the `for i` loop level is the outer loop which causes all `b` elements to be revisited. `for j` and `for k` are therefore candidate loops to block. This can be done in different ways, for instance only splitting the `k` loop:

```
for (int i = 0; i < SIZE; i++)
  for (int j = 0; j < SIZE; j++)
    c[i][j] = 0;

for (int kk = 0; kk < SIZE; kk += BLOCK)
  for (int i = 0; i < SIZE; i++) // "outer" loop
    for (int j = 0; j < SIZE; j++)
      for (int k = kk; k < kk + BLOCK && k < SIZE; k++)
        c[i][j] += a[i][k] * b[k][j];
```

Or you could even split all the loops, possibly with different blocking factors:

```
for (int i = 0; i < SIZE; i++)
  for (int j = 0; j < SIZE; j++)
    c[i][j] = 0;

for (int ii = 0; ii < SIZE; ii += BLOCK_I)
  for (int kk = 0; kk < SIZE; kk += BLOCK_K)
    for (int jj = 0; jj < SIZE; jj += BLOCK_J)
      for (int i = ii; i < ii + BLOCK_I && i < SIZE; i++)
        for (int k = kk; k < kk + BLOCK_K && k < SIZE; k++)
          for (int j = jj; j < jj + BLOCK_J && j < SIZE; j++)
            c[i][j] += a[i][k] * b[k][j];
```

When blocking, special care needs to be taken to look for data dependencies. If one iteration depends on values produced in an earlier iteration, it follows that one must give special treatment to the boundary zones of the blocks to preserve the application semantics.

The blocking factor should be chosen to make the active problem size fit inside the target cache size. Different loop levels can be blocked to fit different cache levels.

5.2.3.2. Loop Fusion

When two loops use the same data it is beneficial to reduce the amount of data accessed between the loops. Cache lines fetched into the cache by the first loop may then still be in the cache when the second loop is run, so that it doesn't have to fetch them. The less data touched between the two loops, the better. Ultimately, if it is possible to completely merge the loop bodies, then all potential misses from the second loop will disappear.

It is not always possible to move the operations between loops. Special care must be taken not to move any data accesses past a write accesses to the same address, since that will change the meaning of the program.

ThreadSpotter™ will identify such accesses that may preclude loop fusion. However, there are cases where it may fail, for example, when a dependence is carried in a processor register and never stored to memory, or when ThreadSpotter™'s sparse sampling of information missed some dependency.

You therefore must always verify that the loops really are fusible, even if ThreadSpotter™ suggests fusion.

Depending on the data dependencies at hand, there are several variations of how loop fusion can be achieved:

- Moving instructions down from the first loop into the second loop.
- Moving instructions up from the second loop into the first loop.
- Moving instructions down from the second loop into the next iteration and into the first loop
- Moving instructions up from the first loop into the previous iteration of the second loop.
- Perform a partial loop merge and only move the instructions that do not have data dependencies, in any of the directions listed above.

Note that since there are four different directions to move the instructions, the code motion barriers are evaluated independently for each potential motion direction.

Consider this example:

```
double vector[SIZE];
double vector2[SIZE];
int i;

for (i = 0; i < SIZE; i++)
  vector[i] = i;

f(vector2);

for (i = 0; i < SIZE; i++)
  vector[i] *= vector2[i];
```

These two loops both use all elements in `vector`. The first loop always misses in the cache, assuming that `vector` has not been accessed before.

If `vector` is smaller than the available cache, then it might still be in the cache when the second loop starts, but that depends on how much data is accessed in the function `f`. One thing to consider is whether the call to `f` can be moved out of the way. We can't tell from looking at this snippet whether it would be possible, since the function body is not revealed.

Assuming that `f` only reads the elements in `vector2` it would be possible to move `f` before the first loop, and thus increase the likelihood that data remains in the cache between the loops.

```
double vector[SIZE];
double vector2[SIZE];
int i;

f(vector2);

for (i = 0; i < SIZE; i++)
  vector[i] = i;

for (i = 0; i < SIZE; i++)
  vector[i] *= vector2[i];
```

If `vector` is larger than the available cache the second loop will still suffer from cache misses when accessing `vector`, even though `f` has been moved away from between the loops. We should therefore try to merge the two loops.

```
double vector[SIZE];
double vector2[SIZE];
int i;

f(vector2);

for (i = 0; i < SIZE; i++) {
  vector[i] = i;
  vector[i] *= vector2[i];
}
```

Now ThreadSpotter™ may indicate another fusion possibility between a potential loop inside `f` and the one remaining loop. In that case the programmer needs to decide whether it is worth duplicating the contents of `f` to merge these loops, sacrificing legibility for performance.

5.3. Non-Temporal Data

Some algorithms have data uses where we know that the accessed data cannot be reused before it gets evicted from the cache. Such data is said to be *non-temporal*. It may, for example, be an algorithm that does transformations on data streams, reading the data once from one location and writing it once to another location, so that there are no data reuses at all.

It can also be an algorithm that is run on a data set that does not fit in the cache, and where the reuse of data cannot be improved using blocking or any of the other methods described in Section 5.2.3, “Unexploited Data Reuse Opportunities”.

Or, in the case of an algorithm where we have managed to improve the reuse of data using, for example, blocking, we may know that the data in each block will still be evicted between iterations of the algorithm.

In these cases we know that the data will be evicted from the cache before it is reused, and that trying to cache the data is pointless. The processor, on the other hand, does not know this and will try to cache the non-temporal data like any other data. The non-temporal data will occupy space in the cache, and may thereby cause unrelated data that could otherwise be successfully cached to be evicted.

5.3.1. Example of Non-Temporal Data Optimization

As a simple example, assume that we have a program that uses two arrays. The first array is 2 MB large, and the second array is 8 MB large. The program first iterates through the 2 MB array from beginning to end, then iterates through the 8 MB array from beginning to end, and then repeats this a number of times.

Now assume that this program is run on a processor with a 3 MB cache. When it starts from the beginning of the 2 MB array it has just iterated through the 8 MB array, so the 2 MB array will have been evicted from the cache and it will have to fetch each cache line in the array from memory. When it then starts from the beginning of the 8 MB array, it will have touched the 2 MB array and rest of the 8 MB array since it last touched each cache line, so each cache line will have been evicted and have to be fetched from memory.

We get a cache line fetch for each cache line each time we go through each of the arrays. However, we know that the larger array is not going to fit in the cache anyway. If we could tell the processor not to try to cache it at all, the smaller array would actually fit in the cache. Instead of getting cache line fetches in both the small and large arrays, we could at least get cache hits in the small array.

In this situation like this we would like to be able to tell the processor not to try to cache the larger of the arrays, and most modern processors actually implement instructions that allow us to do that. These instructions are said to have non-temporal hints, and allow you to tell the processor what data is non-temporal and should not be cached.

5.3.2. Singlethreaded Uses of Non-Temporal Hints

The most common use of non-temporal hints in singlethreaded programs is to avoid caching a data structure that we know will not fit in the cache in order to avoid evicting another data structure that does fit in the cache, as in the example above.

However, it is also possible to use non-temporal hints to reduce the number of cache line fetches if we have single data structure that is too big to fit in the cache. For example, assume that we have a program that repeatedly iterates over a single 8 MB array, and that we know it will run on a processor with a 6 MB cache using LRU replacement.

If we do not use non-temporal hints, this program will get a cache miss for each cache line it accesses in every iteration. The cache will contain the most recently accessed 6 MB of data, but when we access a cache line we will always have accessed 8 MB of data since we last accessed it.

It seems unnecessary that we should get a 100% miss ratio when our data set is quite close to fitting in the cache. To improve on this we can tell the processor not to try to cache the last 2 MB of the array using non-temporal hints. This way the first 6 MB of the array will fit into the cache, and we will now only get cache misses for the last 2 MB. We get a 25% miss ratio instead of 100%.

In reality you should probably not try use the entire cache like this, since there it will most likely be other small data structure and code that need some space. For example, using 5 MB of the cache and leaving 1 MB for other things may better in this case.

5.3.3. Multithreaded Uses of Non-Temporal Hints

The types of non-temporal optimizations possible in singlethreaded programs are of course also possible in multithreaded programs, but there are also other optimization opportunities if the threads share some level of the cache hierarchy.

If one thread with a small data set that fits into the cache and another thread with a large data set that does not fit in the cache share a cache, the thread with the large data set may cause the data of the other thread to be evicted. Both threads then get cache misses and lose performance.

We know that the thread with the larger data set will get cache misses anyway, so by adding a non-temporal hint its accesses we can prevent its data from being cached and the data of the other thread from being evicted. Instead of both threads missing in the cache, only the thread with the large data set that would miss anyway now misses.

5.3.4. Concurrent Uses of Non-Temporal Hints

If you know that a program is going to be run concurrently with other programs, or multiple instances of the program are going to be run concurrently, it is possible to make similar optimizations to those in multithreaded programs. By adding non-temporal hints to accesses that would miss in the cache anyway, you can reduce the cache pressure and the number of cache misses in other programs.

The difference is that in the case of multithreaded programs, ThreadSpotter™ calculates how much of the cache each thread uses in the analysis and reflects this in the report. This is not possible with multiple programs as each program is analyzed separately. It is therefore useful to specify a smaller cache size than the actual cache when doing the analysis, to simulate the effect of sharing the cache with the other programs.

If you will be running multiple instances of the same program, it might make sense to divide the cache size by the number of instances sharing the cache. For example, with four threads sharing a 6 MB cache, you could specify a 1.5 MB cache when doing the analysis.

If you will be running several different programs concurrently, it might make sense to specify an even smaller cache size than that, to take into account that different programs may claim different amounts of the cache. For example, with four programs sharing a 6 MB cache, you could specify a 1 MB cache when doing the analysis.

5.3.5. Types of Non-Temporal Hint Instructions

Modern x86 processors implement two types of instructions with non-temporal hints; non-temporal prefetches and non-temporal stores.

Non-temporal prefetches are easy to use. Simply doing a non-temporal prefetch of a cache line indicates to the processor that the data is non-temporal and should not be allowed to evict other data from the cache.

Non-temporal stores can offer further performance benefits compared to non-temporal prefetches in some situations. They are, however, much more complicated to use, and can easily cause severe performance degradations, or even bugs in multithreaded programs, if misused.

5.3.5.1. Non-Temporal Prefetches

The `prefetchnta` instruction is a prefetch with non-temporal hint. In addition to fetching the cache line into the cache like a regular prefetch, it also tells the processor that the data in the cache line is non-temporal and should not be allowed to evict other data from the cache.

A useful way to think about non-temporal prefetches is that they fetch the cache line straight into the first-level cache and mark it as non-temporal, and that when the cache line marked as non-temporal is evicted from the first-level cache it is not added to the higher-level caches. This means that non-temporal prefetches cannot be used to keep non-temporal data out of the first-level cache, but only higher-level caches.

This is not the way that all processors actually implement non-temporal prefetches. Some may not prefetch the cache line straight into the first-level cache, or they may use a small part of the higher-level caches for non-temporal data, but model above works well for these processors too.

This means that a program can prefetch a cache line with a `prefetchnta` instruction and use it for a short period while it is in the first-level cache. Once it is evicted it is then completely evicted from the cache hierarchy.

As with regular prefetch instructions, it is enough to do one non-temporal prefetch of each cache line. Doing multiple prefetches of the same cache line may cause a small performance penalty.

Note that using non-temporal prefetches incorrectly may increase the number of cache line fetches and decrease performance. Doing non-temporal prefetches of cache lines that would otherwise not have been evicted from the cache will force fetches of those cache lines. You should therefore only add non-temporal prefetches where the instructions later reusing the data have a very high fetch ratio. Measure the performance before and after adding a non-temporal prefetch to verify that it is effective.

5.3.5.2. Non-Temporal Stores

Non-temporal stores have an additional benefit over non-temporal prefetches. A cache line written using non-temporal stores will not be added to the caches, just as if it had been accessed by a non-temporal prefetch. But a non-temporal store also hints to the processor that the program intends to write the entire cache line, completely replacing the current content, so that the cache line does not need to first be fetched from memory.

Since the processor now only has to write the finished cache line to memory, and not fetch it first, this essentially reduces the memory bandwidth used by half.

The following non-temporal store instructions are available on x86 processors:

- For general-purpose registers the `movnti` instruction can be used.
- For MMX registers the `movntq`, `maskmovq` and `maskmovdqu` instructions can be used.
- For XMM registers the `movntdq`, `movntpd` and `movntps` instructions can be used.

The processor collects the data written to a cache line using non-temporal stores in special non-temporal store buffers. Once an entire cache line has been written, the processor writes it straight back to memory without first fetching it into or adding it to the caches.

Processors have a very small number of these non-temporal store buffers, typically 4-6 cache lines. If a program has more than this number of partially written cache lines in flight, the processor has to start writing partially written cache lines back memory. This is a very slow operation and may lead to severe performance degradations. Programs should therefore only keep a very small number of partially written cache lines in flight, ideally only one or two cache lines.

A typical use for non-temporal stores is copying memory regions that are too large to fit in the cache. Using ordinary stores for that would waste memory bandwidth by unnecessarily fetching all the data in the destination region into the cache before overwriting it. Any useful data that is in the cache before the copy would also be replaced with data from the source and destination regions.

Such a routine can instead use non-temporal prefetches on the source region and write to the destination region using non-temporal stores. Using non-temporal stores to write to the destination region saves memory bandwidth by overwriting the destination area without first fetching it into the cache, and using non-temporal accesses for both the source and destination region preserves any useful data that was already in the cache before the copy.

Another typical use is initialization of data structures too large to fit in the cache, for example, setting a large array to all zeros.

A major drawback of non-temporal stores is that they are fairly complex to work with. If they are improperly used they can easily cause performance degradations, or even hard-to-debug bugs in the case of multithreaded programs.

Here is a list of potential problems to keep in mind:

- The program should write entire cache lines at a time using non-temporal stores. Writing partial cache lines may lead to severe performance degradations.
- The program should not mix non-temporal stores and regular stores to the same cache line. Doing so may lead to severe performance degradations.
- The program should not read from a cache line while it is being written using non-temporal stores. Doing so may lead to severe performance degradations.
- The program should only write each part of the cache line once. Writing the same part of the cache line multiple times may lead to severe performance degradations.
- The program should only keep a very small number of partially written cache lines in flight. Keeping too many cache lines in flight may lead to severe performance degradations.
- Some of the non-temporal store instructions require the destination address to be 16-byte aligned. Using such instructions for unaligned stores may cause the program to crash.
- Non-temporal stores use a weaker memory consistency model than regular stores. This means that fencing operations must be used in conjunction with non-temporal stores to ensure correct operation in multithreaded programs. See the processor manual for more information.

Note

If you are unsure what this means, avoid using non-temporal stores in multithreaded programs to avoid hard-to-debug bugs.

5.3.6. Using Non-Temporal Hint Instructions

5.3.6.1. Adding Non-Temporal Hints to the Code

No major programming language currently supports the concept of non-temporal data, so instructions with non-temporal hints have to be added to a program manually. There are two ways to do this; compiler intrinsic functions or inline assembly.

A compiler intrinsic function is built-in function provided by the compiler, that is substituted for the desired instruction in the generated machine code. The name and arguments of these functions vary between compilers, so you have to consult the documentation of your compiler for specifics.

If your compiler does not provide an intrinsic function for the instruction you want to use, or if you for some reason want to avoid the intrinsic function, you can insert the instruction using inline assembly instead. Again, the syntax of inline assembly statements varies between compilers, so you have to consult the documentation of your compiler for specifics.

5.3.6.2. Processor Compatibility

Another thing to consider when using instructions with non-temporal hints is processor compatibility. The instructions with non-temporal hints were added in the SSE and SSE2 instruction set extensions, so not all processors support them.

Using instructions with non-temporal hints in 64-bit code is risk free, as all 64-bit processors implement all the instructions.

As for 32-bit code, AMD processors from the Athlon 64 and Opteron and newer, and Intel processors from the Pentium 4 and newer, support all non-temporal hint instructions in that mode.

If you want your code to run on older 32-bit processors, you either have to avoid these instructions, or check if the processor supports the instructions at run-time and implement separate routines for processors with and without them.

5.4. Multithreading Problems

Programs with multiple threads can be affected by an entirely new class of performance problems, related to how the data accesses of the threads interact with each other.

5.4.1. False Sharing

Section 3.7, “Multithreading and Cache Coherence” describes how cache coherence affects caching when multiple threads accesses the same cache line. When multiple threads access same cache line and at least one of them writes to it, it causes costly invalidation misses and upgrades. When the threads actually communicate by accessing the same data, this is a necessary overhead.

However, it may also be the case that the threads do not actually communicate. They may be accessing unrelated data that just happen to be allocated in the same cache line. In that case the costly invalidation misses and upgrades are completely unnecessary. By splitting the data accessed by the different threads to different cache lines, the invalidation misses and upgrades can be completely avoided.

5.4.1.1. False Sharing Example

Consider this simple example in C:

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

The functions `thread1` and `thread2` sum the values in the arrays they get as arguments to the variables `sum1` and `sum2`. Since `sum1` and `sum2` are defined next to each other, the compiler is likely to allocate them next to each other in memory, in the same cache line.

If the functions are run concurrently by two different threads, the execution may then look something like this:

1. First, `thread1` reads `sum1` into its cache. Since the line is not present in any other cache `thread1` gets it in exclusive state:

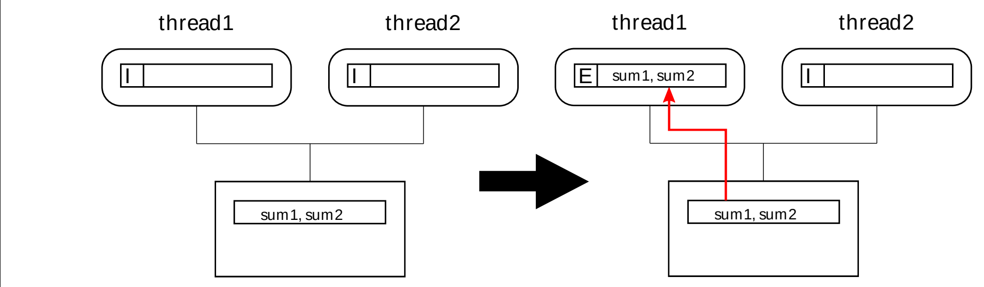


Figure 5.12. False Sharing Example, Step 1

2. `thread2` now reads `sum2`. Since `thread1` already had the cache line in exclusive state, this causes a downgrade of the line in `thread1`'s cache and the line is now in shared state in both caches:

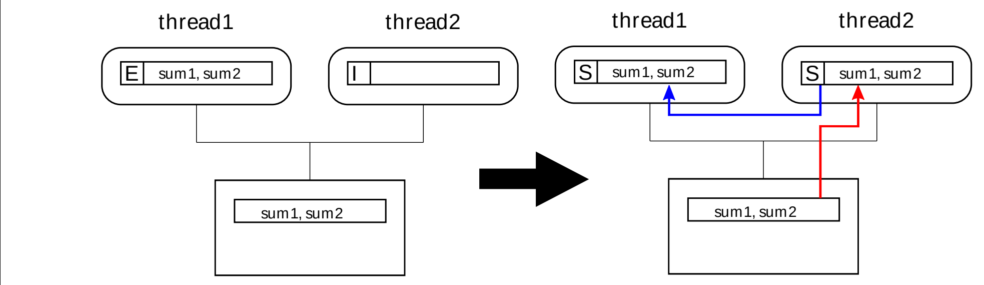


Figure 5.13. False Sharing Example, Step 2

3. `thread1` now writes its updated sum to `sum1`. Since it only has the line in shared state, it must upgrade the line and invalidate the line in `thread2`'s cache:

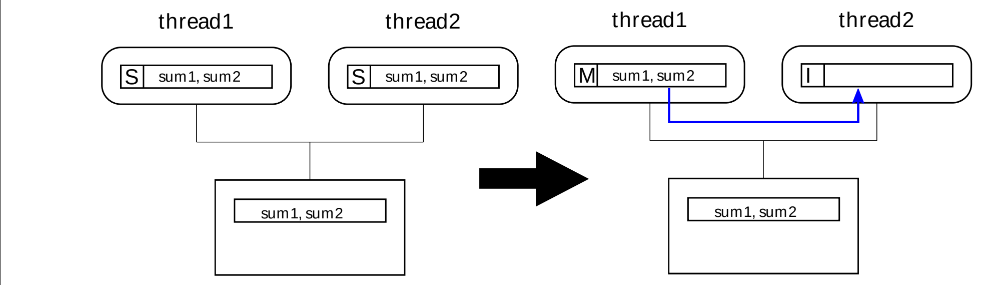


Figure 5.14. False Sharing Example, Step 3

4. `thread2` now writes its updated sum to `sum2`. Since `thread1` has invalidate the cache line in it's cache it gets a coherence miss, and must invalidate the line in `thread1`'s cache forcing `thread1` to do a coherence write-back:

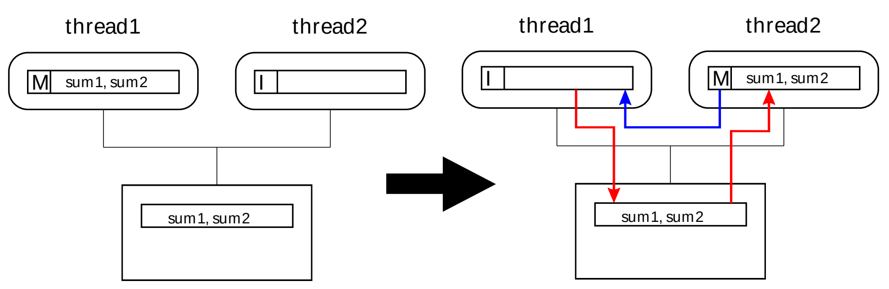


Figure 5.15. False Sharing Example, Step 4

5. The next iteration of the loops now starts, and `thread1` again reads `sum1`. Since `thread2` just invalidated the cache line in `thread1`'s cache, it gets a coherence miss. It must also downgrade the line in `thread2`'s cache, forcing `thread2` to do a coherence write-back:

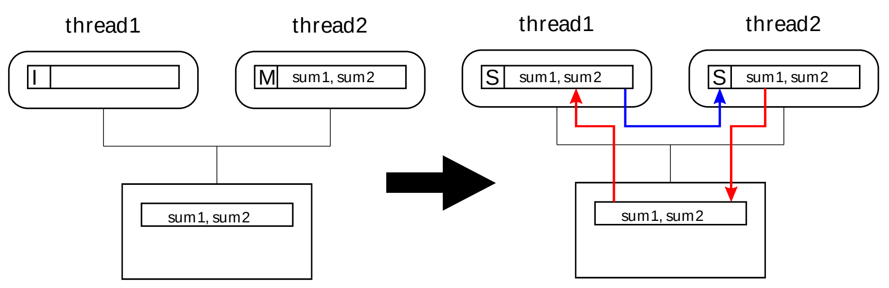


Figure 5.16. False Sharing Example, Step 5

6. `thread2` finally reads `sum2`. Since it has the cache line in shared state, it can read it without and coherence activity, and we are back in the same situation as after step 2:

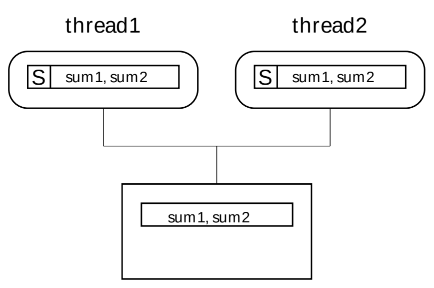


Figure 5.17. False Sharing Example, Step 6

For each iteration or the loops, steps 3 to 6 will repeat, each time with costly upgrades, coherence misses and coherence write-backs.

In reality, the memory accesses would probably not interleave exactly like this, but the same updates, coherence misses and coherence write-backs would still occur. In a simple example like this the compiler may also allocate `sum1` and `sum2` to registers, avoiding the memory accesses causing the false sharing, but in a more complex program they compiler may not be able to do this.

To fix a false sharing problem you need to make sure that the data accessed by the different threads is allocated to different cache lines.

In our simple example we could do this by assuring that both variables are aligned to the start of a cache line. Unfortunately, the C language does not have a standard mechanism to specify the alignment of data. However, most C compilers have some extension to do this. For example, assuming that our processors have 64-byte cache lines and that we are using the GCC compiler, our fixed program would look like this:

```
int __attribute__((aligned(64))) sum1;
int __attribute__((aligned(64))) sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

Since the variables are now allocated in different cache lines, `thread1` and `thread2` can keep a copy of the cache line they need and execute without upgrades, coherence misses or coherence write-backs:

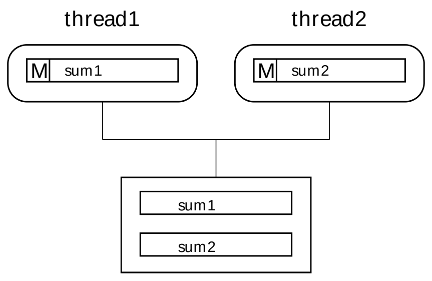


Figure 5.18. False Sharing Example, Fixed

5.4.1.2. Causes of False Sharing

There are a number of common causes of false sharing:

- Arrays of per-thread data.

  A typical programming pattern that causes false sharing is creating an array with one element per thread, such as an array of per-thread counters. Consider this example:

```
int sums[NUM_THREADS];

void threaded_sum(int thread_num, int v[], int v_count) {
    sum[thread_num] = 0;
    for (int i = 0; i < v_count; i++)
        sum[thread_num] += v[i];
}
```

This is a function that would be called by multiple threads, each summing the elements of some array into a per-thread sum in an array indexed by thread number. This creates a situation similar to the example with the `sum1` and `sum2` variables above. Since the per-thread data is allocated in array it will allocated together in memory, most likely in the same cache line, causing false sharing.

Avoid creating such per-thread arrays for frequently accessed data. Use function local variables for frequently accessed data, or assure that the data of each thread is allocated to a separate cache line in some other way, for example, using cache line alignment or padding.

- Access patterns in parallelized matrix operations.

  Another common cause of false sharing is parallelizations of algorithms that work on matrices or multi-dimensional arrays.

  If such parallelizations make a fine grained division of the matrix between the threads, it can cause threads to write to elements adjacent to elements being accessed by other threads. Adjacent elements are likely allocated in the same cache line, making the writes likely to cause false sharing. For example, assume the elements marked in blue and yellow in this example are written by two different threads:

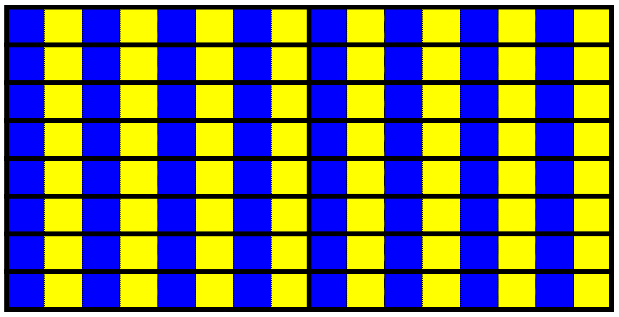


Figure 5.19. Matrix Accesses with False Sharing

In this case each write by the two threads will invalidate the cache line in the cache of the other thread, causing lots of coherence misses.

A more coarse grained division of the matrix between the threads will allow the threads to work on different cache lines to a greater degree, avoiding false sharing:

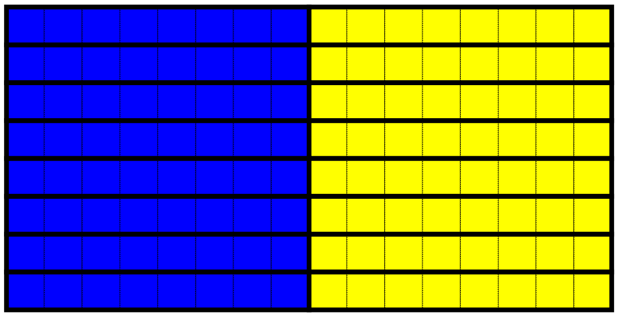


Figure 5.20. Matrix Accesses without False Sharing

If you get problems with false sharing running parallel matrix operations, it is therefore often a good idea to make division of the matrix between the threads more coarse grained.

- Accesses to different fields in a structure from different threads.

  Accesses to different fields in a structure from different threads can also cause false sharing problems. Consider this example in C:

```
struct ab {
    int a;
    int b;
    struct ab *next;
};

struct ab *abs;

void inc_a(void) {
    struct ab *this_ab = abs;
    while (this_ab != NULL) {
        this_ab->a++;
        this_ab = this_ab->next;
    }
}

int sum_b(void) {
    int sum = 0;
    struct ab *this_ab = abs;
    while (this_ab != NULL) {
        sum += this_ab->b;
        this_ab = this_ab->next;
    }
    return sum;
}
```

  The function `inc_a` increments the value of the `a` field of each element in the `abs` linked list, while the function `sum_b` sums the values of the `b` field of each element in the list.

If the functions are run concurrently or alternatingly by different threads, the writes to the `a` field of each element in the list by the `inc_a` function will invalidate the list elements in the cache of the thread running the `sum_b` function. Similarly, the read of the `b` field of each element in the list by the `sum_b` function will downgrade the corresponding cache line in the cache of the thread running the `inc_a` function.

In a case like this it is probably best to split the structure into two structures containing the data accessed by each thread. If memory consumption is not a concern, you could also add enough padding between the fields to make sure they end up in different cache lines.

- Statically allocated variables.

  The example at the beginning of this section about false sharing showed how the statically allocated variables in this program can cause false sharing:

```
int sum1;
int sum2;

void thread1(int v[], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

  In cases like this, if the variables do not need to be statically allocated and are not excessively large you can make them function local variables instead. This causes them to be allocated on the threads' stacks, where they will most likely be safe from false sharing.

  If it is not possible to make the variables thread local, you have to change their allocation in memory. One way to make sure that two variables are not allocated to the same cache line is to specify that both should be aligned to the start of a cache line.

  Another way is be collect all variables accessed by a thread in a structure, and then add enough padding to the end of the structure to make sure no other variables can be allocated to the same cache line. This is a way to avoid false sharing if the compiler does not allow you to specify the alignment of data.

- Dynamically memory allocation.

  If your program uses dynamic memory allocation and allocates small data objects, data objects used by different threads may be allocated in the same cache line, causing false sharing.

  Trying to allocate larger chunks of memory for use by one thread is probably the best fix in this case. For example, instead of allocating data objects for use by one thread individually you can allocate an array of data objects at once. This may reduce the interleaving of data objects used by different threads and thereby also false sharing.

Otherwise, aligning the data objects causing false sharing to the beginning of a cache lines is again a way to avoid false sharing. Depending on the operating system you are using there may be memory allocation functions you can use to do this.

5.4.2. Poor Communication Utilization

Threads in a multithreaded programs often communicate by reading and writing shared data in memory. Section 3.7, “Multithreading and Cache Coherence” describes how this causes costly upgrades, coherence misses and coherence write-backs. But, if the threads truly communicate unlike false sharing described in Section 5.4.1, “False Sharing”, this is to some degree an unavoidable cost.

However, there is a performance cost for each cache line transferred from the cache of one thread to another, so you want to minimize the number.

One way to measure the efficiency of the communication is to calculate communication utilization, the fraction of each transferred cache line that was actually written by the producer and read by the consumer, as described in Section 4.7, “Communication Utilization”.

The communication utilization is similar to fetch utilization and write-back utilization, but measures the efficiency of the communication between caches instead of between the cache and memory. If the communication utilization is low it means we are moving unused data between the processors. By using the communicated cache lines more efficiently it may be possible to reduce the communication overhead.

The causes of poor communication utilization are largely the same as the causes of poor fetch utilization and poor write-back utilization:

- Unused fields in structures.

  Unused fields in a structure being written by one thread and read by another may waste space in the communicated cache lines. See Section 5.1.1, “Partially Used Structures” for more information.

- Alignment problems.

  Data alignment problems may cause unused padding in or between communicated data structures. See Section 5.1.3, “Alignment Problems” for more information.

- Dynamic memory allocation.

  If the communicated data objects have been dynamically allocated, they may be spread out in memory and be interleaved with the dynamic memory allocator's bookkeeping data. See Section 5.1.4, “Dynamic Memory Allocation” for more information.

- Irregular access patterns.

  Arranging the communicated data in a way that causes irregular access patterns may cause poor communication utilization. See Section 5.2.2, “Random Access Pattern”.

- Too fine grained communication.

  One cause of poor communication utilization that does not correspond to a fetch or write-back utilization problem is too fine grained communication. If the communication between two threads is too fine grained, the communicated data may not fill entire cache lines and unused data is transferred between the caches.

  For example, assume that we have a producer thread and a consumer thread communicating 8-byte data objects through a queue based on a circular buffer, and that they run on a computer with 64-byte cache lines.

If the producer writes one data object to the queue at a time and then signals the consumer to read it, a cache line with only 8 bytes of useful data out of the 64 bytes will be sent from the producer to the consumer. This will cause one upgrade in the producer thread's cache and one coherence miss in the consumer thread's cache for each communicated object.

However, if the producer thread knows that there are more data objects waiting to be sent, it can delay signaling the consumer thread until it has written a whole cache line of eight data objects to the queue. This way the entire communicated cache line is used, and you only get one upgrade and one coherence miss for every eight data objects.

Making the communication more coarse grained is also likely to reduce the amount of synchronization between the threads, and thereby reduce the synchronization overhead.

5.5. Common Data Structures

Certain data structures such as linked lists, trees and hash tables typically have quite bad cache behavior. There are two reasons for this. Firstly, individual elements are often allocated dynamically, which means housekeeping data for the memory allocator is placed between the elements. This causes poor cache line utilization.

Secondly, using these data structures often causes random access patterns, see Section 5.2.2, “Random Access Pattern”, caused either by dynamic memory allocation or the way the data structures are traversed.

When using data structures from a library, for example, from the STL library in C++, one thing to consider is that they have been designed to work reasonably well in all cases, especially with very large numbers of elements. This means that for structures with a small numbers of elements they often add unnecessary overhead. Implementing your own data structure adapted specifically to your application may provide significant performance gains.

Generally, a structure that is never, or very rarely, changed can be more efficiently implemented than a structure that needs to support efficient updating. Supporting efficient changes often requires pointer indirections between elements and dynamic memory allocation of individual elements. This can often be avoided in a static representation.

Sometimes a structure is used in two phases, first being filled in with data and then only being used for lookups. It may then pay off to first use a representation that supports efficient updating, and then convert it to a more efficient read-only data structure once it has been filled in.

5.5.1. Arrays

From a cache performance point of view a linear search in an array is a quite ideal workload. The linear search has a good spatial locality, and the regular access pattern means that the hardware prefetcher can effectively prefetch the accessed data.

Elements in an array are also efficiently packed. Only a single memory allocation is done for all the elements and the elements are stored contiguously in memory. In comparison a linked list, for example, often uses more memory for different kinds of overhead than for actually storing the data.

The drawback of a linear search is of course that the number of accesses that need to be made grows linearly with the number of elements, so it does not work well for large numbers of elements. However, the low overhead usually makes the performance superior to more complex structures when there are only a few elements, and depending on cache pressure and other factors it can still be competitive up to a few tens of elements.

If all, or the vast majority, of the accesses to a structure with a large number of elements are lookups, using a binary search in a sorted array may be more efficient than using a search tree structure. For lookups the binary search offers the same logarithmic time complexity as a search tree. However, the elements in an array are much more efficiently packed than a tree with dynamically allocated nodes, and it will therefore achieve much better cache performance.

5.5.2. Linked Lists

Linked lists are commonly plagued by two problems, they cause a high degree of memory overhead and they cause random access patterns.

A linked list typically allocates its elements dynamically, which causes some memory management overhead for each element. Each element also has to contain a pointer to the next element and possibly to the previous element. On a 64-bit processor this can easily add up to 16 - 32 bytes of overhead per element. This means that, for example, in a linked list of pointers typically only a third or less of the memory is actually used for the stored data, the rest is overhead.

Dynamic memory allocation also means that the list elements may be spread out in memory. Even if the list elements can only be accessed sequentially, the access pattern when the list is traversed may therefore still be irregular.

The memory layout depends on the memory allocator, as well as the algorithms that assemble and manipulate the list. If a linked list is causing performance problems, look over the node allocation.

If the list is never updated it should be replaced with an array. This may also be true if the list is only rarely updated or only contains a few elements. If the list is changed now and then, but a part of the application reads the linked list repeatedly between the changes, it may even pay off to copy the list to a temporary array to use in that part of the application.

Another way to hide latencies in linked lists is to add a pointer to a node several steps further ahead to the elements, and when traversing the linked list use this pointer to prefetch elements further ahead:

```
struct node {
  struct node *next;
  struct node *prefetch;
  int data;
}

/* Traverse a list and populate prefetch hints to point
 * distance steps ahead.
 */
void prepare_prefetch_hint(struct node *head, int distance)
{
    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 = p;
}

void traverse(struct node *head) {
  struct node *p = head;
  for (p = head; p != NULL; p = p->next) {
    prefetch(p->prefetch);
    access(p->data);
  }
}

/* Call function once after updating list */
prepare_prefetch_hint(head, 8);

/* Assuming many traversals between each update */
traverse(head);
```

Since prefetch instructions have no side effects the prefetch pointers do not need to be kept completely accurate at all times, It may be enough to only update them now and then if the list is frequently changed.

The tricky thing is to determine how many elements ahead to prefetch. This is a function of memory latencies, cache size and how much processing is done for each node. ThreadSpotter™ will help you determine the right distance.

5.5.3. Trees

Trees, like linked lists, cause a lot of memory overhead from dynamic memory allocation and pointers between nodes, and the nodes may be spread out in memory because of dynamic memory allocation.

Tree operations also have an inherently random access pattern, since element lookups and changes will follow different paths through the tree in an irregular fashion. Rebalancing operations also keep rearranging the tree layout as the tree is being modified.

Iterating through the elements in the sorting order of the keys is very inefficient compared to iterating over a sorted array, so if that is done repeatedly between changes in the tree it may pay off to copy the tree contents to a temporary array and iterate over it instead.

If the tree contents are static, it may be worth while to spend some time to arrange the nodes so that adjacent nodes also reside close to each other in memory. Replacing the tree lookups with a binary searches in a sorted array will provide the same logarithmic time complexity, but with less memory overhead and therefore better cache line utilization and performance.

It may also be possible to collapse several layers of a tree, or a cluster of graph nodes into a more densely allocated sub-structure.

5.5.4. Hash Tables

Hash tables often suffer from random access patterns and poor cache line utilization. One reason for the random access patterns can be the hash function itself. Hash functions are often to designed to map keys to more or less random indexes in the hash table to avoid collisions. However, this means that lookups will also access randomly distributed indexes. If you know that some keys are likely to be looked up in sequence, try to map those keys to adjacent indexes.

For example, if the hash key is an integer and you know that lookups are likely to be done on sequential keys, just using the key modulo the hash table size will work well. The sequential keys will then map to sequential locations in the hash table.

A random access pattern is often enough to cause poor cache line utilization by itself, but another factor in hash tables is the fill ratio. Hash tables are often sized to be larger than the number of elements to reduce the number of collisions. This leaves unused indexes in the table, lowering the cache line utilization.

Decreasing the size of the table will increase the number of collisions, but may pay off in increased cache line utilization and reduced cache misses.

In hash tables using collision lists, these lists may cause problems as described in Section 5.5.2, “Linked Lists”. Replacing them with arrays may improve performance.

5.6. Final Remedies

Very complex code may not lend itself to transforming, or the code may actually be optimal but still suffer from cache misses. Remaining misses can then be reduced by adding software prefetch operations.

Once software prefetch instructions are added to the program, ThreadSpotter™ will evaluate their efficiency.

To be efficient, the prefetch instruction needs to fulfill three conditions:

- It has to do real work. Data which is addressed by the prefetch instruction must not already be in the cache.
- It must appear at a sufficiently large distance from the subsequent instruction using the prefetched data. This is memory and architecture specific. A rough approximation is about 40 memory operations prior to the subsequent instruction if fetching the data from main memory, which translates to about 100-150 instructions.
- The data must not be evicted before it is subsequently used. The prefetch instruction must therefore not be too far away from the instruction that makes use of the data.

The GNU compiler suite has built-in support for software prefetch instructions (__builtin_prefetch). You then also need to tell the compiler to generate code for a processor model that supports prefetch instructions.

Other compilers may have their own intrinsic functions, or the programmer may need to resort to writing inline assembly code.

Cache misses occurring close in time can often be overlapped on x86 architectures (so-called memory-level parallelism, MLP). An alternative to prefetching is therefore to try to move cache-miss accesses closer together (and hoping that the compiler will keep it that way...).

Chapter 6. Optimization Workflow

The process to optimize an application for good cache performance involves distinct phases, each targeting a specific category of problems. The order of the phases is somewhat important, as some problems obscure others, and certain transformations will enable other approaches.

This chapter outlines one way to approach this rather complex situation from a memory hierarchy standpoint. There are numerous other aspects of improving application performance than is listed here, ranging from the macro scale of properly establishing an efficient architecture and a matching development process, selecting the optimal algorithms, managing time and space complexity of different storage methods, database schema optimization, minimizing database and communication overheads, arranging for parallelization, judicious inlining and denormalization, and all the way down to CPU pipeline granularity tuning.

You may need to perform optimization in any one of these areas.

6.1. Initial State: Correct, Measurable Program, Good Test Case

The first step is to make sure the program is running correctly, and that it produces some measurable output that can be used to verify the correctness of later versions of the program.

As far as possible, try to devise a test case with predictable, repeatable behavior, which is independent of execution speed (since the sampler will impose a slowdown).

It will be difficult for a programmer to ensure unmodified behavior, and to determine whether he has achieved any performance improvements if these conditions are not met. Real time programs, in particular, may start experiencing altered behavior if the processor load is increased. There is no universal solution for this, but some real time applications can be changed to reduce their time-base corresponding to the apparent slowdown.

Some applications run from start to completion and their execution is easily repeatable. These are the simplest applications to analyze. Others run more or less continuously, and are rated on number of transactions per second, average throughput or something similar. Establish a criteria for starting and stopping acquisition of fingerprint data, which can be timed, measured, and above all, repeated.

Some source code transforms change the number of memory operations. To be able to compare cache miss behavior of a run before the change with a run of the changed code, there are a few options. It is possible to compare miss and fetch ratios, but if an algorithm change alters the number of accesses as well as the number of misses, the calculated miss ratio will change in an unpredictable way. Under repeatable circumstances, it makes more sense to compare the absolute number of misses before and after the algorithm change.

If the test case is not absolutely repeatable, it will be difficult to correlate two measurements of the absolute number of misses or fetches. In such cases it makes more sense to look at ratio numbers, as they are less affected by exact execution times.

Tip

Ensure correct execution.

Tip

Devise a repeatable test case.

Tip
Take a reference measurement of the the execution time.

6.2. Avoid Unnecessary Memory Accesses

Generally speaking, huge performance improvements come from algorithm changes.

Use an ordinary execution profiler (such as gprof or callgrind) to find hot-spots in the code. That will estimate the amount of time spent in each part and direct your attention to highly used code sections. Evaluate issues like:

- Are the right compiler flags used? Is the best compiler used? Some compilers are better than others on optimizing the code.
- Is this the best algorithm to perform the job?
- Is the program doing unnecessary work, such as unnecessarily repeating an expensive operation?
- Can data that is expensive to calculate be cached? For instance, use "dirty" flags to avoid scanning large structures for changes.
- Can loops be unrolled?
- Can vector instructions be used?
- Do loops contain conditional branching? Sometimes these parts can be moved out of the loop.
- Is the compiler forced to be conservative about register allocation (due to potential aliasing)? Consider helping the compiler to optimize by introducing temporary variables.
- Does the program contain unnecessary copying of structures? This often occurs with programming paradigms that advocate layering, encapsulation and data hiding. Consider relaxing encapsulation rules where it offers performance improvements.

Tip
Use optimizing compilers and turn on suitable optimization flags.

Tip
Avoid unnecessary work by caching data

Tip
Use the correct algorithms

Tip
Consider favoring algorithms that touch data sequentially

Tip
Help the compiler by rearranging loops and introducing temporary variables

Tip
Judiciously break encapsulation to avoid copying data

6.3. Optimize Data Layout

After making sure the program is algorithmically optimal, we turn to memory related optimization areas. The first such area is to ensure that data is laid out in memory in an optimal way.

Generally speaking, data needs to be packed to utilize memory well. It also needs to be sorted in such a way as to utilize spatial reuse. With spatial reuse, we mean that data being used in the same code context should be placed next to each other.

The following situations are but some that can cause wasted space in cache lines:

- Internal alignment gaps. The compiler needs to obey rules that tell which addresses certain data types can reside on. This can cause subsequent variables or fields in a record to be allocated with unused data in between. Sorting fields by their alignment requirements will minimize this waste.
- External alignment gaps. Alignment problems can also occur between records in an array, due to alignment requirements of the first field in a record. Make sure the record has a size which is a multiple of the largest sub-field size.
- Using too wide data types. Applicable both to bit fields, numeric types and over-sized partially used fixed size arrays.
- Dynamical memory allocation adds housekeeping structures, which are used much less often than data in the allocated region.
- Fields sorted in an suboptimal way. Not all fields being used in every situation.
- Incorrect padding causes records to be split between cache lines in an adverse way.
- Dynamic memory allocation allocates subsequent blocks non-contiguously.
- Certain data structures tend to destroy locality when placing data. (Hash tables, trees).
- Indirections instead of directly storing data
- Linked lists trade-offs: dynamic memory allocation spreads memory accesses, extra space required for pointers.

Tip
Use correct data types

Tip
Sort record fields according to size and usage pattern.

Tip
Organize data to avoid mixing read-only and write fields in the same cache line. If many fields needs to be updated at the same time, place them in the same cache line.

Tip
Avoid dynamic allocation of small objects, and consider using custom allocators

Tip
Use data structures that pack data efficiently

Tip

Avoid "pointer chasing" and indirections

6.4. Optimize Access Patterns

A program with poor access patterns exhibit some of the same traits as a program with inefficient memory layout, and it is not always obvious what the fundamental problem is. For instance, is using only one field from a record a memory layout problem or is the problem related to the access pattern?

By relating inefficient access patterns specifically to late spatial reuse, we can narrow the definition somewhat.

By addressing memory layout problems before access patterns, the analysis will produce more accurate estimates on the performance potential for fixing access pattern.

Generally, inefficient access patterns arise from:

- Traversing a data set along an incorrect dimension, also known as inefficient loop nesting.
- Random, or "pseudo-random" accesses to a data set. This will happen if the data set is not sorted in memory as the same order as the general order of accesses occur in. Certain high level data structures will also cause accesses to spread wildly. Indirections by an index or by following pointer chains will also spread memory accesses in a seemingly random way.

Tip

Organize data or change data access order to utilize all data in a cache line

Tip

Avoid patterns or data structures that are random or non-deterministic in nature, or cause random access patterns.

Tip

Consider building up local copies of data, where the local copy is transformed, filtered and sorted in an optimal way for the current algorithm.

Tip

For data structures involving following pointers, such as linked lists, trees or hash table collision lists, consider collapsing several tree levels or ranges of nodes into densely packed vectors. This lowers the space overhead, while improving locality. The draw-back is more complex traversal logic.

6.5. Utilize Reuse Opportunities

Basically there are two variants of unused reuse opportunities that the ThreadSpotter™ tools detect: reuse within a loop, and reuse between different loops. The former is addressed by applying a technique called blocking or tiling, and the latter is handled by bringing the two loops closer together or actually merging the loop bodies completely.

Reuse come in two distinct flavors: spatial reuse, where the benefit comes from that subsequent data is already fetched into the cache, and temporal reuse, where the very same data is revisited multiple times. Minimizing the time before the revisit will lower miss ratios.

Addressing these issues will usually cause loop hierarchies to be partially turned around and broken up. Loop fusion will also necessarily change the features of the program. This changed scene should be re-analyzed for unnecessary memory accesses, poor layout and poor access patterns.

Tip

Bring related loops closer together, and optimally merge related parts of their loop bodies.

Tip

Look for spatial reuse, and change loop nesting or block with respect to the data structure that displays long reuse distance.

Tip

Look for temporal reuse, and change algorithms to perform as many iterations as possible for a given data chunk before moving along.

Tip

After merging loop bodies or transforming loop structures, rerun analysis to find advice for the new program structure.

6.6. Use Non-Temporal Hints for Data without Temporal Reuse

Sometimes it is simply not possible to increase the temporal reuse of an algorithm. Most modern processors have instructions for handling non-temporal data that can be used to optimize cache usage in such cases, see Section 5.3, “Non-Temporal Data”.

Use non-temporal prefetches to hint to the processor what cache lines you know will be evicted from the cache before they are reused. This frees up cache space, which may allow other data that was previously evicted to be successfully cached.

When writing continuous regions of non-temporal data, use non-temporal store instructions instead of non-temporal prefetches to avoid fetching the overwritten data from memory.

Tip

Having too many active non-temporal store streams will result in partially filled store buffers being written back to memory. This severely impacts the performance of the application.

Tip

Consider blocking algorithms that would otherwise have too many parallel non-temporal store streams in flight.

Tip

Iteratively apply non-temporal prefetches to data structures with large reuse distances. Start adding prefetches to accesses to those data structures that contribute a lot to the total number of fetches.

6.7. Avoid False Sharing

In the presence of multiple threads that share data, there are a number of sharing effects that may affect performance. One such sharing pattern is *false sharing*. It arises if at least two threads are both using unrelated data placed close enough to end up in the same cache line. False sharing occurs when they repeatedly update their respective data in such a way that the cache line migrates back and forth between the two threads' caches.

Often this can be avoided by giving explicit alignment pragmas to the compiler.

In OpenMP programs False sharing arises when several threads maintain their respective partial result in a vector indexed by the thread rank. Replacing this with thread local variables often helps.

Tip
Avoid writing to global data that is accessed from multiple threads.

Tip
Align shared global data to cache line boundaries.

Tip
Don't store temporary, thread specific data in an array indexed by the thread id or rank.

Tip
When parallelizing an algorithm, partition data sets *along* cache lines, not across cache lines.

6.8. Avoid Communication between Caches (Coherence Traffic)

Analogous to optimizing data layout for efficient cache usage and bandwidth usage, for multithreaded applications it is important to arrange data for efficient inter-cache communication.

Whenever one thread uses data that a different thread has written, there is some communication occurring between the caches. This involves synchronizing the caches' contents, and maintaining a notion of the current owner. Just like the memory communication, synchronization and ownership is managed for cache-line chunks.

If the consuming thread is not using every byte in the communicated cache line, then this is wasteful. It would be better to reorganize data to fill cache lines fully before letting the consumer start reading data.

Tip
Add complete cache lines' worth of data to shared memory buffers before letting the consumer thread start reading data

Tip
When performing matrix calculations, align the calculation frontier along cache lines instead of across cache lines.

6.9. Hide Remaining Misses

After all algorithms and data structures are optimal, we can sometimes further improve program behavior by carefully adding prefetch instructions.

Many times it is difficult to determine the correct place to insert the prefetch instruction and what address to prefetch. Sometimes it helps to augment the data structure that is being traversed with a field containing a hint of the proper address to prefetch, for instance the address of a node several steps away along the linked list.

Tip

Look for remaining hot-spots and look for good places to insert prefetch instructions. Evaluate the effectiveness of added prefetch instructions.

Tip

Consider augmenting "tricky" data structures with a prefetch hint field.

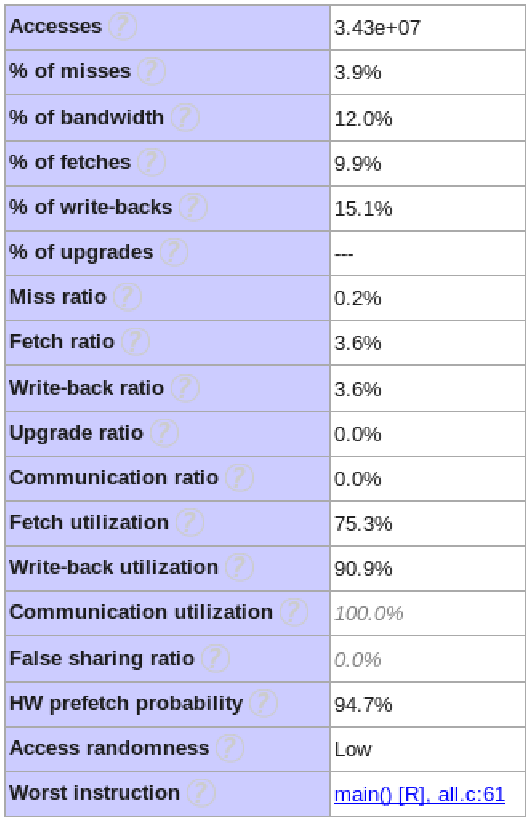
Chapter 7. Reading the Report

The output from ThreadSpotter™ is an report file, typically having file extension `.tsr`. The report file is presented in a web browser through a special tool called **view**. Invoking this command on a report file will bring up the default web browser and present the report.

The default filename for the report is `report.tsr`, although that can be overridden when the report is generated. The report file contains everything that is needed to present the report, so it can be conveniently moved, renamed or shared, for instance through e-mail.

7.1. Statistics

In the report you will find statistics sections in lots of places. There is a summary statistics section for the entire application, and there are smaller statistics sections for issues, loops and instruction groups.



Accesses	3.43e+07
% of misses	3.9%
% of bandwidth	12.0%
% of fetches	9.9%
% of write-backs	15.1%
% of upgrades	---
Miss ratio	0.2%
Fetch ratio	3.6%
Write-back ratio	3.6%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	75.3%
Write-back utilization	90.9%
Communication utilization	*100.0%*
False sharing ratio	*0.0%*
HW prefetch probability	94.7%
Access randomness	Low
Worst instruction	main() [R], all.c:61

Figure 7.1. Issue Statistics Section

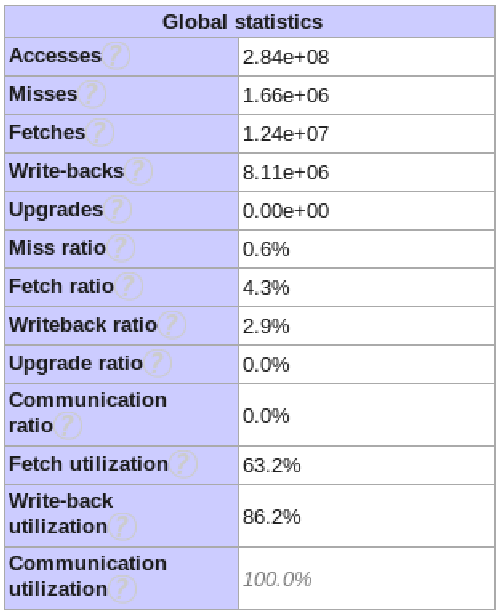
A statistics section consists of two parts, some statistics in numerical form and two diagrams. The diagrams plot some statistics for different cache sizes while the numerical statistics provide the exact values for the cache size the report focuses on.

The number of available fields in most statistics sections depends on how the input data was sampled. Absolute values, for example accesses and misses, are unavailable when analyzing burst sampled data. Also, note that the absolute values only apply to the sampled region when attach/detach is used.

ThreadSpotter™ is compatible with SlowSpotter™, however some statistics can not be generated when analyzing sample files created with SlowSpotter™. The most obvious difference is that all statistics related to communication will be disabled for such files. Other differences include, but are not limited to, fetch and write-back utilization and prefetch handling.

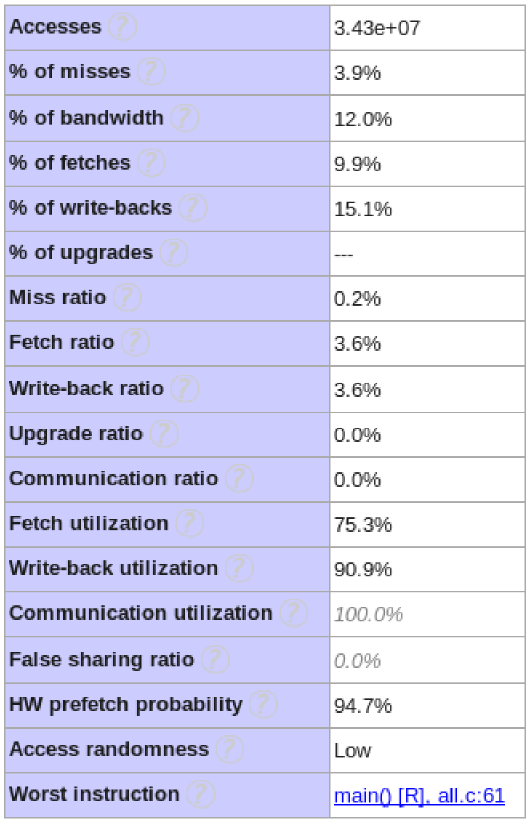
7.1.1. Reading the Statistics

The summary section, issues and loops all contain numerical statistics. The fields shown in statistics sections in different parts of the report differ somewhat, but there is a large overlap.



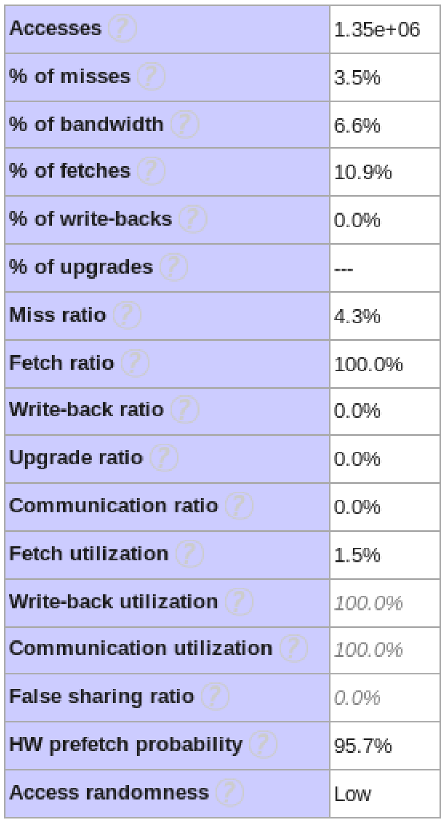
Global statistics	
Accesses	2.84e+08
Misses	1.66e+06
Fetches	1.24e+07
Write-backs	8.11e+06
Upgrades	0.00e+00
Miss ratio	0.6%
Fetch ratio	4.3%
Writeback ratio	2.9%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	63.2%
Write-back utilization	86.2%
Communication utilization	*100.0%*

Figure 7.2. Summary Statistics



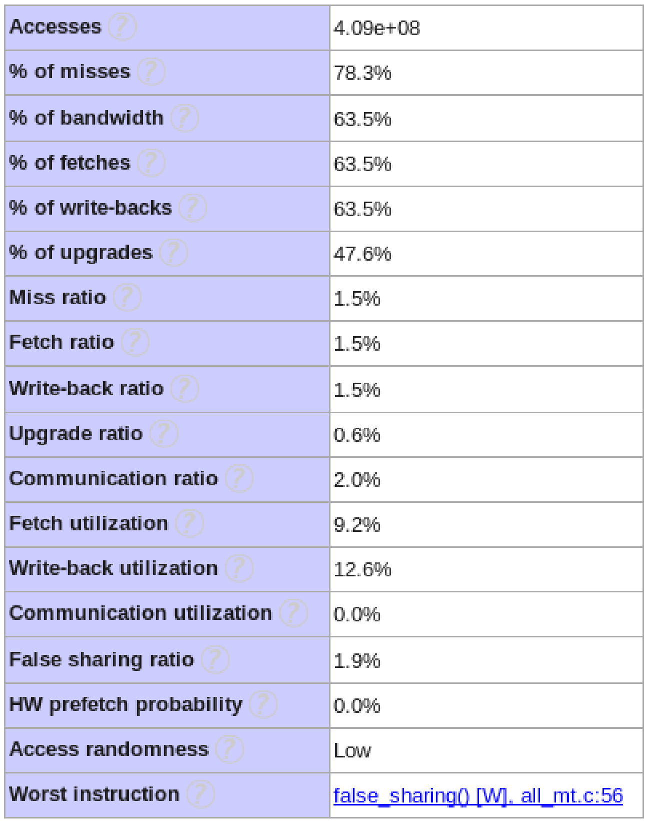
Accesses	3.43e+07
% of misses	3.9%
% of bandwidth	12.0%
% of fetches	9.9%
% of write-backs	15.1%
% of upgrades	---
Miss ratio	0.2%
Fetch ratio	3.6%
Write-back ratio	3.6%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	75.3%
Write-back utilization	90.9%
Communication utilization	*100.0%*
False sharing ratio	*0.0%*
HW prefetch probability	94.7%
Access randomness	Low
Worst instruction	main() [R], all.c:61

Figure 7.3. Issue Statistics



Accesses	1.35e+06
% of misses	3.5%
% of bandwidth	6.6%
% of fetches	10.9%
% of write-backs	0.0%
% of upgrades	---
Miss ratio	4.3%
Fetch ratio	100.0%
Write-back ratio	0.0%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	1.5%
Write-back utilization	*100.0%*
Communication utilization	*100.0%*
False sharing ratio	*0.0%*
HW prefetch probability	95.7%
Access randomness	Low

Figure 7.4. Loop Statistics



Accesses ?	4.09e+08
% of misses ?	78.3%
% of bandwidth ?	63.5%
% of fetches ?	63.5%
% of write-backs ?	63.5%
% of upgrades ?	47.6%
Miss ratio ?	1.5%
Fetch ratio ?	1.5%
Write-back ratio ?	1.5%
Upgrade ratio ?	0.6%
Communication ratio ?	2.0%
Fetch utilization ?	9.2%
Write-back utilization ?	12.6%
Communication utilization ?	0.0%
False sharing ratio ?	1.9%
HW prefetch probability ?	0.0%
Access randomness ?	Low
Worst instruction ?	false_sharing() [W], all_mt.c:56

Figure 7.5. Instruction Group Statistics

The value in some of the statistics fields may become gray and italic to indicate that it has a weak statistical base. This usually happens because the instruction group only has an insignificant amount of fetches.

Accesses	The total number of memory accesses performed by the entire application, or a specific part of the application when not shown in the summary view. This value is calculated for the duration of the sampling and only corresponds to the total number of accesses performed by the application if the application was sampled from start to end. This value is not available for burst sampled applications.
Misses	The total number of cache misses caused by the application during the sampling, see Section 3.4, “Cache Misses”. This value is not available for burst sampled applications.
Fetches	The total number of cache fetches caused by the application during the sampling, including those originating from hardware or software prefetches. See Section 3.8, “Fetch Ratio”. This value is not available for burst sampled applications.
Write-backs	The total number of write-backs caused by the application during sampling.

Upgrades	The total number of cache line upgrades caused by the application during the sampling. See Section 3.9, “Upgrade Ratio”. This value is not available for burst sampled applications.
Miss ratio	The cache miss ratio of the entire application, see Section 3.4, “Cache Misses”.
Fetch ratio	The cache line fetch ratio of the entire application when displayed in the summary view, or for the specific part of the program when displayed in an issue, loop or instruction group statistics section. Includes fetches originating from hardware or software prefetches. See Section 3.8, “Fetch Ratio”.
Write-back ratio	The likelihood that a write instruction causes a cache line to be written back to memory. See Section 3.10, “Write-Back Ratio”
Upgrade ratio	The upgrade ratio of the entire application when displayed in the summary view, or for the specific part of the program when displayed in an issue, loop or instruction group statistics section. See Section 3.9, “Upgrade Ratio”.
Communication Ratio	The fraction of memory accesses that cause communication between caches. See Section 3.7, “Multithreading and Cache Coherence”
Utilization	Fraction of a cache line that is touched (read or written) before the cache line is evicted. This value is shown instead of the separate fetch and write-back utilization values when analyzing sample files produced by old versions of SlowSpotter™.
Fetch utilization	The average fraction of each cache line fetched from memory or the next cache level that is actually read before the cache line is evicted from the cache. See Section 4.5, “Fetch Utilization”.
Write-back utilization	The average fraction of each cache line written back to memory or the next cache level that has actually been written by the time it gets written back. See Section 4.6, “Write-Back Utilization”
Communication utilization	The average fraction of each cache line communicated from one cache to another cache at the same level that is actually read in the receiving cache before it is evicted. See Section 4.7, “Communication Utilization”
Processor model	The cpu model that the report focuses on. The cpu model tells how many and how large caches there are, how many cores there are and how they share caches on various levels, how prefetch instructions work and how the non-temporal write instructions work.
Number of CPUs	The number of CPUs assumed for this analys.
Number of caches	The number of caches in the system on the selected cache level. Application threads are considered to populate this many caches. Note, this is not the total number of caches in the system.

Cache level	The cache level this report focuses on. This setting interacts with the cpu selection with respect to prefetch analysis, as depending on the cpu model, not all cache levels are affected by prefetch instructions.
Cache size	The cache size the report focuses on in bytes. This can be the actual size (default), or it can be overridden. See Section 3.2, “Cache Lines and Cache Size” [21].
Cache line size	The cache line size the report focuses on in bytes. This can be the actual size (default), or it can be overridden. See Section 3.2, “Cache Lines and Cache Size” [20].
Replacement policy	The cache replacement policy the report focuses on. See Section 3.3, “Replacement Policies”.
Software prefetches active	Indicates whether the effects of software prefetches are visible on this cache level.
% of misses	The fraction of the total number of cache misses of the application that are caused by the selected issue, loop or instruction group.
% of bandwidth	The fraction of the total bandwidth requirement of the application, that is caused by the selected issue, loop or instruction group.
% of fetches	The fraction of the total number of cache line fetches of the application, including those originating from hardware or software prefetches, that are caused by the selected issue, loop or instruction group.
% of write-backs	The fraction of the total number of write-backs of the application that are caused by the selected issue, loop or instruction group.
% of upgrades	The fraction of the total number of cache line upgrades of the application that are caused by the selected issue, loop or instruction group.
False sharing ratio	The likelihood that an access causes a cache line to be communicated between two caches without actually sharing any data between the two threads. This is related to the communication ratio, but only includes useless communication. See Section 5.4.1, “False Sharing” This value is not available when analyzing sample files produced by SlowSpotter™.
HW prefetch probability	An estimate of the fraction of the cache misses that are avoided by the hardware prefetcher, assuming that the memory bandwidth limit is not hit. See Section 3.6.2, “Hardware Prefetching”.
Access randomness	An estimate of randomness of the memory access pattern of this part of the application. Random access patterns are generally harmful to performance, see Section 4.11, “Access Randomness”.
Worst instruction	Points out the instruction that causes causes the largest number of cache line fetches in this part of the program, and the source code line that generated it.

7.1.2. Reading the Diagrams

The report contains diagrams describing several cache size dependent application characteristics. The summary tab in the summary frame shows application global values, while the individual issues and loops show values related to their respective instruction groups.

The diagrams plot their values for different cache sizes, from an 8 kilobyte cache to a 16 megabyte cache in the following example. The cache size that the report focuses on is marked with a vertical black line, in this case at 64 kilobytes.

7.1.2.1. Fetch/Miss Ratio Diagram

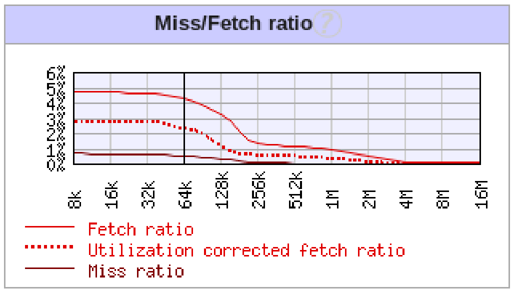


Figure 7.6. Fetch/Miss Ratio Diagram

- *Bright red line*

  Fetch ratio, the ratio of memory operations in the program, loop, issue or instruction group that, directly or indirectly through hardware prefetching, cause a data transfer between memory and cache. See Section 3.8, “Fetch Ratio”.

- *Red dotted line*

  Utilization corrected fetch ratio. Fetch ratio if the fetch utilization was raised to 100%. See Section 4.8, “Utilization Corrected Fetch Ratio”.

- *Dark red line*

  Miss ratio, the ratio of memory operations in the program that stall due to cache misses. The difference between the fetch ratio and the miss ratio is caused be software and hardware prefetching. See Section 3.4, “Cache Misses”.

7.1.2.2. Write-Back Ratio Diagram

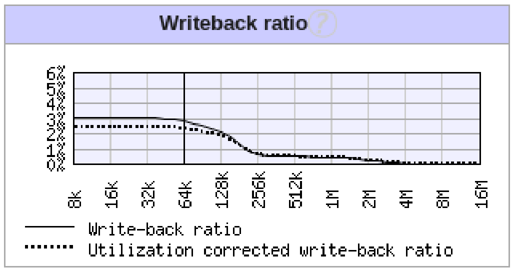


Figure 7.7. Write-Back Ratio Diagram

- *Black line*

  Write-back ratio, the ratio of memory accesses that cause a cache line to be written back to memory. See Section 3.10, “Write-Back Ratio”.

- *Black dotted line*

  Utilization corrected write-back ratio, the ratio of memory accesses that would cause a write-back if the write-back utilization was raised to 100%. See Section 4.9, “Utilization Corrected Write-Back Ratio”.

7.1.2.3. Utilization Diagram

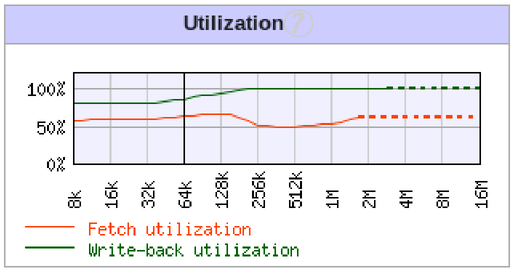


Figure 7.8. Utilization Diagram

- *Blue line*

  Cache line utilization for the program, loop, issue or instruction group. Shows how large fraction of the data that is loaded into the cache is actually used by read or write operations.

  This line is shown instead of a separate fetch and write-back utilization when analyzing files produced by old versions of SlowSpotter™.

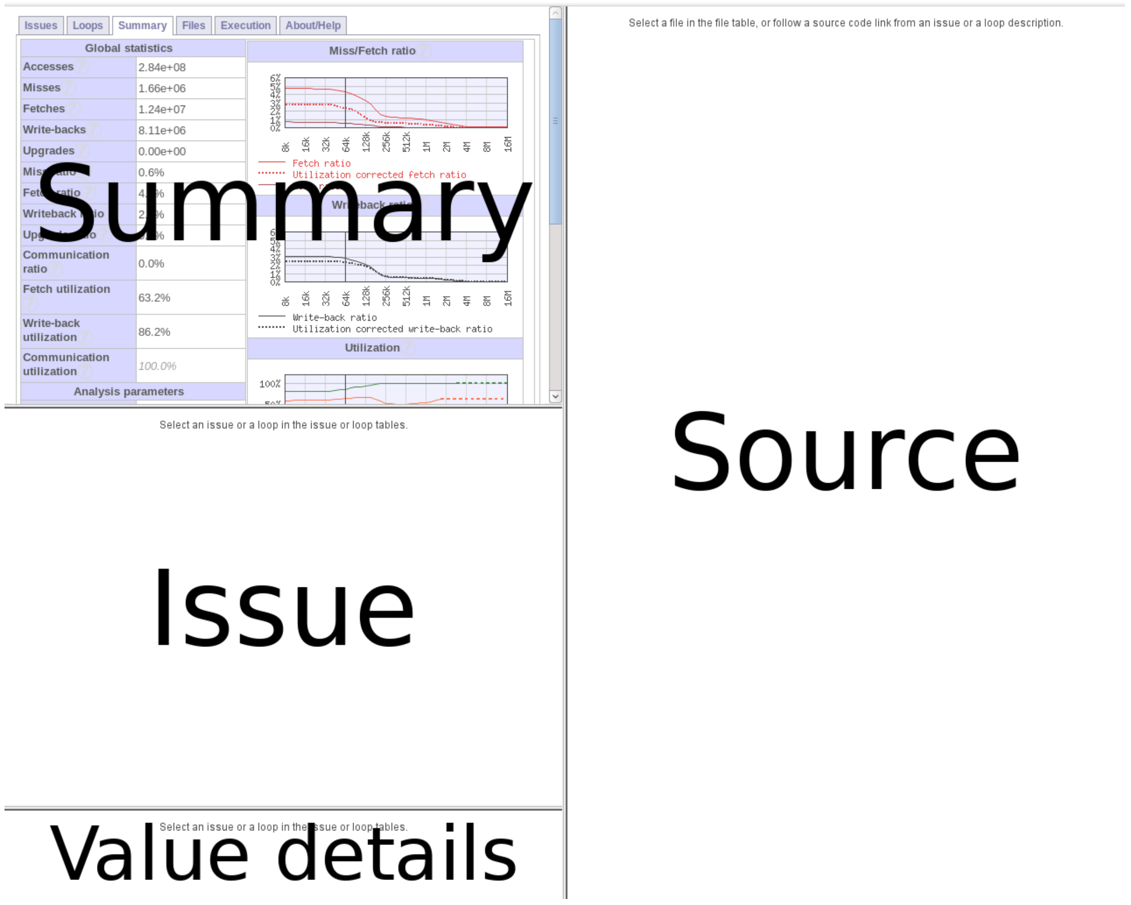
- *Orange line*

  Fetch utilization of the program, loop, issue or instruction group. Shows how large fraction of the data that is loaded into the cache is actually read. The utilization curve is dashed beyond the point where the estimates have a weak statistical base. See Section 4.5, “Fetch Utilization”.

- *Green line*

  Write-back utilization of the program, loop, issue or instruction group. Shows how large fraction of a cache line that is written prior to writing the line to memory. The utilization curve is dashed beyond the point where the estimates have a weak statistical base. See Section 4.6, “Write-Back Utilization”.

7.2. The Report Layout



Global statistics	
Accesses	2.84e+08
Misses	1.66e+06
Fetches	1.24e+07
Write-backs	8.11e+06
Upgrades	0.00e+00
Miss ratio	0.6%
Fetch ratio	4.[illegible]%
Writeback ratio	2.[illegible]%
Upgrade ratio	[illegible]%
Communication ratio	0.0%
Fetch utilization	63.2%
Write-back utilization	86.2%
Communication utilization	100.0%
Analysis parameters	

Miss/Fetch ratio

Fetch ratio
Utilization corrected fetch ratio

Writeback ratio

Write-back ratio
Utilization corrected write-back ratio

Utilization

Select a file in the file table, or follow a source code link from an issue or a loop description.

Select an issue or a loop in the issue or loop tables.

Select an issue or a loop in the issue or loop tables.

Figure 7.9. Report Outline

The report consists of three frames:

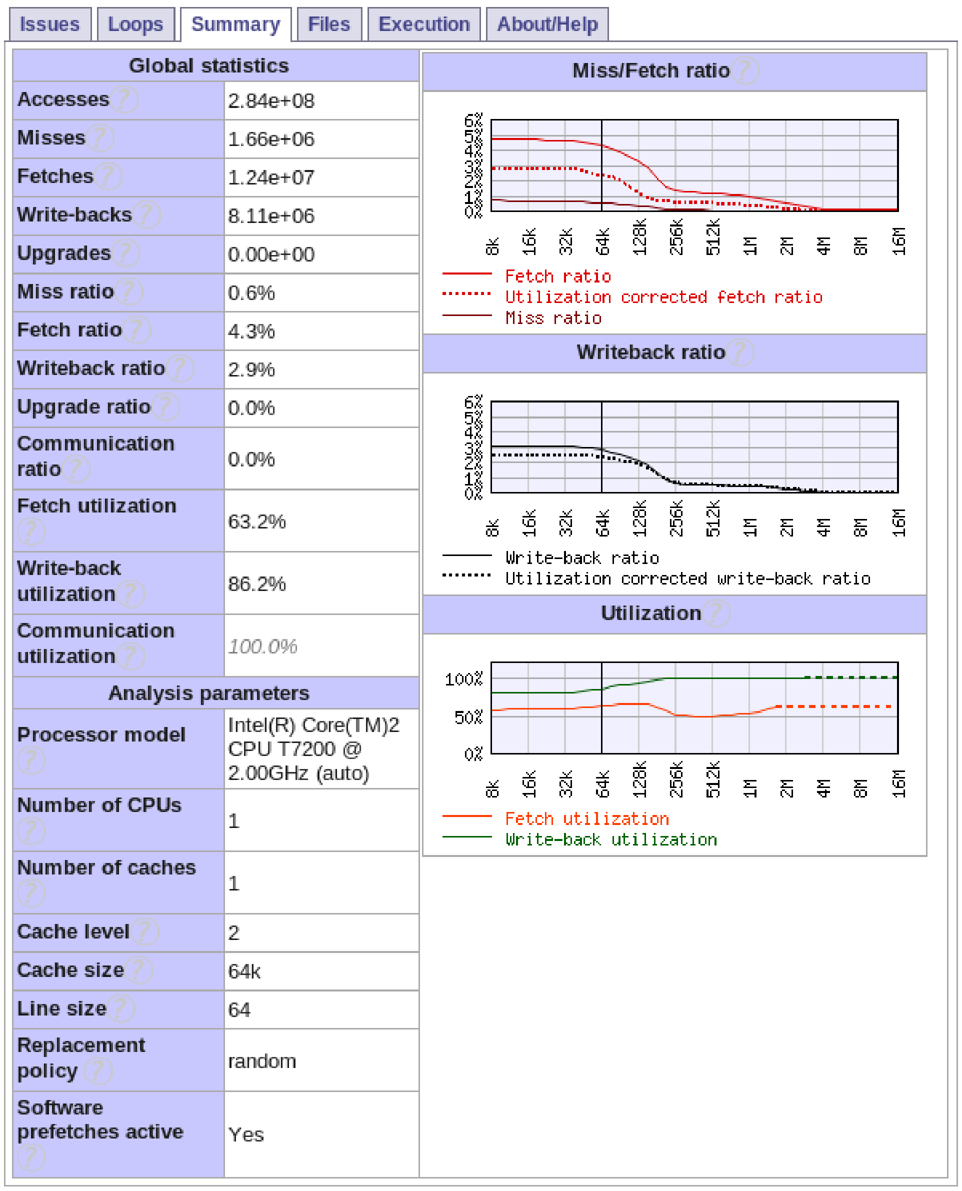
- The *summary* frame to the upper left shows summary results for the whole application. This frame can show lists of issues that were found by the analysis, the loops identified in the application that suffer from one or more issues and general information about the application and sampling. The summary frame also a provides navigation for the other two frames, which elaborate on a selected issue or loop.
- The *issue* frame to the lower left shows detailed information about a specific issue or loop.
- The *source* frame to the right shows the source code related to the currently viewed issue or loop.

7.3. The Summary Frame

The summary frame contains different views which can be chosen by clicking on their respective tab.

7.3.1. The Summary Tab

The summary tab presents statistics for the entire application and information about the parameters used in the analysis.



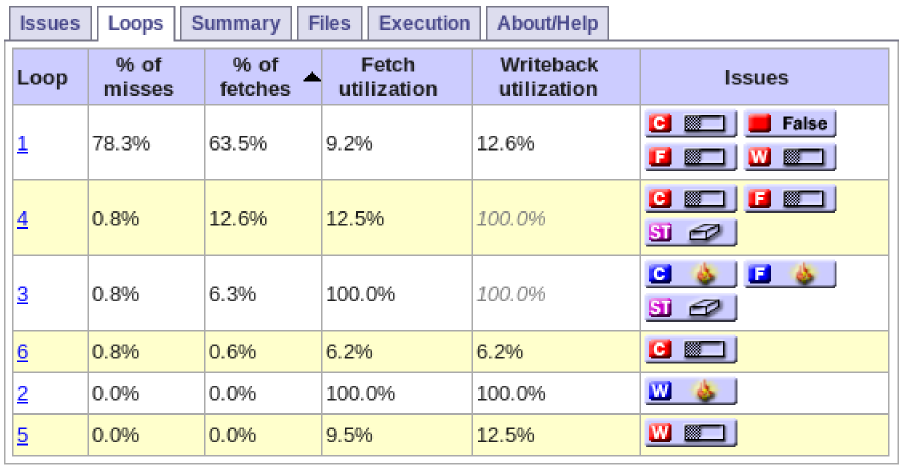
Global statistics	
Accesses	2.84e+08
Misses	1.66e+06
Fetches	1.24e+07
Write-backs	8.11e+06
Upgrades	0.00e+00
Miss ratio	0.6%
Fetch ratio	4.3%
Writeback ratio	2.9%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	63.2%
Write-back utilization	86.2%
Communication utilization	100.0%
Analysis parameters	
Processor model	Intel(R) Core(TM)2 CPU T7200 @ 2.00GHz (auto)
Number of CPUs	1
Number of caches	1
Cache level	2
Cache size	64k
Line size	64
Replacement policy	random
Software prefetches active	Yes

Figure 7.10. The Summary Tab

The diagrams show various cache size dependent metrics for the whole application. See Section 7.1.2, “Reading the Diagrams”.

The table below the diagram shows the values of some statistics for the entire application at the specific cache size the analysis focuses on and the values of some parameters used in the analysis, see Section 7.1.1, “Reading the Statistics”

7.3.2. The Loops Tab



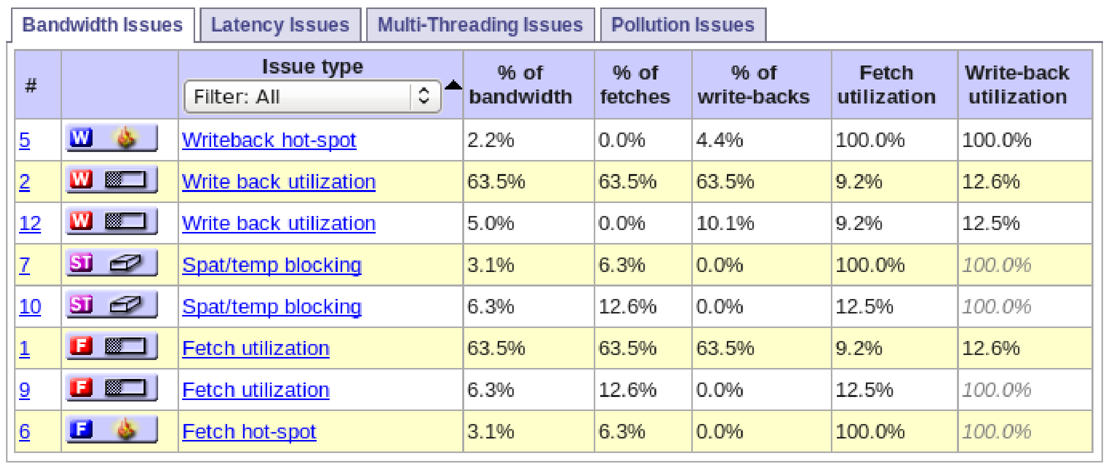
Loop	% of misses	% of fetches ▲	Fetch utilization	Writeback utilization	Issues
1	78.3%	63.5%	9.2%	12.6%	C ▭ 🟥 False F ▭ W ▭
4	0.8%	12.6%	12.5%	*100.0%*	C ▭ F ▭ ST ▱
3	0.8%	6.3%	100.0%	*100.0%*	C ● F ● ST ▱
6	0.8%	0.6%	6.2%	6.2%	C ▭
2	0.0%	0.0%	100.0%	100.0%	W ●
5	0.0%	0.0%	9.5%	12.5%	W ▭

Figure 7.11. The Loops Tab

The loops tab enumerates the loops associated with issues that have been found in the application and some statistics about them. Sometimes the same loop may be the cause of several issues and it may then be a good take an over all look at the loop instead of looking at each of the issues separately.

Loop	Numerical identifier for the current loop. Clicking on this number will bring up information about the loop in the issue frame, see Section 7.4.4, “Issue Details”. It will also highlight the instructions related to the loop in the source frame.
% of misses	The proportion of the total cache misses that are caused by the loop.
% of fetches	The proportion of the total cache fetches that are caused by the loop. Loops are sorted by this value by default.
Fetch utilization	Average fetch utilization for the instructions involved in this loop. See Section 4.5, “Fetch Utilization”.
Write-back utilization	Average write-back utilization for the instructions in this loop. See Section 4.6, “Write-Back Utilization”
Issues	Icons symbolizing the issues associated with this loop.

7.3.3. The Bandwidth Issues Tab



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

#		Issue type (Filter: All)	% of bandwidth	% of fetches	% of write-backs	Fetch utilization	Write-back utilization
5	W	Writeback hot-spot	2.2%	0.0%	4.4%	100.0%	100.0%
2	W	Write back utilization	63.5%	63.5%	63.5%	9.2%	12.6%
12	W	Write back utilization	5.0%	0.0%	10.1%	9.2%	12.5%
7	ST	Spat/temp blocking	3.1%	6.3%	0.0%	100.0%	100.0%
10	ST	Spat/temp blocking	6.3%	12.6%	0.0%	12.5%	100.0%
1	F	Fetch utilization	63.5%	63.5%	63.5%	9.2%	12.6%
9	F	Fetch utilization	6.3%	12.6%	0.0%	12.5%	100.0%
6	F	Fetch hot-spot	3.1%	6.3%	0.0%	100.0%	100.0%

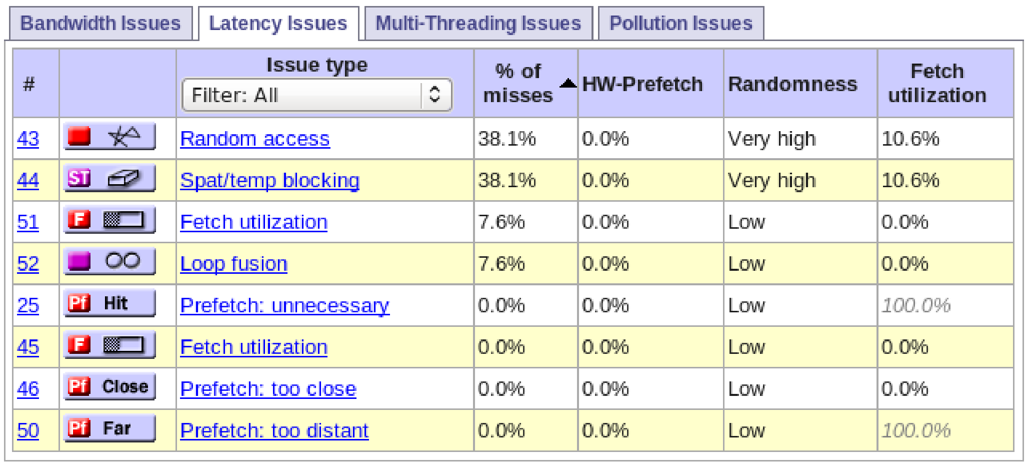
Figure 7.12. The Bandwidth Issues Tab

The bandwidth issues tab lists the issues that primarily affect the memory bandwidth requirement of the application. These are issues that may be hidden by hardware prefetching as long as there is enough memory bandwidth available, but will start to cause cache misses if the memory bandwidth is exhausted.

The following columns are available in the bandwidth issue list:

Issue number	Numerical identifier for the current issue. Clicking on this number will bring up information about the issue in the issue frame, see Section 7.4.4, “Issue Details”. It will also bring up the related source code in the source frame, and highlight the worst line.
Issue type	Issue type and icon. The issue icon is also shown in the source frame.
% of bandwidth	The proportion of the total fetches and write-backs that are caused by the issue. This gives you an idea of which issues cause the most memory-bandwidth usage and therefore are most important to address. Bandwidth issues are sorted by this value by default.
% of fetches	The percentage of the total cache line fetches that are related to the issue.
% of write-backs	The percentage of the total cache line write-backs that are related to the issue.
Fetch utilization	Average fetch utilization for the instructions involved in this issue. See Section 4.5, “Fetch Utilization”.
Write-back utilization	Average write-back utilization for the instructions involved in this issue. See Section 4.6, “Write-Back Utilization”

7.3.4. The Latency Issues Tab



#		Issue type Filter: All	% of misses	HW-Prefetch	Randomness	Fetch utilization
43		Random access	38.1%	0.0%	Very high	10.6%
44	ST	Spat/temp blocking	38.1%	0.0%	Very high	10.6%
51	F	Fetch utilization	7.6%	0.0%	Low	0.0%
52	OO	Loop fusion	7.6%	0.0%	Low	0.0%
25	Pf Hit	Prefetch: unnecessary	0.0%	0.0%	Low	100.0%
45	F	Fetch utilization	0.0%	0.0%	Low	0.0%
46	Pf Close	Prefetch: too close	0.0%	0.0%	Low	0.0%
50	Pf Far	Prefetch: too distant	0.0%	0.0%	Low	100.0%

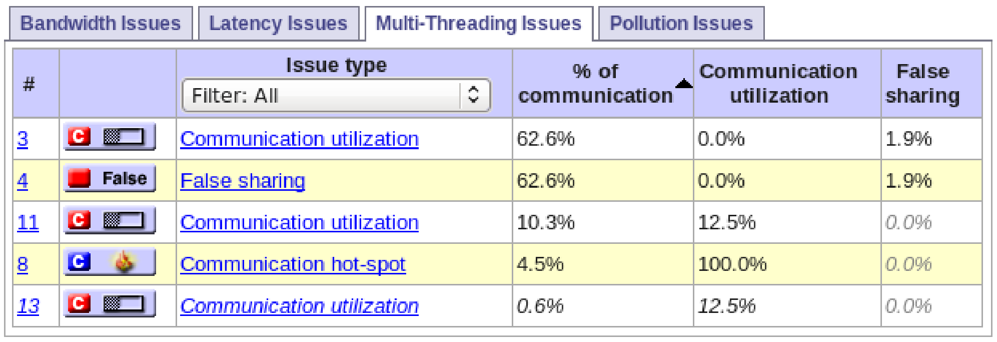
Figure 7.13. The Latency Issues Tab

The latency issues tab lists the issues that primarily cause cache misses. Cache misses cause stalls in the execution and have an immediate negative effect on the execution speed.

The following columns are available in the latency issue list:

Issue number	Numerical identifier for the current issue. Clicking on this number will bring up information about the issue in the issue frame, see Section 7.4.4, “Issue Details”. It will also bring up the related source code in the source frame, and highlight the worst line.
Issue type	Issue type and icon. The issue icon is also shown in the source frame.
% of misses	The proportion of the total cache misses that are caused by the issue. This gives you an idea of which issues cause the most misses and therefore are most important to address. Latency issues are sorted by this value by default.
HW-Prefetch	An estimate of the fraction of the cache misses related to this issue that are avoided by the hardware prefetcher, assuming that the memory bandwidth limit is not hit. See Section 3.6.2, “Hardware Prefetching”.
Randomness	An estimate of randomness of the memory access pattern for the instructions related to this issue. See Section 4.11, “Access Randomness”.
Fetch utilization	Average fetch utilization for the instructions involved in this issue. See Section 4.5, “Fetch Utilization”.

7.3.5. The Multi-Threading Issues Tab



#		Issue type Filter: All	% of communication	Communication utilization	False sharing
3	C	Communication utilization	62.6%	0.0%	1.9%
4	False	False sharing	62.6%	0.0%	1.9%
11	C	Communication utilization	10.3%	12.5%	0.0%
8	C	Communication hot-spot	4.5%	100.0%	0.0%
13	C	*Communication utilization*	*0.6%*	*12.5%*	*0.0%*

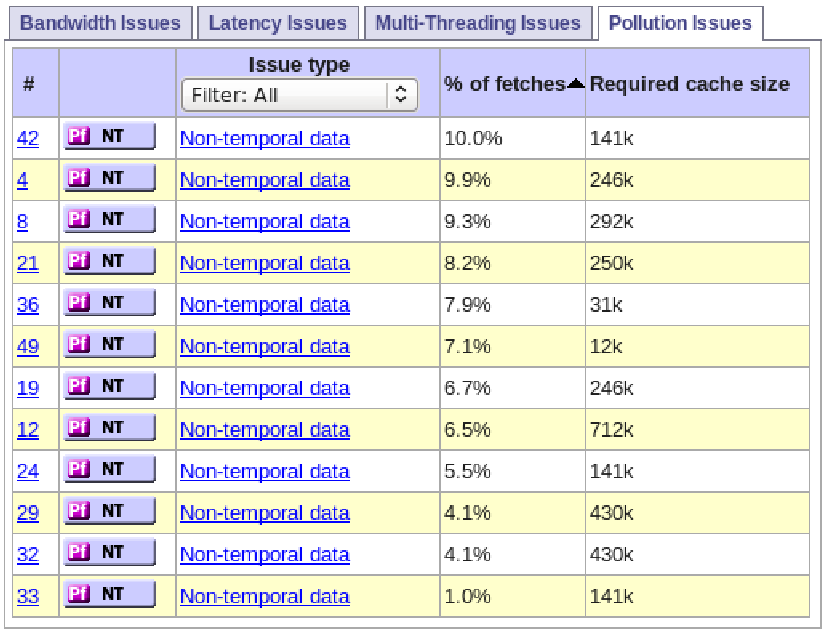
Figure 7.14. The Multi-Threading Issues Tab

The multi-threading issues tab lists issues related to multithreading and data sharing between threads.

The following columns are available in the multi-threading issue list:

Issue number	Numerical identifier for the current issue. Clicking on this number will bring up information about the issue in the issue frame, see Section 7.4.4, “Issue Details”. It will also bring up the source code for the related to the issue in the source frame.
Issue type	Issue type and icon. The issue icon is also shown in the source frame.
% of communication	The percentage of all cache line communication of the application that is associated with this issue.
Communication utilization	The average communication utilization for the instructions involved an issue. See Section 4.7, “Communication Utilization”.
False sharing	The probability that an access related to an issue will suffer from false sharing. See Section 5.4.1, “False Sharing”.

7.3.6. The Pollution Issues Tab



#		Issue type (Filter: All)	% of fetches	Required cache size
42	Pf NT	Non-temporal data	10.0%	141k
4	Pf NT	Non-temporal data	9.9%	246k
8	Pf NT	Non-temporal data	9.3%	292k
21	Pf NT	Non-temporal data	8.2%	250k
36	Pf NT	Non-temporal data	7.9%	31k
49	Pf NT	Non-temporal data	7.1%	12k
19	Pf NT	Non-temporal data	6.7%	246k
12	Pf NT	Non-temporal data	6.5%	712k
24	Pf NT	Non-temporal data	5.5%	141k
29	Pf NT	Non-temporal data	4.1%	430k
32	Pf NT	Non-temporal data	4.1%	430k
33	Pf NT	Non-temporal data	1.0%	141k

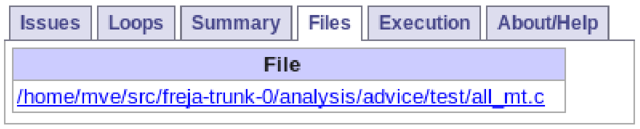
Figure 7.15. The Pollution Issues Tab

The pollution issues tab lists issues related to cache pollution and non-temporal data.

The following columns are available in the pollution issue list:

Issue number	Numerical identifier for the current issue. Clicking on this number will bring up information about the issue in the issue frame, see Section 7.4.4, “Issue Details”. It will also bring up the source code for the related to the issue in the source frame.
Issue type	Issue type and icon. The issue icon is also shown in the source frame.
% of fetches	The percentage of all fetches of the application that are caused by these non-temporal reuses.
Required cache size	An estimate of the cache size that would be required for the fetch ratio the non-temporal reuses to fall below 80%.

7.3.7. The Files Tab



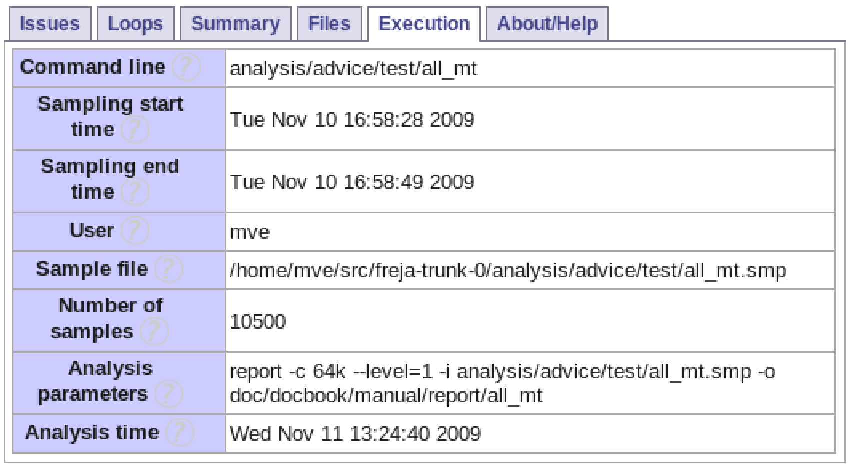
File
/home/mve/src/freja-trunk-0/analysis/advice/test/all_mt.c

Figure 7.16. The Files Tab

The files tab lists the source code files of the application. Clicking on a file opens it in the source code frame.

This tab also lists all loaded modules, binaries and shared libraries (.so, .dll), along with their respective load addresses. This table is useful to understanding code paths and call stack chains in third-party code, for which no debug information is available.

7.3.8. The Execution Tab



Command line	analysis/advice/test/all_mt
Sampling start time	Tue Nov 10 16:58:28 2009
Sampling end time	Tue Nov 10 16:58:49 2009
User	mve
Sample file	/home/mve/src/freja-trunk-0/analysis/advice/test/all_mt.smp
Number of samples	10500
Analysis parameters	report -c 64k --level=1 -i analysis/advice/test/all_mt.smp -o doc/docbook/manual/report/all_mt
Analysis time	Wed Nov 11 13:24:40 2009

Figure 7.17. The Execution Tab

The execution tab shows some information about the sampling and analysis the report is based on.

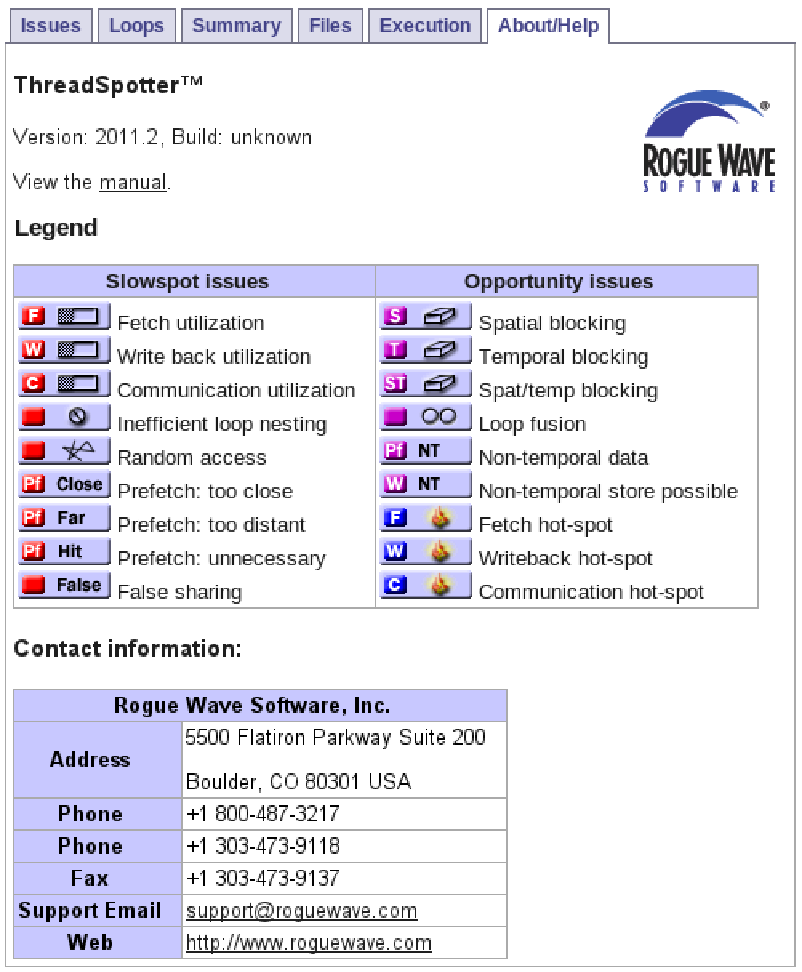
Command line	The command line of the sampled application. Reported to make it easier to reproduce a run.
Sampling start time	The time when the sampling of the application started.
Sampling end time	The time when the sampling of the application ended.
User	The user account that was used when sampling the application.
Sample file	The name of the sample file the report is based non.
Number of samples	The number of samples in the sample file.
Analysis parameters	The command line of the report command when generating the report. Provided to make it easier to repeat the same analysis.
Analysis time	The time when the report was generated.
Effective thread binding	A list of how the threads in the application populate the caches according to the thread binding currently in effect. *Cache number* denotes the sibling number of the cache at the selected cache level.

Thread id is the operating thread id number. Depending on the active thread bindning, one or more threads can operate in each cache.

For each thread, ThreadSpotter™ sequentially assigns a *virtual thread id*, starting with 0.

Use this table to see the effect of different thread bindings.

7.3.9. The About/Help Tab



ThreadSpotter™

Version: 2011.2, Build: unknown

View the manual.

Legend

Slowspot issues		Opportunity issues	
F	Fetch utilization	S	Spatial blocking
W	Write back utilization	T	Temporal blocking
C	Communication utilization	ST	Spat/temp blocking
	Inefficient loop nesting		Loop fusion
	Random access	Pf NT	Non-temporal data
Pf Close	Prefetch: too close	W NT	Non-temporal store possible
Pf Far	Prefetch: too distant	F	Fetch hot-spot
Pf Hit	Prefetch: unnecessary	W	Writeback hot-spot
False	False sharing	C	Communication hot-spot

Contact information:

Rogue Wave Software, Inc.	
Address	5500 Flatiron Parkway Suite 200 Boulder, CO 80301 USA
Phone	+1 800-487-3217
Phone	+1 303-473-9118
Fax	+1 303-473-9137
Support Email	support@roguewave.com
Web	http://www.roguewave.com

Figure 7.18. The About/Help Tab

The about/help tab shows version information for the ThreadSpotter™ package that generated the report, contact information for Rogue Wave Software and information about the license used to generate the report.

7.4. The Issue Frame

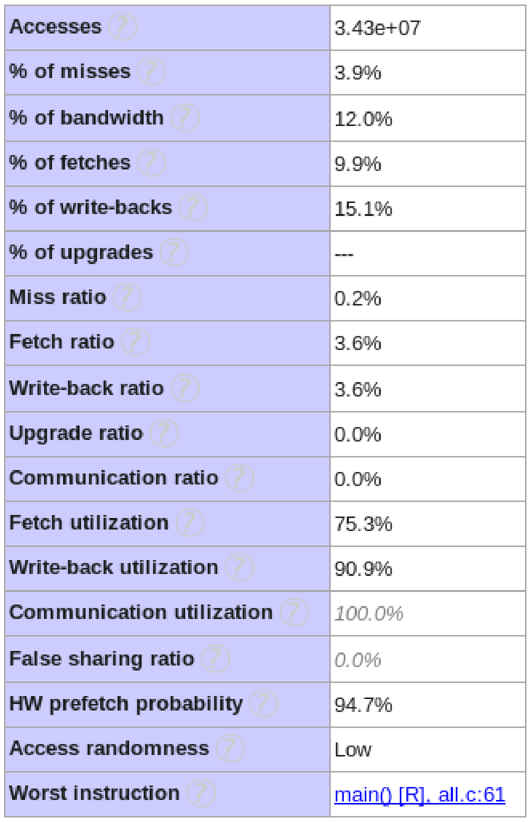
The issue frame shows detailed information about a specific loop or issue. The exact contents of the frame depend on whether it is displaying a loop or an issue, and if it is an issue on the type of issue.

The issue frame consists of a number of sections. By default most of them will be collapsed and only display the title of the section and an expand button, ➕, to the left of the title. Clicking the expand button shows the contents of the section.

To collapse an expanded section click the collapse button, ➖, to the left of the section title.

7.4.1. Statistics

When displaying details for loops, issues or instruction groups, the issue frame will begin with a statistics section.



Accesses	3.43e+07
% of misses	3.9%
% of bandwidth	12.0%
% of fetches	9.9%
% of write-backs	15.1%
% of upgrades	---
Miss ratio	0.2%
Fetch ratio	3.6%
Write-back ratio	3.6%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	75.3%
Write-back utilization	90.9%
Communication utilization	*100.0%*
False sharing ratio	*0.0%*
HW prefetch probability	94.7%
Access randomness	Low
Worst instruction	main() [R], all.c:61

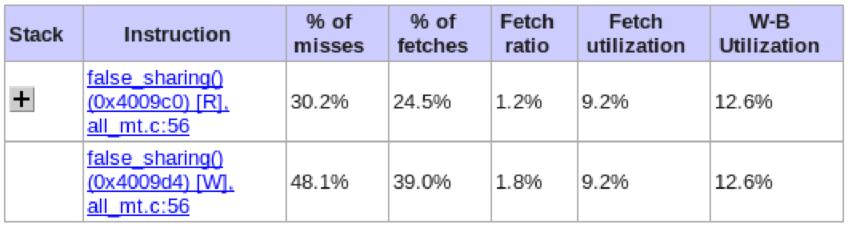
Figure 7.19. Issue Statistic Sections

The diagrams and all numbers presented here are for the currently selected part of the application. See Section 7.1.2, “Reading the Diagrams” and Section 7.1.1, “Reading the Statistics”.

In addition, for cache line utilization related advice, the statistics also report the fraction of the total fetches that would be avoided if the program was modified so that it achieves perfect utilization. See Section 4.8, “Utilization Corrected Fetch Ratio”. This helps you estimate the potential gain in addressing the issue. Other types of actions, such as loop fusion or blocking, can reduce the fetches even further.

7.4.2. Instructions

The issue frame also contains lists of instructions regardless of if it is showing details for a loop or for an issue.



Stack	Instruction	% of misses	% of fetches	Fetch ratio	Fetch utilization	W-B Utilization
⊞	false_sharing() (0x4009c0) [R], all_mt.c:56	30.2%	24.5%	1.2%	9.2%	12.6%
	false_sharing() (0x4009d4) [W], all_mt.c:56	48.1%	39.0%	1.8%	9.2%	12.6%

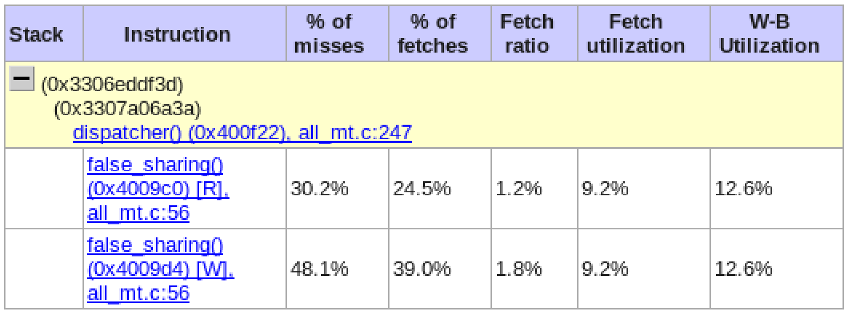
Figure 7.20. Instructions with Collapsed Call Stack

Figure 7.20, “Instructions with Collapsed Call Stack” shows an instruction group containing two instructions.

The address of each instruction is presented in the instruction column. If debug information is available, the function the instruction is part of, and the source file and line number that generated the instruction are also presented.

In this case the instructions at address 0x400516 and 0x400526 belong to the function `main` and is part of line 51 in the file `demo.c`.

The expand button, ⊞, in front of the line indicates that there is call stack information available telling us where the instruction was called from. Clicking the button expands the call stack information.



Stack	Instruction	% of misses	% of fetches	Fetch ratio	Fetch utilization	W-B Utilization
⊟ (0x3306eddf3d) (0x3307a06a3a) dispatcher() (0x400f22), all_mt.c:247						
	false_sharing() (0x4009c0) [R], all_mt.c:56	30.2%	24.5%	1.2%	9.2%	12.6%
	false_sharing() (0x4009d4) [W], all_mt.c:56	48.1%	39.0%	1.8%	9.2%	12.6%

Figure 7.21. Instructions with Expanded Call Stack

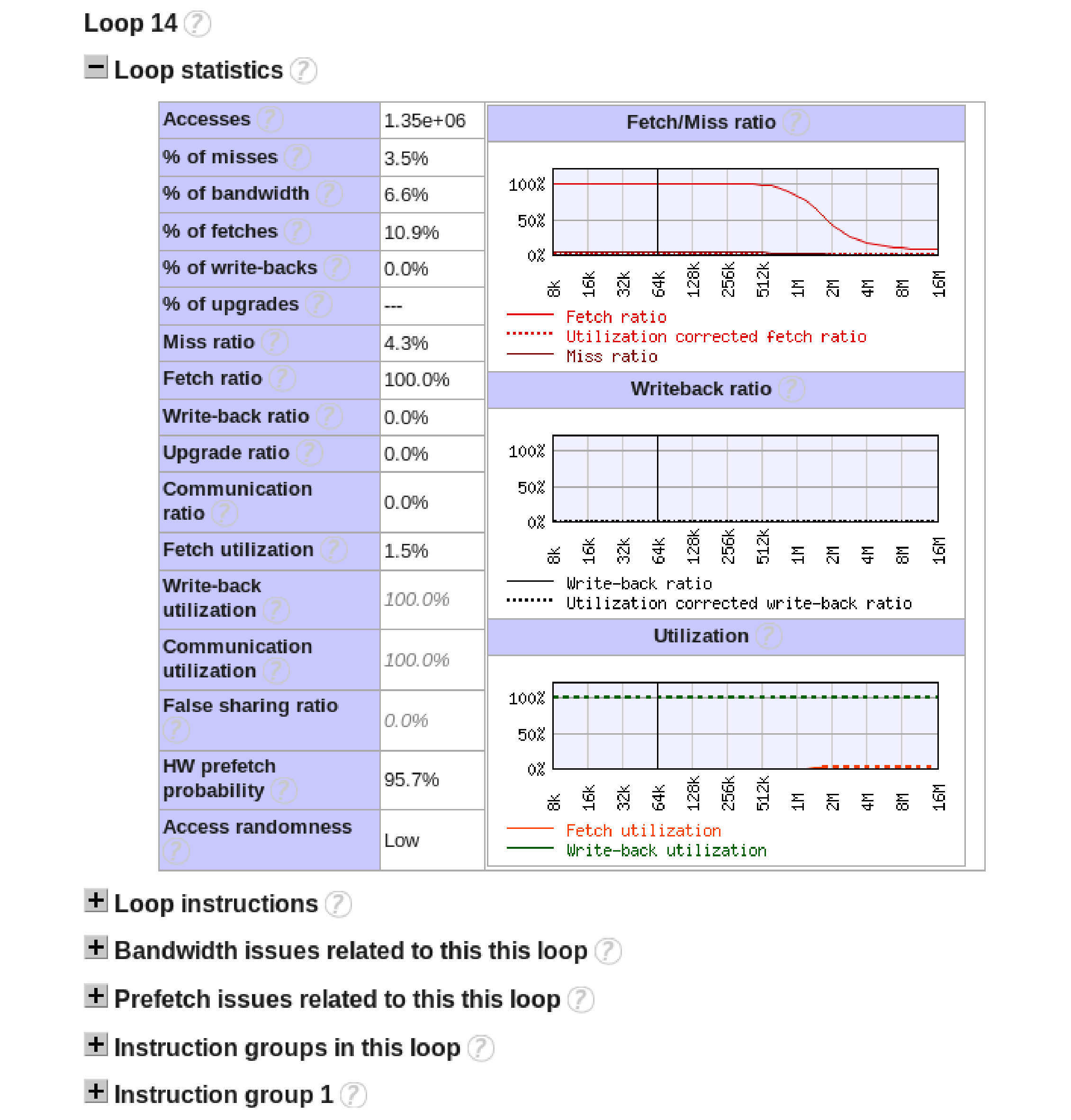
This tells us that the function the instructions are part of was called from the instruction at address 0x386fe1e074, and that function was in turn called from the instruction at address 0x4003b9. Debug information was not available for these functions, otherwise the functions names, file names and line numbers would have been displayed.

When there are several instructions with the same call stack, like in this case, the call stack is only shown for the first instruction with a particular call stack. When there are many different locations invoking a particular function, there would be many alternative call stacks. ThreadSpotter™ displays the dominant one, along with a percentage that describes the fraction of calls coming from that particular call site.

Depending on the call stack analysis depth, it can happen that the same instruction is reported more than once in an instruction group or a loop. In this case, the analysis engine has differentiated the statistics for this instruction depending on which functions it was called from.

7.4.3. Loop Details

The loop is the primary interesting structural element of a program from a cache performance standpoint, see Section 4.2, “Loops”.



Loop 14

− Loop statistics

Accesses	1.35e+06
% of misses	3.5%
% of bandwidth	6.6%
% of fetches	10.9%
% of write-backs	0.0%
% of upgrades	---
Miss ratio	4.3%
Fetch ratio	100.0%
Write-back ratio	0.0%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	1.5%
Write-back utilization	100.0%
Communication utilization	100.0%
False sharing ratio	0.0%
HW prefetch probability	95.7%
Access randomness	Low

+ Loop instructions

+ Bandwidth issues related to this this loop

+ Prefetch issues related to this this loop

+ Instruction groups in this loop

+ Instruction group 1

Figure 7.22. Loop

When the issue frame is displaying a loop it contains these sections:

- *Loop statistics*

  Statistics for the instructions of the loop, see Section 7.1, “Statistics”.

- *Loop instructions*

  List of all the instructions of the loop, see Section 7.4.2, “Instructions”.

- *Issues related to this loop*

  List of all issues related to this this loop.

- *Instruction groups in this loop*

List of all instruction groups in the loop, with some statistics.

- *Instruction group*

  There is a section for each instruction group in the loop. It contains statistics for the instruction group, a list of issues the instruction group is involved in and a list of the instructions in the instruction group.

7.4.4. Issue Details

There are a few common sections that occur in many of the issue types:

- *Instructions previously writing to related data*

  Points to the previous locations in the source code where the data structure involved in the issue was written, making it easier to determine which part of a complex expression is involved in the issue. See Section 4.4, “Last Writer” for more information. Some types of issues also point to the next instruction writing to related data.

- *Loop statistics*

  All numbers presented here are in reference to the entire loop containing the memory instructions involved in this performance issue, see Section 7.1, “Statistics”.

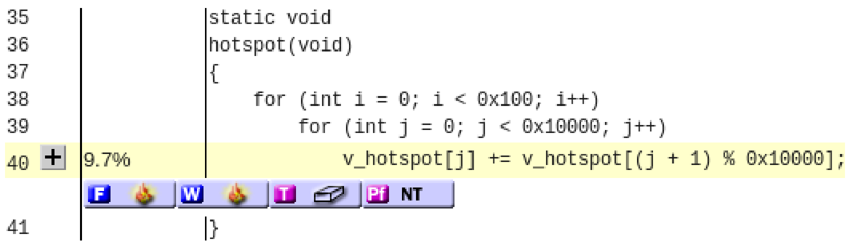
- *Loop instructions*

  Lists all the instructions in a loop. See Section 7.4.2, “Instructions”.

The exact contents of the issue frame varies with the type of issue, see Chapter 8, *Issue Reference* for more information about specific issue types.

7.5. The Source Code Frame

The frame to the right in the report displays the source code of the application.

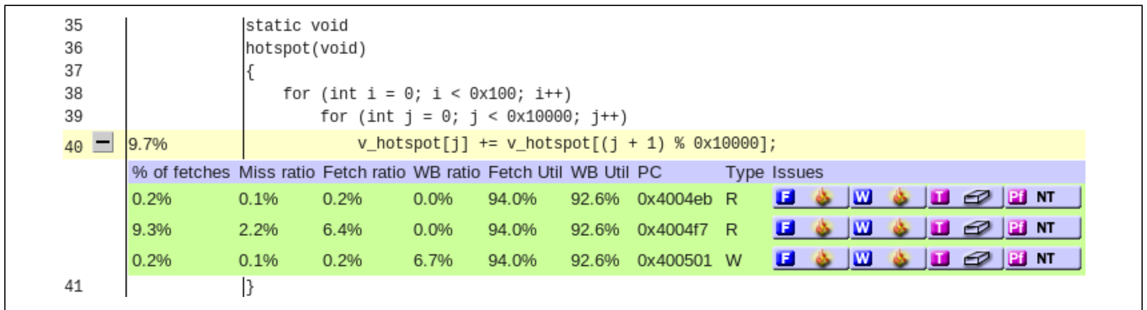


```
35              static void
36              hotspot(void)
37              {
38                  for (int i = 0; i < 0x100; i++)
39                      for (int j = 0; j < 0x10000; j++)
40 +   9.7%                 v_hotspot[j] += v_hotspot[(j + 1) % 0x10000];
41              }
```

Figure 7.23. Source Code with Collapsed Lines

Clicking on a source code line reference or on a file name in the *Files* tab opens a new page in the source code frame. As can be seen in Figure 7.23, “Source Code with Collapsed Lines”, the source code is annotated. There are three columns showing the line number, the percentage of all the cache line fetches of the application directly or indirectly caused by the line, and the source code itself.

Lines that significantly affect the memory behavior are highlighted in yellow and the line number column of each line in the currently selected loop is highlighted in purple. Each significant line also has a set of icons representing the issues it is causing, and there is and expand button you can click to show more information about the line.



```
                static void
                hotspot(void)
                {
                    for (int i = 0; i < 0x100; i++)
                        for (int j = 0; j < 0x10000; j++)
9.7%                        v_hotspot[j] += v_hotspot[(j + 1) % 0x10000];
                }
```

% of fetches	Miss ratio	Fetch ratio	WB ratio	Fetch Util	WB Util	PC	Type	Issues
0.2%	0.1%	0.2%	0.0%	94.0%	92.6%	0x4004eb	R	F W T Pf NT
9.3%	2.2%	6.4%	0.0%	94.0%	92.6%	0x4004f7	R	F W T Pf NT
0.2%	0.1%	0.2%	6.7%	94.0%	92.6%	0x400501	W	F W T Pf NT

Figure 7.24. Source Code with Expanded Lines

Clicking on the expand button shows more information about each significant machine code instructions belonging to the line, as seen in Figure 7.24, “Source Code with Expanded Lines”. If the function is called from different places and ThreadSpotter™ separates the different call stacks, the statistics for each instruction is display separate for each call stack, and the different call stacks are shown in alternating green shades.

The statistics shown for each instruction are:

- Percentage of all cache line fetches of the application caused by the instruction.
- The cache miss ratio of the instruction.

  Note that the miss ratios of prefetch instructions are displayed here. Misses caused by prefetch instructions are not included in the statistics of instruction groups, loops or the entire application, because unlike other cache misses prefetches that miss in the cache do not cause stalls. See Section 3.4, “Cache Misses” for more information.
- The fetch and write-back utilization of the instruction.
- The address of the instruction.
- The access type of the instruction; read and/or write, or prefetch.
- Icons for the issues that the instruction is involved in.

Chapter 8. Issue Reference

This chapter serves as a reference for the issue types that ThreadSpotter™ reports about.

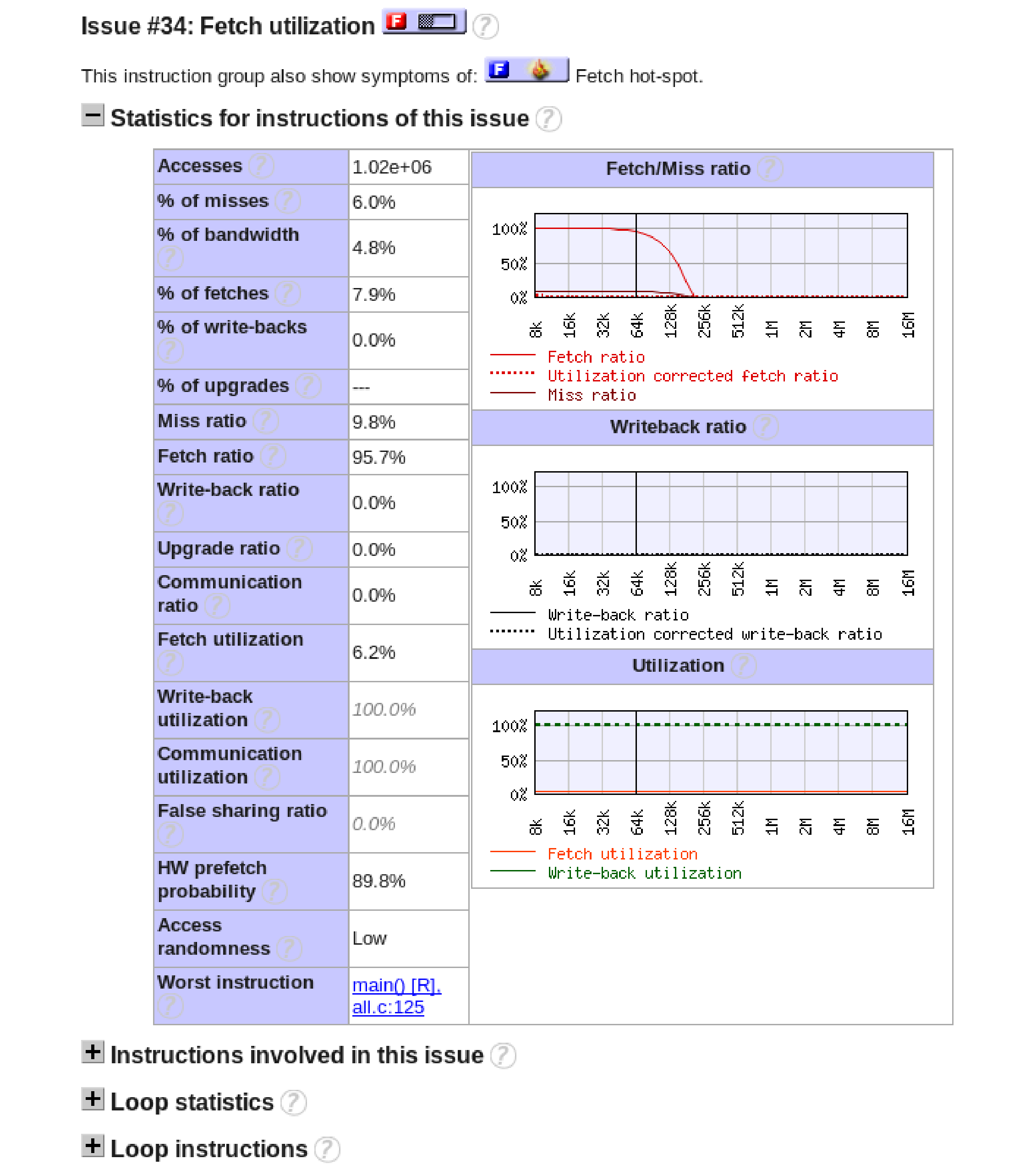
8.1. Utilization Issues

ThreadSpotter™ can identify three different utilization issues. Fetch and write-back utilization, which apply to the communication with memory or higher level caches that are local to the current thread. The communication utilization issue applies to communication between threads that are mapped to different caches.

Utilization issues can have a number of causes:

- There may be structures with unused fields, see Section 5.1.1, “Partially Used Structures”.
- There may be padding inserted into structures or between elements in an array to ensure data alignment, see Section 5.1.3, “Alignment Problems”.
- There may be housekeeping data from the dynamic memory allocation between data objects, see Section 5.1.4, “Dynamic Memory Allocation”.
- It may be caused by irregular access patterns, see Section 5.2.2, “Random Access Pattern”.
- It may be caused by iterating over a multidimensional array in an inefficient direction, see Section 5.2.1, “Inefficient Loop Nesting”.
- It may be caused by several threads accessing a common data set, partitioning the data set in an inappropriate way.

8.1.1. Fetch Utilization



Issue #34: Fetch utilization

This instruction group also show symptoms of: Fetch hot-spot.

Statistics for instructions of this issue

Accesses	1.02e+06
% of misses	6.0%
% of bandwidth	4.8%
% of fetches	7.9%
% of write-backs	0.0%
% of upgrades	---
Miss ratio	9.8%
Fetch ratio	95.7%
Write-back ratio	0.0%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	6.2%
Write-back utilization	100.0%
Communication utilization	100.0%
False sharing ratio	0.0%
HW prefetch probability	89.8%
Access randomness	Low
Worst instruction	main() [R], all.c:125

Instructions involved in this issue

Loop statistics

Loop instructions

Figure 8.1. Fetch Utilization Issue

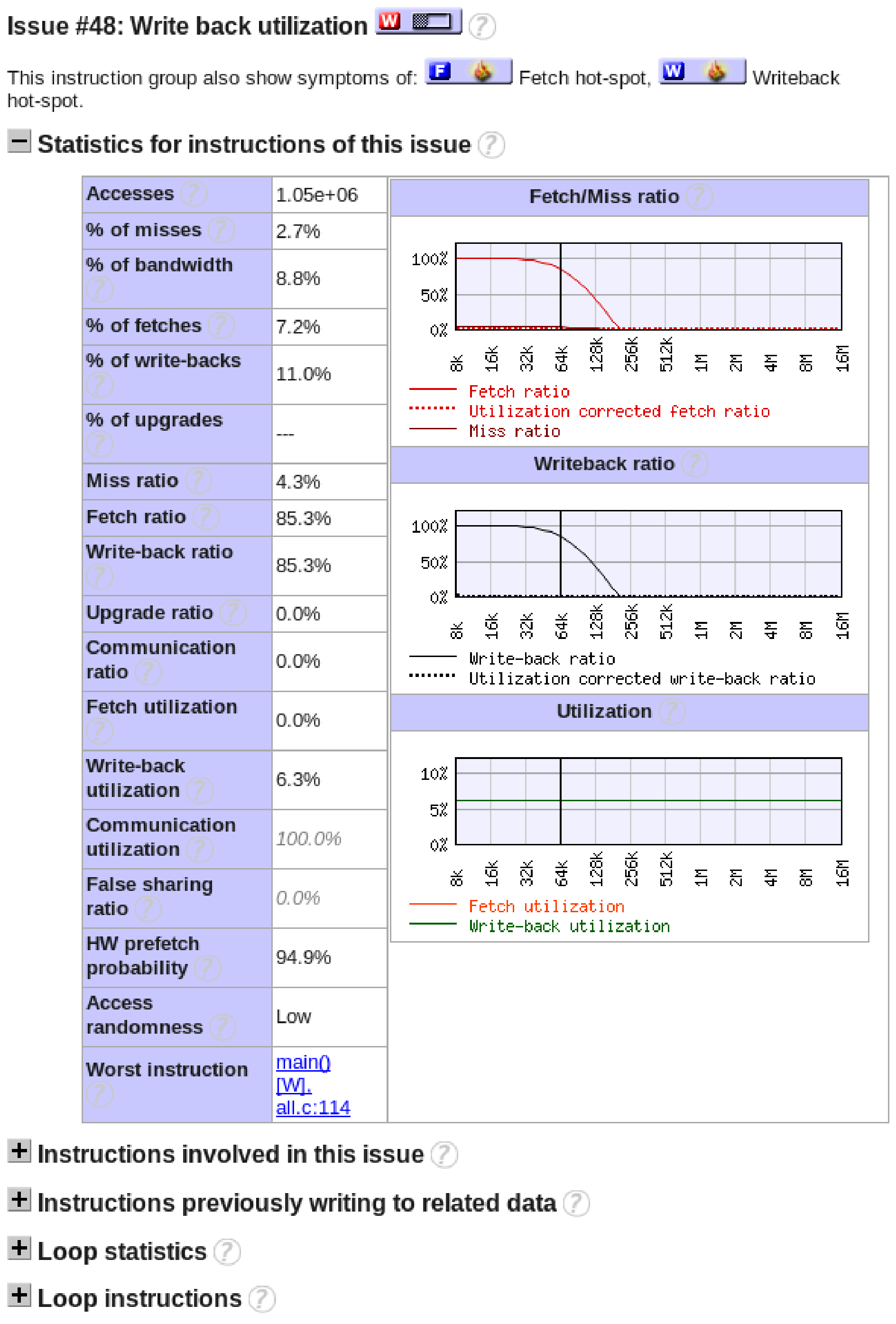
A *fetch utilization* issue indicates that a part of the application exhibits poor spatial locality, that is, cache lines are only partially read. The unused parts are still loaded into the cache, which means that memory bandwidth and cache space that could be used for useful data is wasted.

The fetch utilization issues has these sections:

- Statistics for instructions of this issue
- Instructions involved in this issue

- Instructions previously writing to related data
- Loop statistics
- Loop instructions

8.1.2. Write-Back Utilization



Issue #48: Write back utilization

This instruction group also show symptoms of: Fetch hot-spot, Writeback hot-spot.

Statistics for instructions of this issue

Accesses	1.05e+06
% of misses	2.7%
% of bandwidth	8.8%
% of fetches	7.2%
% of write-backs	11.0%
% of upgrades	---
Miss ratio	4.3%
Fetch ratio	85.3%
Write-back ratio	85.3%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	0.0%
Write-back utilization	6.3%
Communication utilization	100.0%
False sharing ratio	0.0%
HW prefetch probability	94.9%
Access randomness	Low
Worst instruction	main() [W], all.c:114

Instructions involved in this issue

Instructions previously writing to related data

Loop statistics

Loop instructions

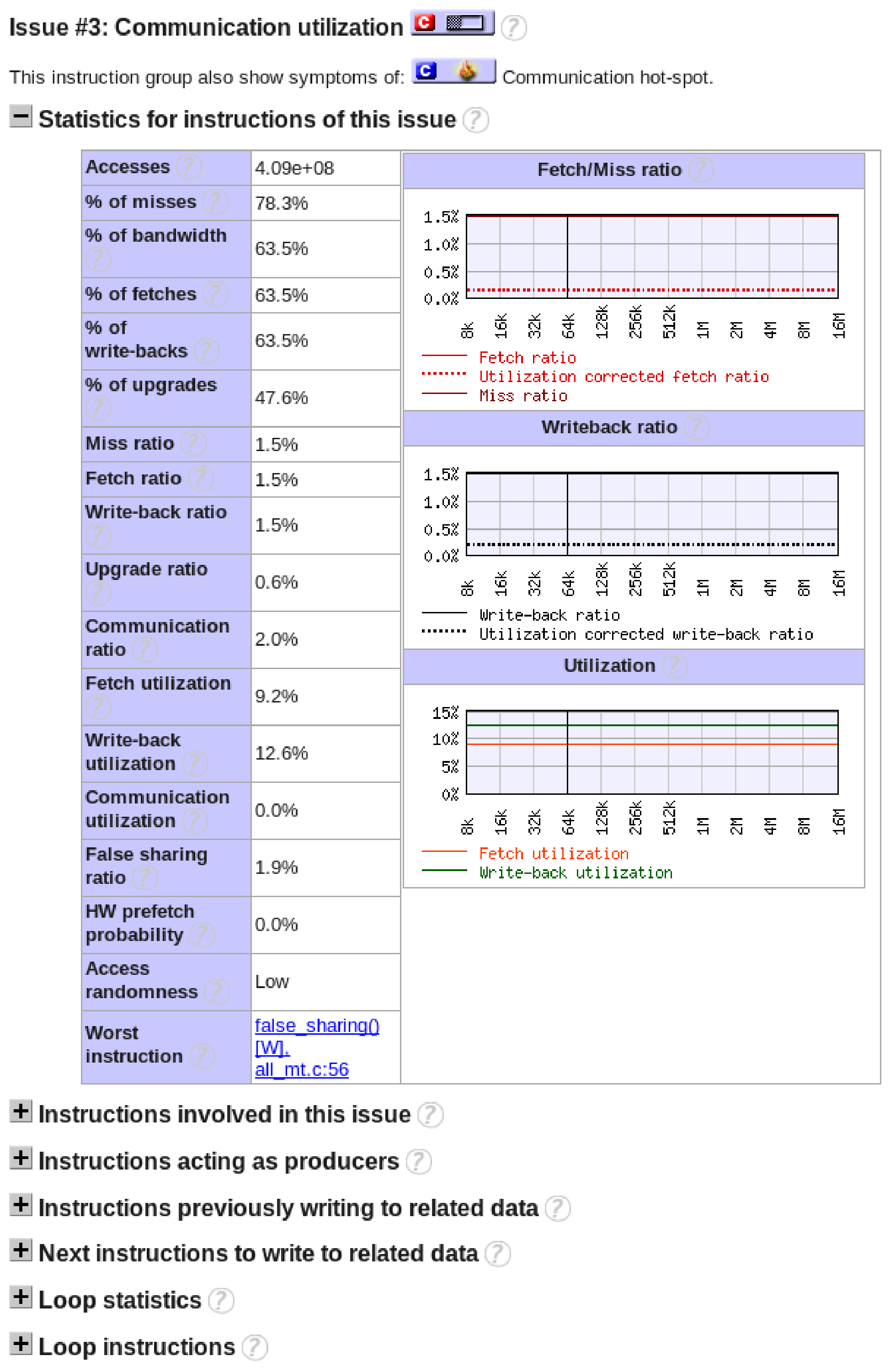
Figure 8.2. Write-Back Utilization Issue

A *write-back utilization* issue indicates that a part of the application has poor write-back utilization, that is, cache lines are only partially updated (written) by the application. The parts that were not updated will still be written back to memory, which results in wasted bandwidth.

The write-back utilization issues has these sections:

- Statistics for instructions of this issue
- Instructions involved in this issue
- Instructions previously writing to related data
- Loop statistics
- Loop instructions

8.1.3. Communication Utilization



Issue #3: Communication utilization

This instruction group also show symptoms of: Communication hot-spot.

Statistics for instructions of this issue

Accesses	4.09e+08
% of misses	78.3%
% of bandwidth	63.5%
% of fetches	63.5%
% of write-backs	63.5%
% of upgrades	47.6%
Miss ratio	1.5%
Fetch ratio	1.5%
Write-back ratio	1.5%
Upgrade ratio	0.6%
Communication ratio	2.0%
Fetch utilization	9.2%
Write-back utilization	12.6%
Communication utilization	0.0%
False sharing ratio	1.9%
HW prefetch probability	0.0%
Access randomness	Low
Worst instruction	false_sharing() [W], all_mt.c:56

Instructions involved in this issue

Instructions acting as producers

Instructions previously writing to related data

Next instructions to write to related data

Loop statistics

Loop instructions

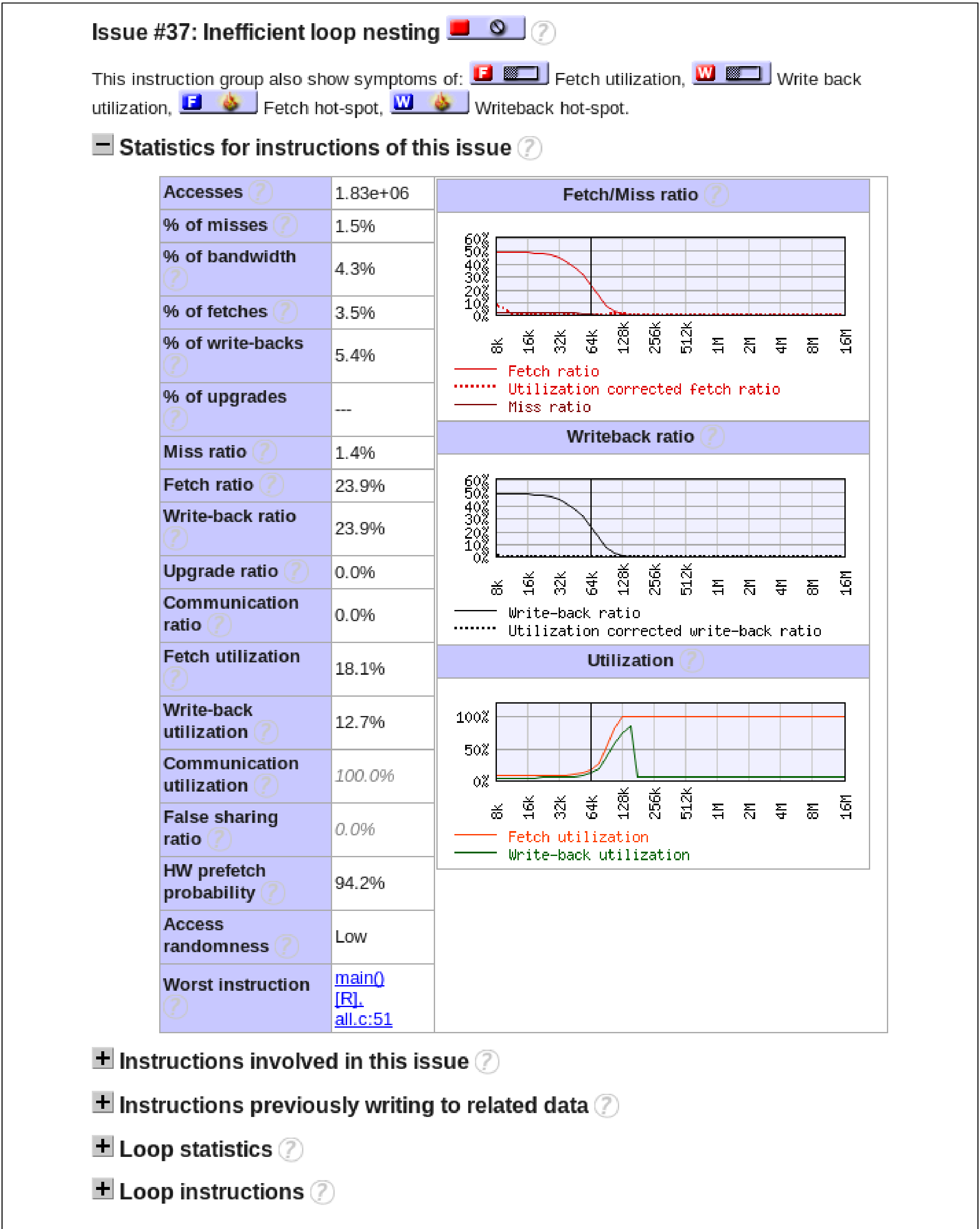
Figure 8.3. Communication Utilization Issue

A *communication utilization* issue is reported when ThreadSpotter™ finds two locations in different threads, or rather in two threads mapped to different caches, that communicate by reading and writing the same cache lines, but only use a small part of the cache lines in each communication cycle. By utilization a larger part of the cache lines for communication the amount of coherence misses in these locations could be reduced. See Section 5.4.2, “Poor Communication Utilization” for more information about the problem.

The communication utilization issue has these sections:

- Statistics for instructions of this issue
- Instructions involved in this issue
- Instructions acting as producers. These are the instructions in other threads that write the data that are being consumed by the instructions identified by this issue.
- Loop statistics
- Loop instructions

8.2. Inefficient Loop Nesting



Issue #37: Inefficient loop nesting

This instruction group also show symptoms of: Fetch utilization, Write back utilization, Fetch hot-spot, Writeback hot-spot.

Statistics for instructions of this issue

Accesses	1.83e+06
% of misses	1.5%
% of bandwidth	4.3%
% of fetches	3.5%
% of write-backs	5.4%
% of upgrades	---
Miss ratio	1.4%
Fetch ratio	23.9%
Write-back ratio	23.9%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	18.1%
Write-back utilization	12.7%
Communication utilization	*100.0%*
False sharing ratio	*0.0%*
HW prefetch probability	94.2%
Access randomness	Low
Worst instruction	main() [R]. all.c:51

Instructions involved in this issue

Instructions previously writing to related data

Loop statistics

Loop instructions

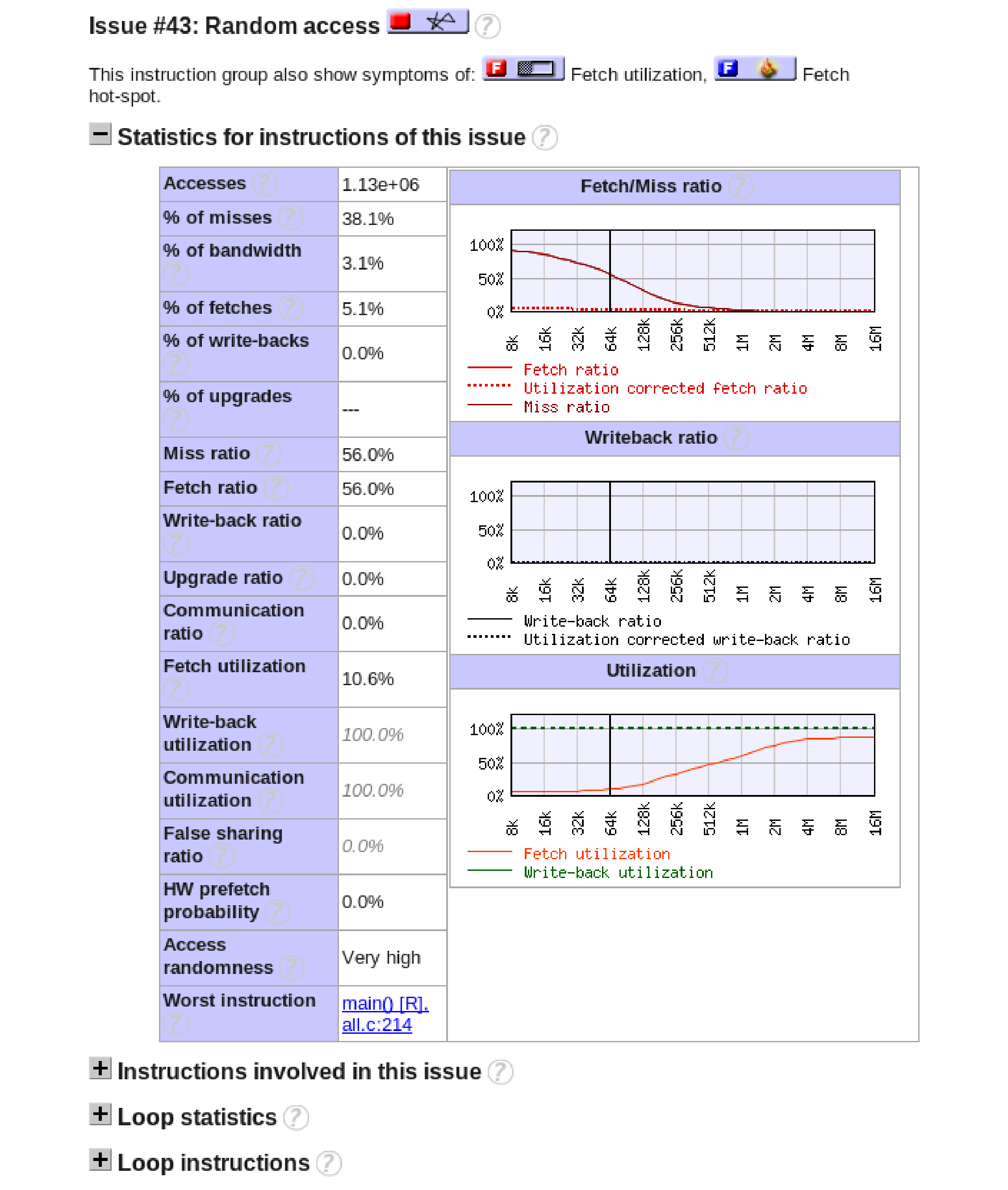
Figure 8.4. Inefficient Loop Nesting Issue

An *inefficient loop nesting* issue indicates that there are loops iterating through a multidimensional array in an inefficient order. See Section 5.2.1, “Inefficient Loop Nesting” for a general description of inefficient loop nesting.

The inefficient loop nesting issue only contains standard sections:

- Statistics for instructions of this issue
- Instructions involved in this issue
- Instructions previously writing to related data
- Loop statistics
- Loop instructions

8.3. Random Access Pattern



Issue #43: Random access

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

Statistics for instructions of this issue

Accesses	1.13e+06
% of misses	38.1%
% of bandwidth	3.1%
% of fetches	5.1%
% of write-backs	0.0%
% of upgrades	---
Miss ratio	56.0%
Fetch ratio	56.0%
Write-back ratio	0.0%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	10.6%
Write-back utilization	*100.0%*
Communication utilization	*100.0%*
False sharing ratio	*0.0%*
HW prefetch probability	0.0%
Access randomness	Very high
Worst instruction	main() [R], all.c:214

Instructions involved in this issue

Loop statistics

Loop instructions

Figure 8.5. Random Access Pattern Issue

A *random access pattern* issue is reported when a random or irregular memory access pattern that negatively affects the cache behavior of the application is found. Random access patterns are generally harmful to performance by reducing the cache line utilization and reducing the effectiveness of the hardware prefetcher. See Section 5.2.2, “Random Access Pattern” for more information.

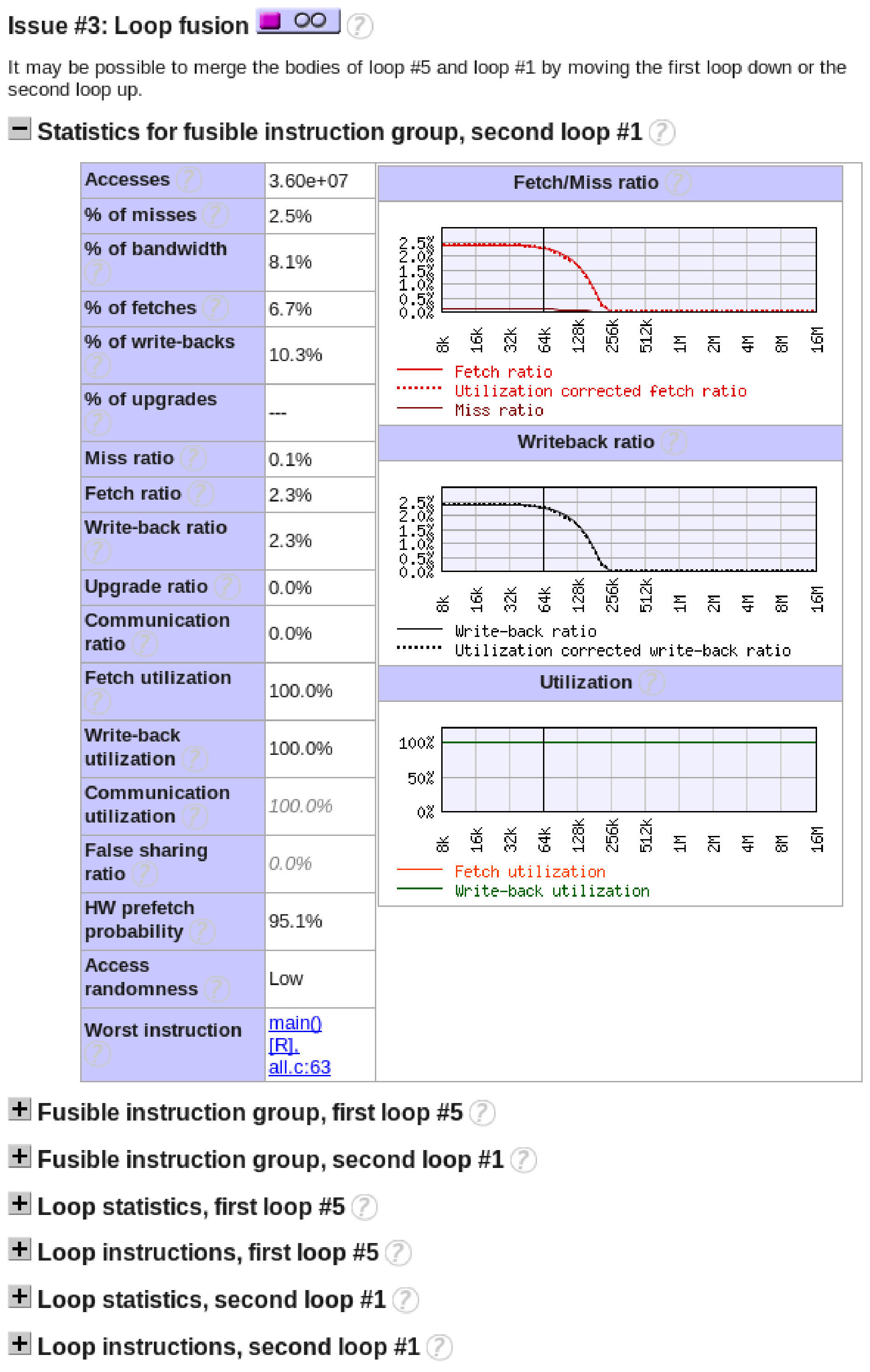
The random access pattern issue has these sections:

- Statistics for instructions of this issue
- Instructions involved in this issue
- Instructions previously writing to related data
- Loop statistics
- Loop instructions

Random access patterns can arise in various circumstances due to different underlying reasons, such as use of inappropriate data structures, use of algorithms that traverse data in complex patterns, and general use of dynamic memory allocation.

8.4. Loop Fusion

A *loop fusion* issue indicates that there are two loops iterating over the same data, but the accessed data is evicted from the cache between the loops. By moving the loops closer together or fusing them it may be possible to avoid the cache misses that the second loop now suffers from. See Section 5.2.3.2, “Loop Fusion” for a general description of loop fusion.



Issue #3: Loop fusion OO

It may be possible to merge the bodies of loop #5 and loop #1 by moving the first loop down or the second loop up.

Statistics for fusible instruction group, second loop #1

Accesses	3.60e+07
% of misses	2.5%
% of bandwidth	8.1%
% of fetches	6.7%
% of write-backs	10.3%
% of upgrades	---
Miss ratio	0.1%
Fetch ratio	2.3%
Write-back ratio	2.3%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	100.0%
Write-back utilization	100.0%
Communication utilization	*100.0%*
False sharing ratio	*0.0%*
HW prefetch probability	95.1%
Access randomness	Low
Worst instruction	main() [R]. all.c:63

Fetch/Miss ratio
2.5%
2.0%
1.5%
1.0%
0.5%
0.0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch ratio
Utilization corrected fetch ratio
Miss ratio
Writeback ratio
2.5%
2.0%
1.5%
1.0%
0.5%
0.0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Write-back ratio
Utilization corrected write-back ratio
Utilization
100%
50%
0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch utilization
Write-back utilization

Fusible instruction group, first loop #5

Fusible instruction group, second loop #1

Loop statistics, first loop #5

Loop instructions, first loop #5

Loop statistics, second loop #1

Loop instructions, second loop #1

Figure 8.6. Loop Fusion Issue

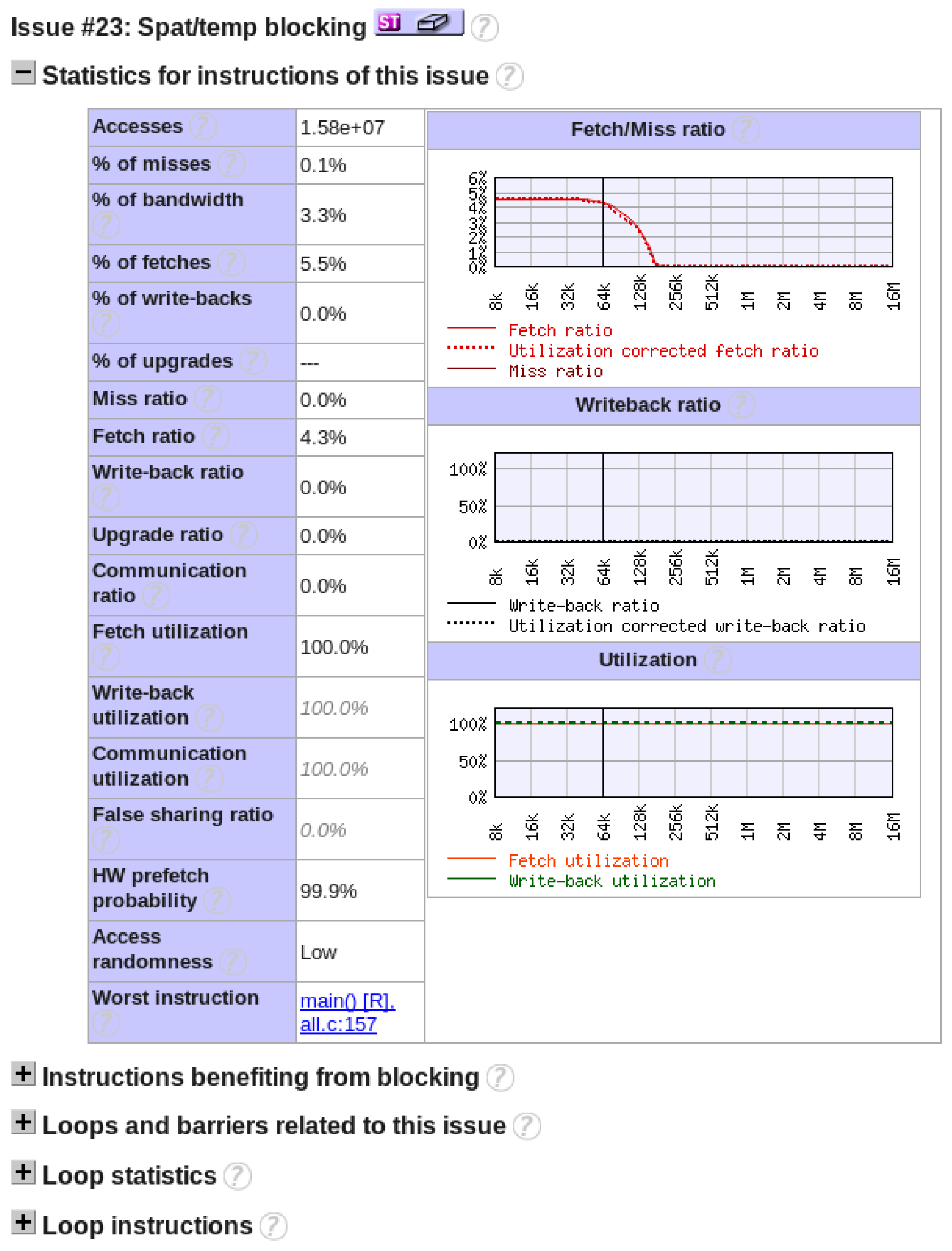
A loop fusion issue involves two loops. Since it is the second of the loops that will benefit from merging the loops, it is the statistics for the instruction group in that loop that are displayed.

The loop fusion issue contains some standard sections, *instructions*, *loop statistics* and *loop instructions*. However, since there are two loops involved you get one one of each for both of the two loops.

The loop fusion issue also contains a couple of sections pointing out potential barriers for fusing the loops. See Section 5.2.3.2, “Loop Fusion”.

- Statistics for fusible instruction group, second loop
- Fusible instruction group, first loop
- Fusible instruction group, second loop
- Potential barriers for moving the body of the first loop
- Potential barriers for moving the body of the second loop
- Loop statistics, first loop
- Loop instructions, first loop
- Loop statistics, second loop
- Loop instructions, second loop

8.5. Blocking



Issue #23: Spat/temp blocking

Statistics for instructions of this issue

Accesses	1.58e+07
% of misses	0.1%
% of bandwidth	3.3%
% of fetches	5.5%
% of write-backs	0.0%
% of upgrades	---
Miss ratio	0.0%
Fetch ratio	4.3%
Write-back ratio	0.0%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	100.0%
Write-back utilization	*100.0%*
Communication utilization	*100.0%*
False sharing ratio	*0.0%*
HW prefetch probability	99.9%
Access randomness	Low
Worst instruction	main() [R], all.c:157

Fetch/Miss ratio
6%
5%
4%
3%
2%
1%
0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch ratio
Utilization corrected fetch ratio
Miss ratio
Writeback ratio
100%
50%
0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Write-back ratio
Utilization corrected write-back ratio
Utilization
100%
50%
0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch utilization
Write-back utilization

Instructions benefiting from blocking

Loops and barriers related to this issue

Loop statistics

Loop instructions

Figure 8.7. Blocking Issue

A *blocking* issue means that there is an opportunity to reduce the number of cache misses or fetches by processing a smaller piece of the data set at a time, thereby reusing cache lines before they are evicted from the cache. For a general description of blocking, see Section 5.2.3.1, “Blocking”.

ThreadSpotter™ can suggest three types of blocking:

- *Temporal blocking.*

It may be possible to increase the reuse of the same data that has already been used. Typically occurs when an algorithm performs multiple iterations over a data set that is too large to fit in the cache. By performing multiple iterations at a time on a smaller part of the data that fits in the cache, the cache line reuse can be improved.

- *Spatial blocking.*

  It may be possible to increase the reuse of other data in the cache lines that have already been used. Typically occurs within an iteration of an algorithm that touches too many other locations in the data set before reusing a location in the original cache line. By running the algorithm on smaller parts of the data set at a time the cache line reuse can be improved.

- *Spatial and temporal blocking.*

  It may be possible improve both kinds of reuse, typically both between iterations and within iterations of an algorithm.

The blocking issues contain several sections describing different loop levels in the loop hierarchy related to the issue, descriptions of these and other issue specific sections follows. The issues contain the following sections:

- Statistics for instructions of this issue
- Instructions benefiting from blocking
- Instructions previously writing to related data (and also: Next instructions to write to related data).
- Loops and barriers related to this issue
  - Outermost loop without barriers
  - Next outer loop, introducing barriers
  - Optimal loop level to block at
- Loop statistics
- Loop instructions

Different loops play different roles for blocking with respect to data reuse of a particular instruction. The following example contains four loop levels, some of which possess important traits to consider when blocking this piece of code. From the inside out:

```
for (i = 0; i < ITER; i++)
  for (j = 0; j < J_SIZE; j++) {
    a[j][0] = 0;
    for (k = 0; k < K_SIZE; k++)
      for (m = 0; m < M_SIZE; m++)
        b[m] += a[j][k];
  }
```

- *Instructions benefiting from blocking*

The instructions whose cache line reuse may be possible to improve.

for m is the loop containing the target instruction `a[][]`. The goal is to have this instruction revisit data more aggressively, even before all its slots have been visited.

- *Outermost loop without barriers.*

  The outermost loop without barriers (data dependencies). This may represent a suitable loop level to block at.

  for k is the outermost loop not having any blocking write instructions. (In this example, we find such a blocking write instruction in the next outer loop level: `a[j][0] = ...`).

- *Next outer loop, introducing barriers.*

  The innermost loop with data dependencies on some variable (which does not have to be the target instruction of the blocking issue). Blocking cannot normally be performed on this or on outer loop levels. The code may first need to be transformed to remove data dependency barriers. All barriers that have been detected are reported in this section.

  for j is the next outer loop level with barriers in this example. (In this case, however, the barriers are inconsequential to the semantics of the program, at least for the example transform below, and would not need to be removed.)

- *Target loop*

  The optimal, outer loop level to block around.

  for i is the optimal loop level to use as the outer loop. All values in `a[][]` are used in each iteration of this loop level.

  To block this loop hierarchy in an optimal way, this is the loop level which should be moved closer to the instruction under consideration, by making each iteration of the inner loops work on a smaller chunk of data. Another way to say this is to split one or several loop nestings outwards past this loop level.

The following would be one proper blocking of this loop. The `for j` loop has been split up by the `for i` loop:

```
for (jj = 0; jj < J_SIZE; jj += BLOCK)
  for (i = 0; i < ITER; i++)
    for (j = jj; j < min(jj + BLOCK, J_SIZE); j++) {
      a[j][0] = 0;
      for (k = 0; k < K_SIZE; k++)
        for (m = 0; m < M_SIZE; m++)
          b[m] += a[j][k];
    }
```

The ideas underlying the blocking issue, and the sections presented here represents an ideal situation. In reality, due to effects of the sampling process, ThreadSpotter™ may not have complete information about all loop levels. In such cases the closest identified loop level tends to be selected.

To help the programmer to judge how close the identified loop is from the optimal blocking level, ThreadSpotter™ displays the number of iterations of that loop that corresponds to one data reuse by the instruction under consideration. The programmer can take this number and, while moving outwards from the target instruction, scan loops until the reported iteration count has been accounted for.

8.6. Software Prefetch Issues

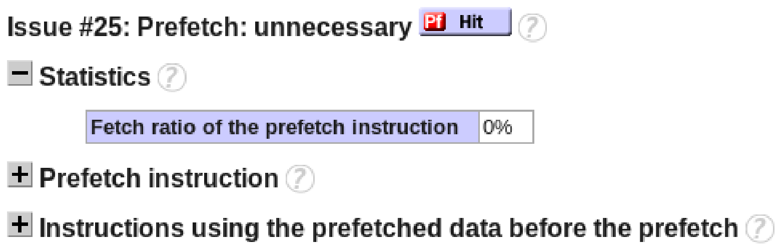
ThreadSpotter™ can identify three types of issues concerning software prefetch instructions; *prefetch unnecessary*, *prefetch too distant* and *prefetch too close*.

The information presented about prefetch issues differs a lot from that of other issues, and it uses none of the standard sections. The different types of prefetch advice only have one section in common:

- *Prefetch instruction*

  Points to the prefetch instruction itself.

8.6.1. Prefetch Unnecessary



Issue #25: Prefetch: unnecessary Pf Hit

Statistics

Fetch ratio of the prefetch instruction	0%

Prefetch instruction

Instructions using the prefetched data before the prefetch

Figure 8.8. Prefetch Unnecessary Issue

A *prefetch unnecessary* issue is reported when a prefetch instruction that nearly always hits in the cache is found. The percentage of times this is the case is given by the fetch ratio of the instruction presented by ThreadSpotter™. Prefetch instructions that almost always hit in the cache consume execution resources without providing any benefit.

The prefetch unnecessary issue has the following sections:

- *Fetch ratio of the prefetch instruction*

  The fetch ratio of the prefetch instruction.

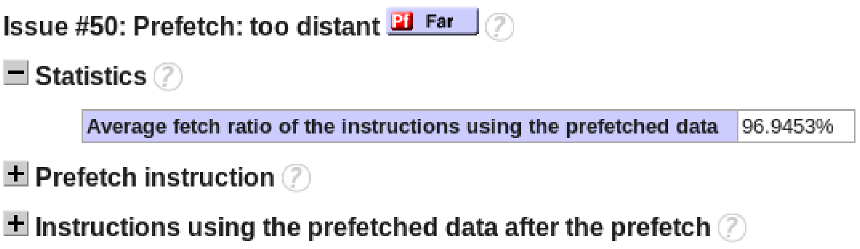
- *Instructions using the prefetched data before the prefetch*

  Lists the instructions that last touched the data the before the prefetch instruction. This is useful when trying to understand why the data already is in the cache, for example, to check if the data is brought into the cache by some part of the application that you did not expect.

A reasonable rule of thumb is that the fetch ratio of prefetch instructions should be above 10% for prefetches from the L2 cache or above 1% for prefetches from RAM. If the fetch ratio is too low the data that the prefetch instruction is prefetching is almost always already in the cache, so the prefetch instruction is not doing useful work and may instead decrease performance by consuming execution resources.

Check that the data is not brought into the cache by some part of the application that you did not expect. Make sure that you do not prefetch the same cache line multiple times. If that is not the case, consider removing the prefetch.

8.6.2. Prefetch too Distant


Issue #50: Prefetch: too distant

Statistics

Average fetch ratio of the instructions using the prefetched data	96.9453%

Prefetch instruction

Instructions using the prefetched data after the prefetch

Figure 8.9. Prefetch too Distant Issue

A *prefetch too distant* issue is reported when a prefetch instruction that fetches data that is not used before it is evicted from the cache again is found. The prefetch instruction may be placed too far ahead of the instructions that use the prefetched data, or the data may not be used as expected. Such a prefetch will consume execution resources and memory bandwidth without providing any benefit.

The prefetch too distant issue has the following sections:

- *Average fetch ratio of the instructions using the prefetched data*

  The fetch ratio of the instructions using the prefetched data. Tells you to what degree the prefetched data is evicted again before it is used.

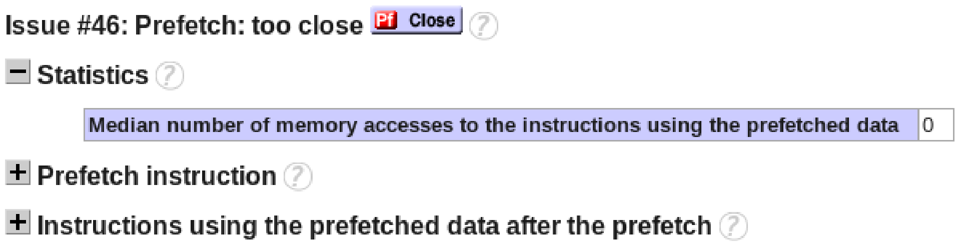
- *Instructions using the prefetched data after the prefetch*

  Lists the instructions that use the prefetched data.

Consider reducing the distance between the prefetch and the instructions using the prefetched data, for example, if you are prefetching data a number of iterations ahead in a loop consider reducing that number of iterations.

This issue can also indicate that the data fetched by the prefetch instruction is actually never used as intended. Check that it is the intended instructions that use the data.

8.6.3. Prefetch too Close


Issue #46: Prefetch: too close

Statistics

Median number of memory accesses to the instructions using the prefetched data	0

Prefetch instruction

Instructions using the prefetched data after the prefetch

Figure 8.10. Prefetch too Close Issue

A *prefetch too close* issue is reported when a prefetch instruction that is too close to the instruction using the data is found. When prefetch instruction is too close the prefetched data does not have time to arrive from the next cache level or main memory before it is needed. The instruction using the data still stalls for some time and you do not get the full benefit of the prefetch.

The prefetch too close issue has the following sections:

- *Median number of memory accesses to the instructions using the prefetched data*

  The number of memory accesses between the prefetch instruction and the instructions using the prefetched data.

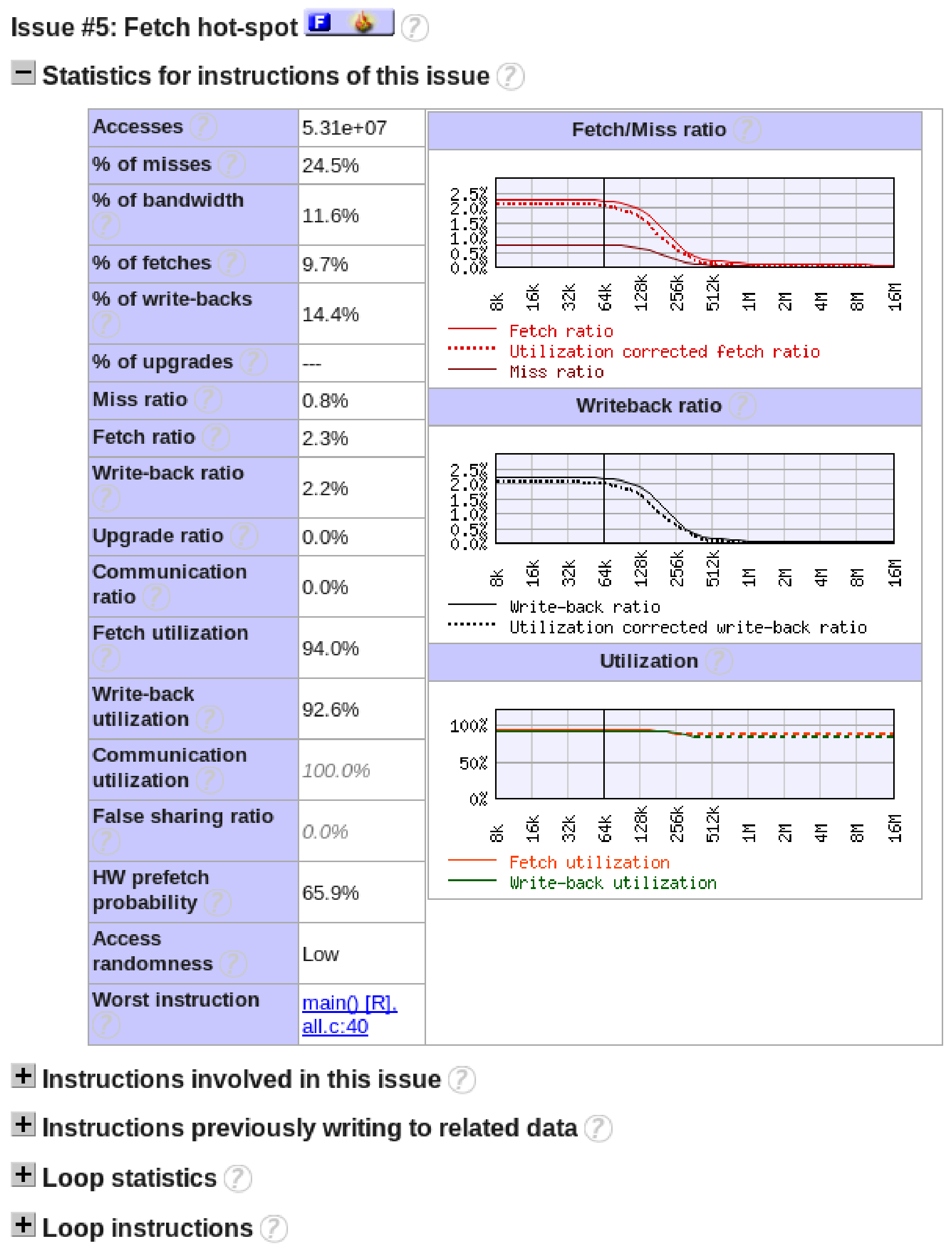
- *Instructions using the prefetched data after the prefetch*

  Lists the instructions that use the prefetched data.

ThreadSpotter™ presents the distance as the median number of memory accesses before the next use of the prefetched data. It also presents which instructions are the next to touch the data. The required distance depends on the latency of the cache level or main memory the data is fetched from. A reasonable rule of thumb is that the distance should be at least 3 accesses for prefetches from the L2 cache or at least 30 accesses for prefetches from RAM.

Consider increasing the distance between the prefetch and the instructions using the prefetched data, for example, if you are prefetching data a number of iterations ahead in a loop consider increasing that number of iterations.

8.7. Fetch Hot-Spot



Issue #5: Fetch hot-spot

Statistics for instructions of this issue

Accesses	5.31e+07
% of misses	24.5%
% of bandwidth	11.6%
% of fetches	9.7%
% of write-backs	14.4%
% of upgrades	---
Miss ratio	0.8%
Fetch ratio	2.3%
Write-back ratio	2.2%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	94.0%
Write-back utilization	92.6%
Communication utilization	100.0%
False sharing ratio	0.0%
HW prefetch probability	65.9%
Access randomness	Low
Worst instruction	main() [R], all.c:40

Fetch/Miss ratio
2.5%
2.0%
1.5%
1.0%
0.5%
0.0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch ratio
Utilization corrected fetch ratio
Miss ratio
Writeback ratio
2.5%
2.0%
1.5%
1.0%
0.5%
0.0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Write-back ratio
Utilization corrected write-back ratio
Utilization
100%
50%
0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch utilization
Write-back utilization

Instructions involved in this issue

Instructions previously writing to related data

Loop statistics

Loop instructions

Figure 8.11. Fetch Hot-Spot Issue

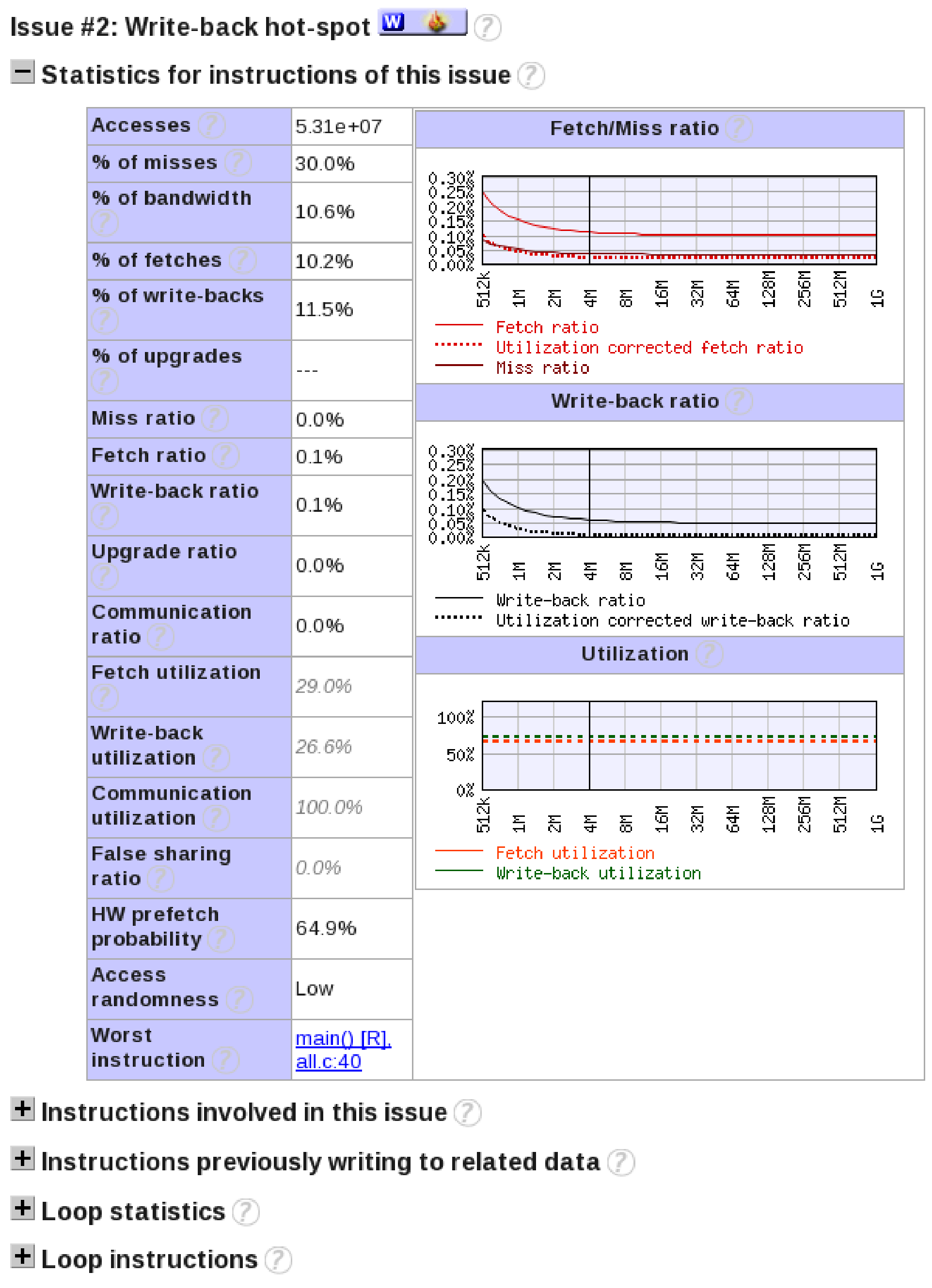
A *fetch hot-spot* issue is reported when ThreadSpotter™ finds a location in the application that causes large numbers cache line fetches, but where there is no obvious way to improve the behavior. It may still be possible to reduce the performance impact, for example, by adding software prefetches. See Section 5.6, “Final Remedies” for more information.

The fetch hot-spot issue has these sections:

- Statistics for instructions of this issue

- Instructions involved in this issue
- Instructions previously writing to related data
- Loop statistics
- Loop instructions

8.8. Write-back Hot-Spot



Issue #2: Write-back hot-spot

Statistics for instructions of this issue

Accesses	5.31e+07
% of misses	30.0%
% of bandwidth	10.6%
% of fetches	10.2%
% of write-backs	11.5%
% of upgrades	---
Miss ratio	0.0%
Fetch ratio	0.1%
Write-back ratio	0.1%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	29.0%
Write-back utilization	26.6%
Communication utilization	100.0%
False sharing ratio	0.0%
HW prefetch probability	64.9%
Access randomness	Low
Worst instruction	main() [R], all.c:40

Fetch/Miss ratio
0.30%
0.25%
0.20%
0.15%
0.10%
0.05%
0.00%
512k 1M 2M 4M 8M 16M 32M 64M 128M 256M 512M 1G
Fetch ratio
Utilization corrected fetch ratio
Miss ratio
Write-back ratio
0.30%
0.25%
0.20%
0.15%
0.10%
0.05%
0.00%
512k 1M 2M 4M 8M 16M 32M 64M 128M 256M 512M 1G
Write-back ratio
Utilization corrected write-back ratio
Utilization
100%
50%
0%
512k 1M 2M 4M 8M 16M 32M 64M 128M 256M 512M 1G
Fetch utilization
Write-back utilization

Instructions involved in this issue

Instructions previously writing to related data

Loop statistics

Loop instructions

Figure 8.12. Write-back Hot-Spot Issue

A *write-back hot-spot* issue is reported when ThreadSpotter™ finds a location in the application that causes large numbers of cache-line write-backs, but where there is no obvious way of improving the behavior.

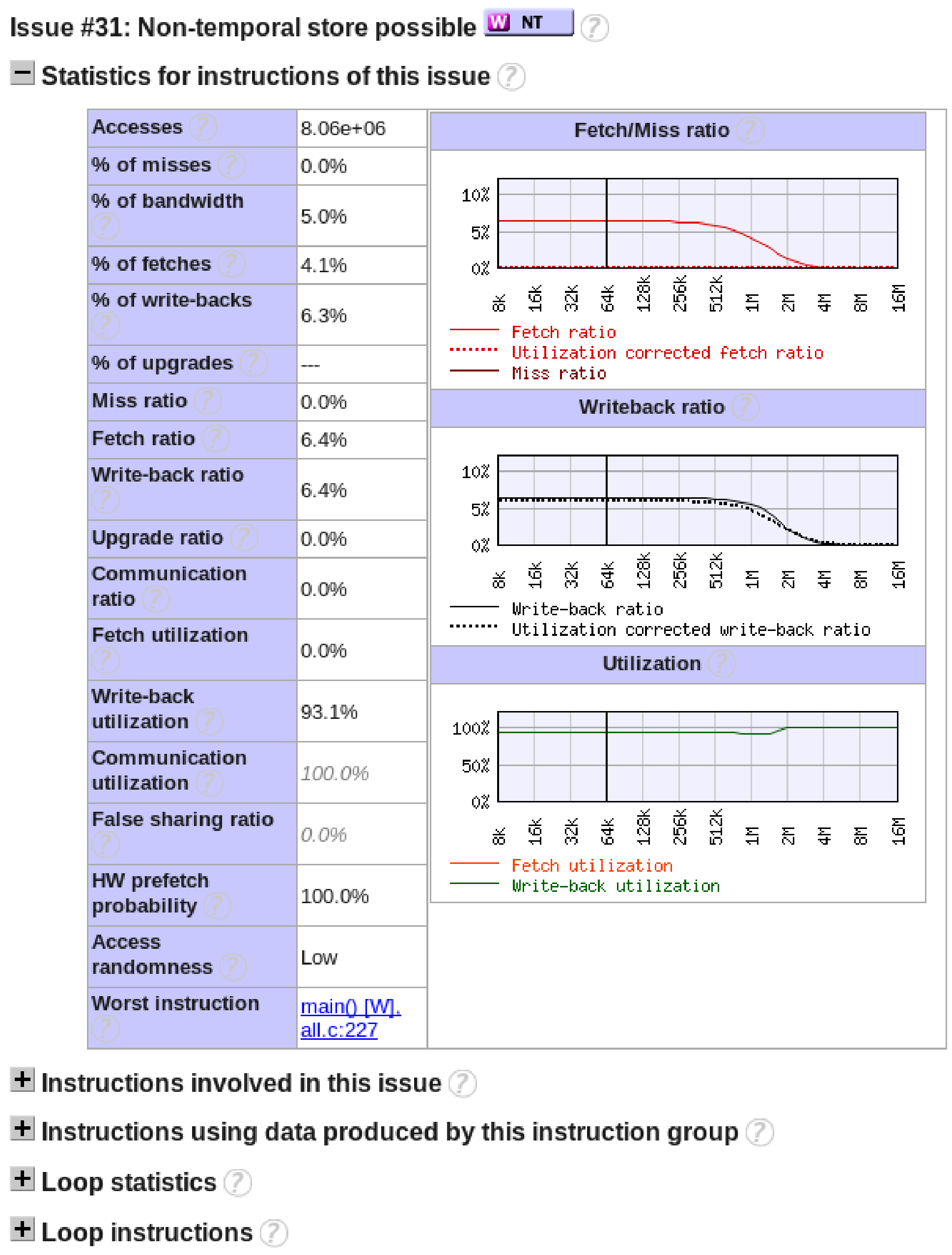
The write-back hot-spot issue has these sections:

- Statistics for instructions of this issue
- Instructions involved in this issue
- Instructions previously writing to related data
- Loop statistics
- Loop instructions

8.9. Non-Temporal Store Possible

A *non-temporal store possible* issue is reported when ThreadSpotter™ sees an opportunity to decrease bandwidth usage using non-temporal stores instead of regular stores. Such opportunities arise when data written in a loop lacks temporal reuse. See Section 5.3, “Non-Temporal Data” for more information about non-temporal memory accesses.

This issue type is normally only included when analyzing the highest cache level, that is the cache level closest to memory. This is due to the non-temporal store instructions using hardware buffers that are located between the last level cache and main memory.



Issue #31: Non-temporal store possible W NT

Statistics for instructions of this issue

Accesses	8.06e+06
% of misses	0.0%
% of bandwidth	5.0%
% of fetches	4.1%
% of write-backs	6.3%
% of upgrades	---
Miss ratio	0.0%
Fetch ratio	6.4%
Write-back ratio	6.4%
Upgrade ratio	0.0%
Communication ratio	0.0%
Fetch utilization	0.0%
Write-back utilization	93.1%
Communication utilization	*100.0%*
False sharing ratio	*0.0%*
HW prefetch probability	100.0%
Access randomness	Low
Worst instruction	main() [W], all.c:227

Instructions involved in this issue

Instructions using data produced by this instruction group

Loop statistics

Loop instructions

Figure 8.13. Non-Temporal Store Possible Issue

Using non-temporal stores prevents non-temporal data from polluting the caches and leaves more space for temporal data. Another effect of the non-temporal hint is that it allows the processors to avoid fetching cache lines that will only be written, which effectively doubles the available bandwidth for a given write loop.

Non-temporal stores should only be used for truly non-temporal data and is no replacement for techniques that improve the temporal locality of the code. Using non-temporal stores for temporal data will hurt performance by removing the data from the cache hierarchy.

The non-temporal store possible issue has the following sections:

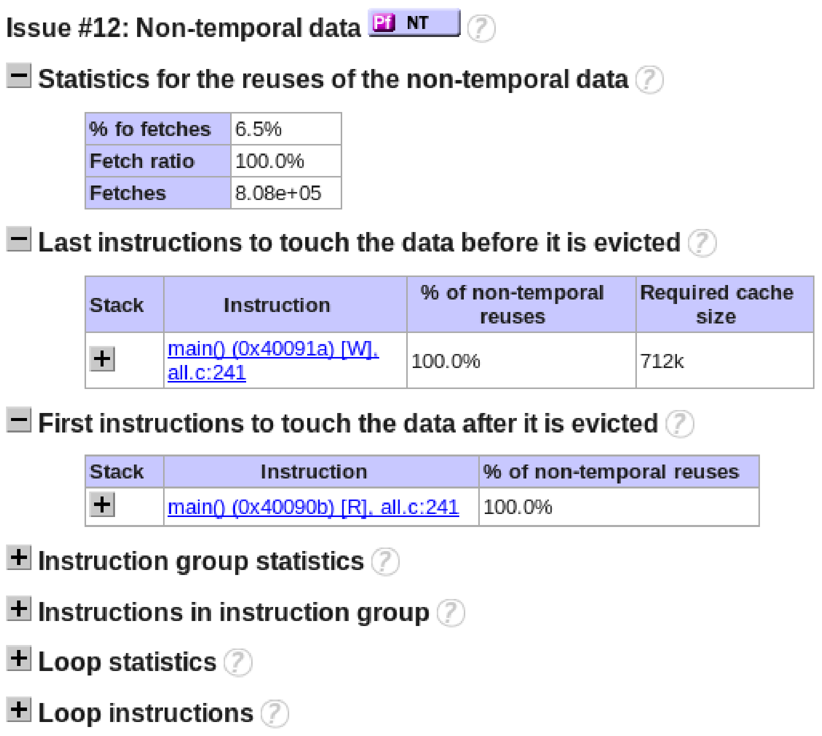
- Statistics for instructions of this issue
- Instructions involved in this issue
- Instructions using data produced by the instructions identified in this issue.
- Loop statistics
- Loop instructions

8.10. Non-Temporal Data

A *non-temporal data* issue is reported when ThreadSpotter™ finds places where accessed cache lines are nearly always evicted from the cache before being reused. However, the cache lines still occupy space in the cache, that could otherwise be put to better use. See Section 5.3, “Non-Temporal Data” for more information about non-temporal data.

Using non-temporal prefetches on the the non-temporal data can prevent the data from being cached in this cache level. This does not hurt performance since the data would have been evicted from the cache before being reused anyway, but may improve performance by leaving more cache space for other data that can be successfully cached, and for data of other threads and processes that are sharing the cache. See Section 5.3.5.1, “Non-Temporal Prefetches” for more information.

This issue type is normally only included when analyzing the highest cache level, that is, the cache level closest to memory, since non-temporal prefetches affect this cache level in most processors.


Issue #12: Non-temporal data

Statistics for the reuses of the non-temporal data

% fo fetches	6.5%
Fetch ratio	100.0%
Fetches	8.08e+05

Last instructions to touch the data before it is evicted

Stack	Instruction	% of non-temporal reuses	Required cache size
+	main() (0x40091a) [W], all.c:241	100.0%	712k

First instructions to touch the data after it is evicted

Stack	Instruction	% of non-temporal reuses
+	main() (0x40090b) [R], all.c:241	100.0%

Instruction group statistics

Instructions in instruction group

Loop statistics

Loop instructions

Figure 8.14. Non-Temporal Data Issue

The most important statistics reported in the non-temporal data issue are for the non-temporal reuses, that is, data reuses when the cache line has been evicted from the first level cache.

The non-temporal data issue has the following sections:

- Statistics for the reuses of the non-temporal data

  This section shows the fetch ratio and number of fetches of the reuses of the non-temporal data. When using non-temporal prefetches you want the fetch ratio to be as high as possible, so that you do not cause additional fetches by preventing caching of the data.

  The section also shows the percentage of all fetches of the application that are caused by these non-temporal reuses. The larger the percentage of all fetches, the greater the potential for performance improvement.

- Last instructions to touch the data before it is evicted

  This section lists the instructions that were last to touch the cache lines before they were evicted, and the percentage of the time each instruction was last.

  For each instruction you also get an estimate of the cache size that would be required for the fetch ratio the reuses where the instruction was last touch the data to fall below 80%.

  For example, if you have done the analysis for a 2 MB cache, and the non-temporal data requires a 200 MB cache not to be evicted you can safely insert non-temporal prefetches since no current processor has a cache that large. On the other hand, if the non-temporal data only requires 6 MB of cache to fit, inserting non-temporal prefetches may cause unnecessary fetches on processors that have 6 MB or larger caches.

- First instructions to touch the data after it is evicted

  This section lists the instructions that were first to touch the cache lines after they were evicted, and the percentage of the time each instruction was first.

- Instruction group statistics

  Statistics for the instruction group containing the last instructions to touch the data before it is evicted.

- Instructions in instruction group

  Instructions in the instruction group containing the last instructions to touch the data before it is evicted.

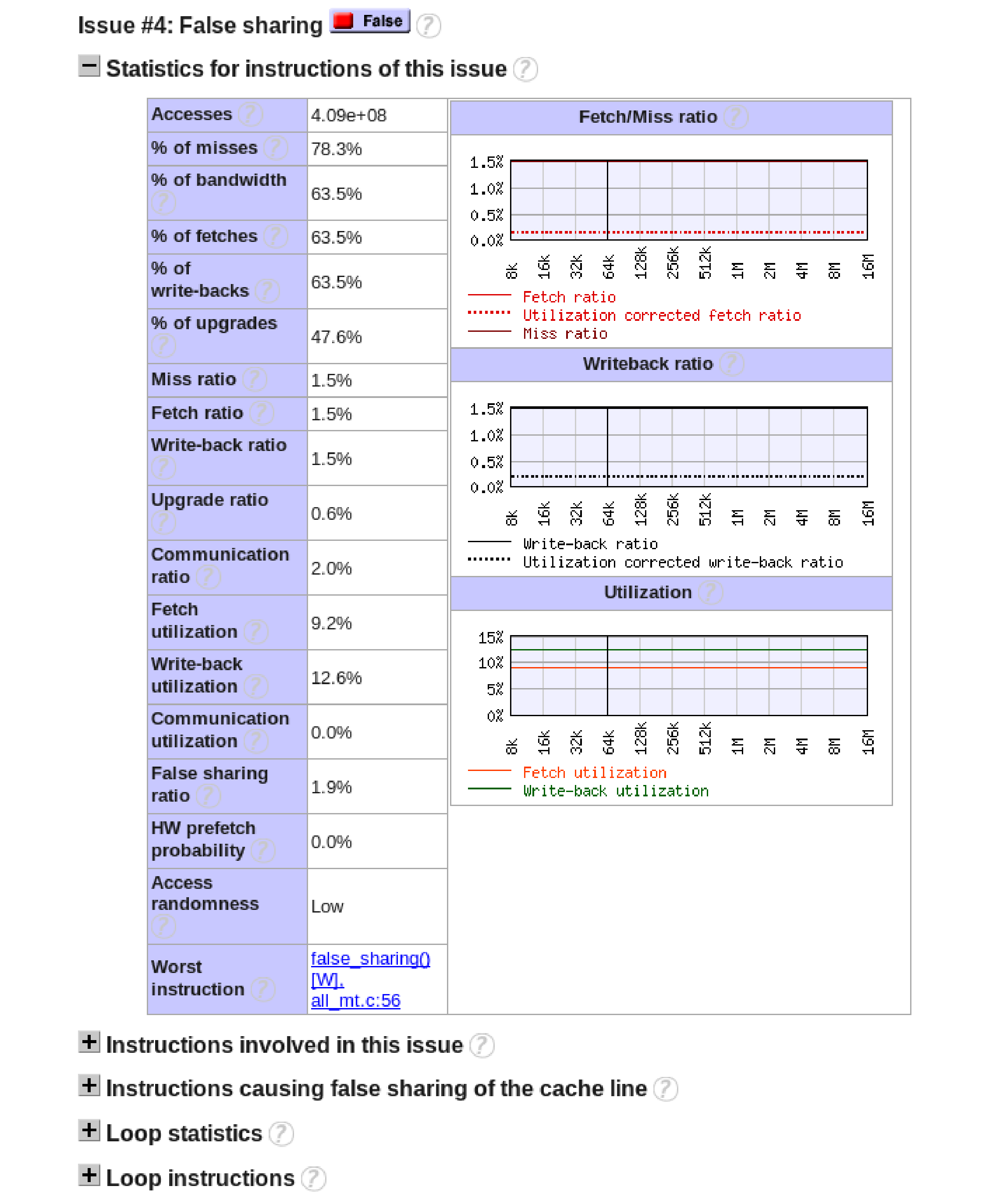
- Loop statistics

  Statistics for the loop containing the last instructions to touch the data before it is evicted.

- Loop instructions

  Instructions in the loop containing the last instructions to touch the data before it is evicted.

8.11. False Sharing



Issue #4: False sharing 🟥 False ?

− Statistics for instructions of this issue ?

Accesses ?	4.09e+08
% of misses ?	78.3%
% of bandwidth ?	63.5%
% of fetches ?	63.5%
% of write-backs ?	63.5%
% of upgrades ?	47.6%
Miss ratio ?	1.5%
Fetch ratio ?	1.5%
Write-back ratio ?	1.5%
Upgrade ratio ?	0.6%
Communication ratio ?	2.0%
Fetch utilization ?	9.2%
Write-back utilization ?	12.6%
Communication utilization ?	0.0%
False sharing ratio ?	1.9%
HW prefetch probability ?	0.0%
Access randomness ?	Low
Worst instruction ?	false_sharing() [W], all_mt.c:56

Fetch/Miss ratio ?
1.5%
1.0%
0.5%
0.0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch ratio
Utilization corrected fetch ratio
Miss ratio
Writeback ratio ?
1.5%
1.0%
0.5%
0.0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Write-back ratio
Utilization corrected write-back ratio
Utilization ?
15%
10%
5%
0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch utilization
Write-back utilization

+ Instructions involved in this issue ?

+ Instructions causing false sharing of the cache line ?

+ Loop statistics ?

+ Loop instructions ?

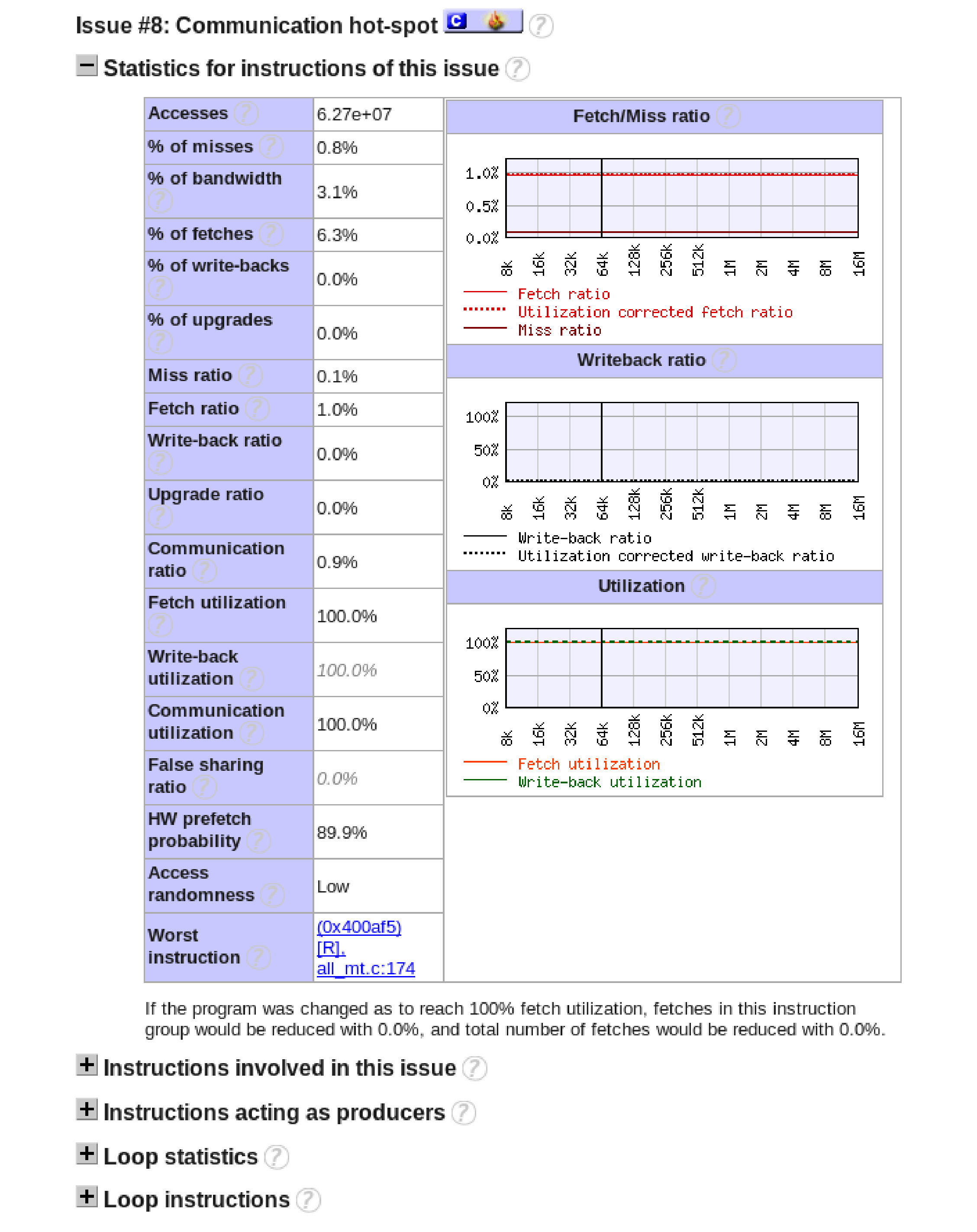
Figure 8.15. False Sharing Issue

A *false sharing* issue is reported when ThreadSpotter™ finds two locations in different threads, or rather in threads bound to different caches, that access unrelated data in the same cache line. This causes coherence misses, coherence write-backs or upgrades, which could be avoided with different placement of the data. See Section 5.4.1, “False Sharing” for a more thorough description of false sharing.

The false sharing issue has these sections:

- Statistics for instructions of this issue
- Instructions involved in this issue
- Instructions causing false sharing of the cache line. There are always pairs of instructions involved in a false sharing situation. This section lists the other instructions. Focus on separating the data accessed by the different threads, or arrange to divide data between the two threads along a cache line.
- Loop statistics
- Loop instructions

8.12. Communication Hot-Spot


Fetch/Miss ratio
1.0%
0.5%
0.0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch ratio
Utilization corrected fetch ratio
Miss ratio
Writeback ratio
100%
50%
0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Write-back ratio
Utilization corrected write-back ratio
Utilization
100%
50%
0%
8k 16k 32k 64k 128k 256k 512k 1M 2M 4M 8M 16M
Fetch utilization
Write-back utilization

Issue #8: Communication hot-spot

Statistics for instructions of this issue

Accesses	6.27e+07
% of misses	0.8%
% of bandwidth	3.1%
% of fetches	6.3%
% of write-backs	0.0%
% of upgrades	0.0%
Miss ratio	0.1%
Fetch ratio	1.0%
Write-back ratio	0.0%
Upgrade ratio	0.0%
Communication ratio	0.9%
Fetch utilization	100.0%
Write-back utilization	100.0%
Communication utilization	100.0%
False sharing ratio	0.0%
HW prefetch probability	89.9%
Access randomness	Low
Worst instruction	(0x400af5) [R], all_mt.c:174

If the program was changed as to reach 100% fetch utilization, fetches in this instruction group would be reduced with 0.0%, and total number of fetches would be reduced with 0.0%.

Instructions involved in this issue

Instructions acting as producers

Loop statistics

Loop instructions

Figure 8.16. Communication Hot-Spot Issue

A *communication hot-spot* issue is reported when ThreadSpotter™ finds two locations in different threads, or rather in threads bound to different caches, that communicate a lot. But, unlike the communication utilization issue described in Section 8.1.3, “Communication Utilization”, this issue indicates that these locations use the communicated cache lines efficiently.

There is, however, always a cost to communication between caches, so you should always try to minimize the communication. It might be possible to address this by using techniques similar to blocking, to avoid communicating data ever so often. Sometimes, it might make sense to critically evaluate if the producer and consumer needs to be in different threads.

In many cases communication has hidden costs, in that additional high level synchronization, or locking, is needed to ensure proper protocol between the communicating threads. Synchronization inevitably leads to serialization of the execution of the threads, which can be a limiting factor for performance. Whenever there is communication, ensure that proper synchronization is used, but that locks are held as short a time as possible.

Finally, carefully binding the two threads to cores sharing a cache can make a difference, since the data will not have to migrate at all as both the producer and the consumer share that cache. This also depends on the cache size and the footprint of each thread sharing this cache.

The communication hot-spot issue has these sections:

- Statistics for instructions of this issue
- Instructions involved in this issue
- Instructions acting as producers. These are the instructions in other threads that write the data that are being consumed by the instructions identified by this issue.
- Instructions previously writing to related data
- Loop statistics
- Loop instructions

Chapter 9. Technical Support

ParaTools ThreadSpotter is free software released under LGPL. If you have problems installing, or running your software and wish to purchase support, please contact ParaTools, Inc.

http://www.paratools.com

Please be prepared to provide the following information when you e-mail ParaTools:

- Name and version number of the product. For example, ThreadSpotter™ 2012.1.1
- The type of system on which the software is being run. For example, Opteron, Intel x86.
- The operating system and version number. For example, Red Hat Enterprise Linux 6 or Windows 7, and whether it is a 32 or 64 bit operating system.
- Your customer ID or entitlement code.
- A detailed description of the problem.

Table 9.1. Electronic Services

Service	**Address**
General e-mail	info@paratools.com
Support e-mail	threadspotter-support@paratools.com
World Wide Web	http://www.paratools.com/
ParaTools ThreadSpotter Software Forums	http://www.paratools.com/threadspotter

Appendix A. Sampling MPI Applications

A.1. Introduction

As systems become larger and applications are distributed, there is a need to sample and analyze MPI applications. For local applications, ThreadSpotter™ analyzes one process at a time. The same is true for distributed application; ThreadSpotter™ samples and analyzes each rank individually.

It is important, however, to analyze the correct process in the distributed environment. As shown in the figure below, it is possible to analyze the bootstrapping program **mpirun** rather than the distributed instances of the application. To enable ThreadSpotter™ to produce useful samples of the application, ThreadSpotter™ needs to be distributed along with the application. Use the command:

```
$ mpirun ... sample -r application
```

to launch the sampler on each node. Each sampler then launches and samples an instance of your application and creates a fingerprint file from the execution. Take care to assign unique filenames to each of these files.

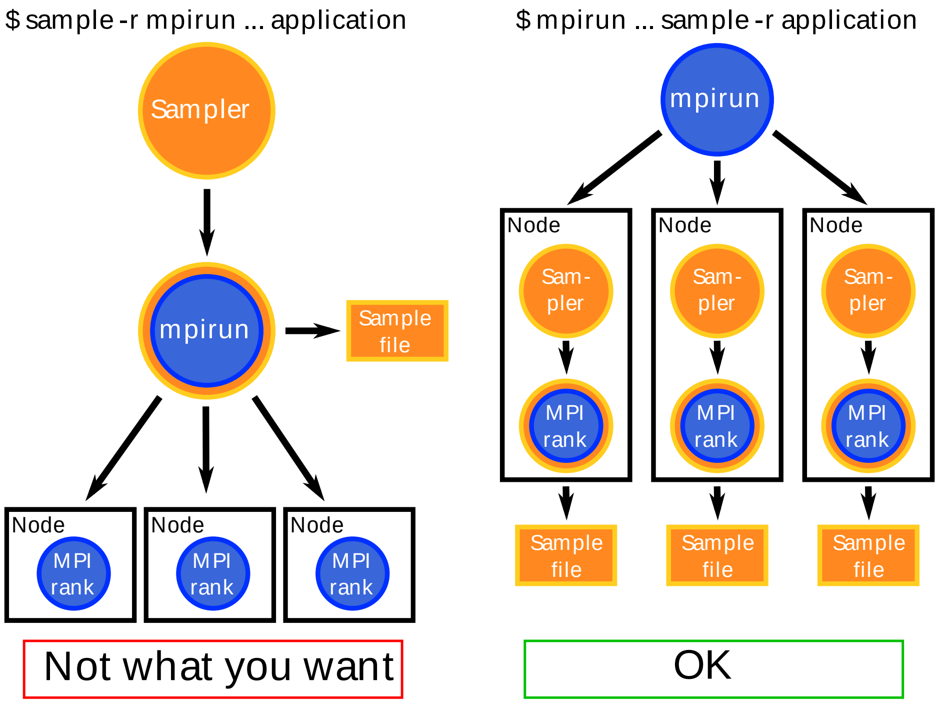


Figure A.1. MPI Sampling Principles

On some systems (e.g., SGI, Cray), and with some MPI variants (e.g., MPT), the MPI runtime system may optimize the process of launching several ranks per node and improve performance in communication between these nodes by making use of an extra *shepherd* process. In such case, it is necessary to tell ThreadSpotter™ to refrain from sampling the shepherd process by telling it which process generation to sample.

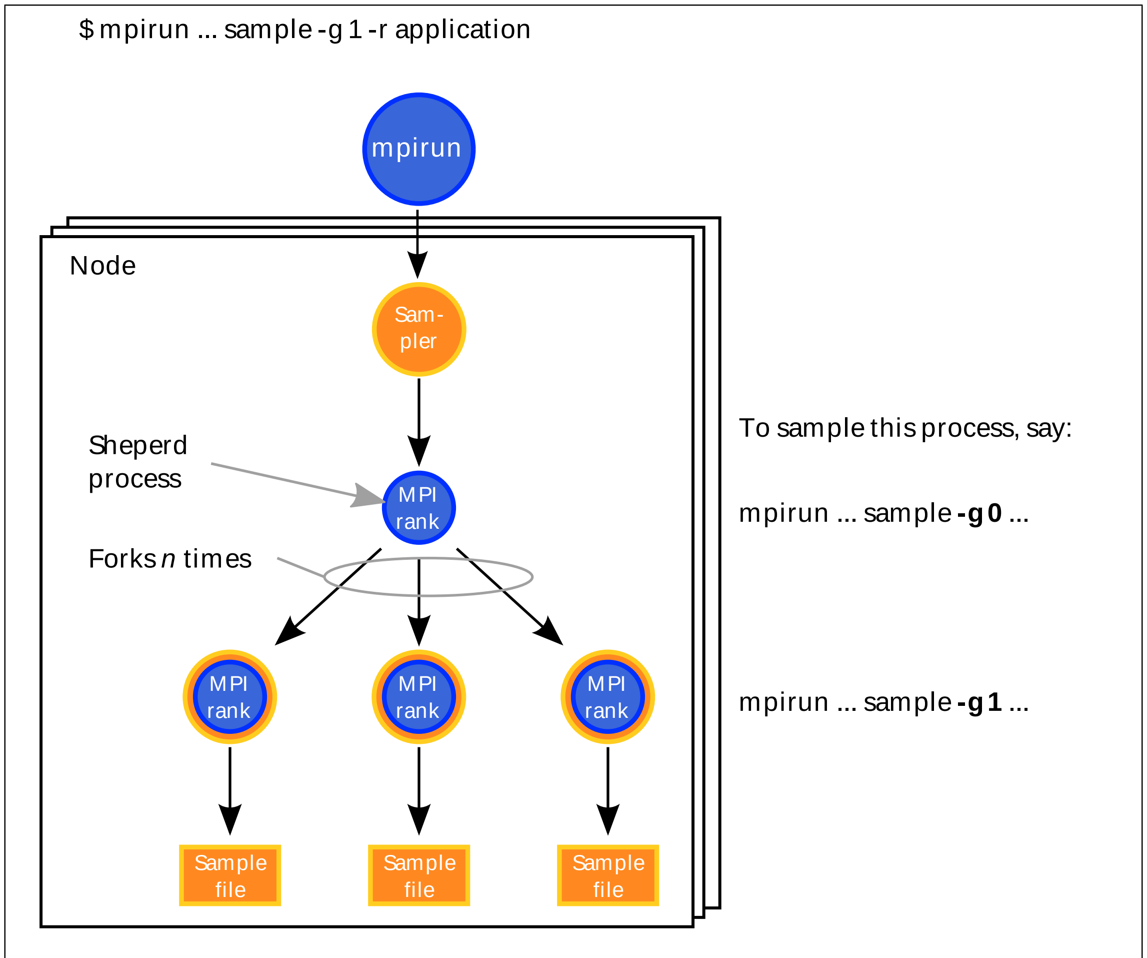

$ mpirun ... sample -g 1 -r application
mpirun
Node
Sam-
pler
Sherperd
process
MPI
rank
Forks *n* times
MPI
rank
MPI
rank
MPI
rank
Sample
file
Sample
file
Sample
file
To sample this process, say:
mpirun ... sample **-g 0** ...
mpirun ... sample **-g 1** ...

Figure A.2. Message Passing Toolkit, runtime system and shepherd process

Use **sample -g 1** to sample the children processes of a sheperd process.

Note

If only one rank is requested, the shepard process turns into a compute process and you should give the command **sample -g 0** instead.

A.2. Scope

As far as the ThreadSpotter™ sampler is concerned, an MPI instance (or rank) is just like any other application. There are two points to keep in mind:

- The MPI library appears like any other library. Its memory behavior is assessed together with the rest of the application. This means that ThreadSpotter™ looks at memory access patterns and complains about poor memory usage in relevant parts of the application, as well as in the MPI library.

  On platforms where several MPI ranks are hosted within a single machine, the MPI library may use shared memory as a mechanism to transfer data between co-located ranks. The analysis of ThreadSpotter™ does not extend into correctly classifying memory use from other processes. It considers each process in isolation.

- Some tools analyze the communication between MPI peers, including judging communication speed and patterns, node imbalances or general effectiveness of the actual MPI-communication.

  Such analysis is outside the scope of ThreadSpotter™, as it only looks at the behavior of each process in isolation.

  On the other hand, different processes in a job may work on data with different characteristics, and therefore have different runtime behavior. In some cases, processes exercise completely different part of the application code. The reports produced by ThreadSpotter™ for nodes with different behavior look different and point to optimizations that are important to their respective process.

A.3. Sampling of MPI Applications

When sampling a normal, non-MPI application, the **sample** command creates a fingerprint file called `sample.smp` by default, unless overridden by an explicit parameter like so:

```
$ sample -o myapp-test4.smp -r ./myapp arg1
```

In a MPI environment, there are several instances of the sampler that all try to create their own fingerprint files, typically in the same directory. Clearly, they need to use different names for their files, or else the files overwrite one another. It is your responsibility to provide a template for the names of these files as an input to the sample command:

```
$ mpirun -np 16 sample -o test4/myapp-%r.smp -r ./myapp arg1
```

The **%r** is a special sequence that expands to a unique value for each rank. On many systems it expands to the MPI rank number, but on systems where the rank number is not available to ThreadSpotter™, it expands to a combination of hostname and a unique number, e.g., `"node45-17"`, where `node45` is the name of the node, and `17` is a sequential number within that node.

The following table lists the special sequences that ThreadSpotter™ recognizes:

Table A.1. Fingerprint filename substitutions

Sequence	Substitution
%r	If executed in an MPI environment this is replaced with the **MPI rank** number, if known. On other systems it is equivalent to **%h-%p** (see below). Example: x-%r.smp becomes x-17.smp (MPI), or x-myhost-123.smp (otherwise)
%h	Replaced with the current hostname of the machine. Example: x-%h.smp becomes x-myhost.smp
%p	Replaced with a unique number within the current host. Note: It is not related to the Pid of the target process. Example: x-%p.smp becomes x-123.smp
%u	Expands to "*`.number`*" in such a way that the resulting filename becomes unique. Numbers are tried sequentially, starting with 0. As a special case, the number ".0" is elided. Example: x%u.smp becomes x.smp (first file), x.1.smp (next file) etc.
%U	Expands to "*`.number`*" in such a way that the resulting filename becomes unique. Numbers are tried sequentially until a unique number is found. Example: x%U.smp becomes x.0.smp (first file), x.1.smp (next file), etc.

Sequence	**Substitution**
%{*ENV*}	Replaced with the content of the environment variable `ENV` Example: x-%{USER}.smp becomes x-demo.smp
%%	Replaced with a single **`%`**. This is sometimes useful on Windows when encoding commands in a batch file. Example: x%%.smp becomes x%.smp

The following table lists the environment variables that ThreadSpotter™ tries in sequence to determine the MPI rank number.

Table A.2. %r substitutions

Environment variable[a]	**Used for MPI type:**
PMI_RANK	MPICH 2 and derivatives
OMPI_COMM_WORLD_RANK	OpenMPI 1.3
OMPI_MCA_ns_nds_vpid	OpenMPI 1.2 and derivatives
PMI_ID	SLURM PMI
SLURM_PROCID	SLURM
LAMRANK	LAM
MPI_RANKID	HP MPI for Linux
MP_CHILD	IBM PE
MP_RANK	Sun CT
MPIRUN_RANK	MVAPICH 1.1

[a] The inclusion of an environment variable in this list does not automatically imply that ThreadSpotter™ is certified for that platform.

A.4. Alternative method: wrapper scripts

If you require more complicated substitutions than the substitution mechanisms provide, or if different sampling options should apply depending on the rank, you have the option of providing a wrapper script that implements the required logic. The script can reference environment variables and call other utilities that exist on the host.

You must then invoke mpirun (mpiexec, srun, aprun etc.) on this wrapper script instead of the sample binary.

This technique relies on a few things to be successful:

- The cluster nodes must have a suitable script interpreter (e.g., `/bin/sh`).
- The binaries of ThreadSpotter™ need to be made available in each cluster node, either by referring to them with absolute paths or proper relative paths or by setting the `PATH` environment variable properly.
- The path to the target application needs to be properly set to locate it in the cluster node (with absolute path name or a proper relative path name).
- On some systems (e.g., Cray), the default application staging mechanism has to be disabled. It is not able to identify all the binaries that need to be staged. All applications (ThreadSpotter™, user applications) need to be visible from each cluster node.

In these examples it is assumed the fingerprint files are created in the working directory where **mpirun** starts the script. The exact capabilities and mechanisms vary with different MPI systems. On some systems you have to write your script to explicitly set the appropriate working directory or use explicit paths.

Example A.1. Sampling Open MPI ranks using a wrapper script

If you are using Open MPI and want to run the MPI application **mpi-test**, your wrapper script could look like the following example:

`sample_mpi_test.sh`:

```
#!/bin/sh
sample -o rank$OMPI_COMM_WORLD_RANK.smp -r mpi-test "$@"
```

Invoke this by:

```
$ mpirun -np ... sample_mpi_test.sh arg1 arg2
```

The script samples each rank of the MPI application mpi-test and creates a sample file named `rank0.smp` for rank 0, `rank1.smp` for rank 1, and so on

Example A.2. Sampling with a wrapper script

If your MPI implementation does not provide an environment variable that identifies the MPI rank, you can replace the rank in the file name with another unique identifier, for example, the host name and the PID of the script. The following script is an example:

`sample_mpi_test2.sh`:

```
#!/bin/sh
sample -o rank-$HOSTNAME-$$.smp -r mpi-test "$@"
```

Invoke this by:

```
$ mpirun -np 16 sample_mpi_test2.sh arg1 arg2
```

The script samples each rank of the MPI application mpi-test and creates a sample file named `rank-node42-4711.smp` for rank 0, `rank-node42-4718.smp` for rank 1, and so on.

Example A.3. Selectively sampling ranks

If the rank is available through an environment variable, you can also selectively sample ranks. For example, the following script would sample rank 0 and run the other ranks without sampling with Open MPI:

```
#!/bin/sh
if [ $OMPI_COMM_WORLD_RANK == 0 ]; then
  sample -o rank$OMPI_COMM_WORLD_RANK.smp -r mpi-test "$@"
else
  mpi-test "$@"
fi
```

A.5. Scratch directories

Compute clusters may have different environments on the front-end nodes, the service nodes and the compute nodes. It does not necessarily hold that the same file systems exist on all nodes or are mounted in the same location.

ThreadSpotter™ produces temporary files during sampling. These can become quite large for long executions, although the growth rate tapers off during the run. To impose a minimum disturbance on the network and I/O system, you need to ensure that these temporary files are stored as close to the node as possible.

To control where temporary files are stored, you should either point one of the environment variables `THREADSPOTTER_TMPDIR` or `TMPDIR` to a suitable directory or provide the directory name in a parameter to the **sample** command. The default is the system temporary directory, typically `/tmp`, which may not be suitable. On some systems this directory is in a memory mapped file system with limited capacity.

A.6. Cray, Torque PBS, and ALPS

The Cray Linux Environment sometime uses the Torque scheduler and the ALPS launcher for spawning jobs. In this environment, you typically write a batch script, and submit it with the **qsub** command. The script invokes the ALPS tool **aprun** to launch your binary across the cluster.

By default **aprun** optimizes the launch by pushing your supplied application binary to a ram disk on each compute node. This staging mechanism improves the launch time for normal runs.

When **aprun** launches the sampler from ThreadSpotter™, the primary sampler binary also enjoys this staging, but unfortunately ALPS does not know about the other ThreadSpotter™ binaries, or even your application, so these binaries are not staged. Consequently, execution fails when these applications cannot be located.

In the Cray environment, you must rewrite your batch script to invoke **aprun** with the **-b** flag. This inhibits the staging mechanism altogether, and the normal rules for launching applications are used again.

> The aprun command accepts the following options:
>
> `-b` Bypasses the transfer of the application executable to compute nodes. By default, the executable is transferred to the compute nodes as part of the aprun process of launching an application. You would likely use the -b option only if the executable to be launched was part of a file system accessible from the compute node. For more information, see the EXAMPLES section.
>
> —From: aprun(1) manual page

The proper way to launch a ThreadSpotter™ sampling in a Cray Linux Environment with Torque and ALPS is to create a batch file:

`my-job.pbs`:

```
#!/bin/bash
#PBS -N my-job
#PBS -l mppwidth=64
#PBS -l mppnppn=32
#PBS -l walltime=00:10:00

# set PATH to include the ThreadSpotter™ bin directory
PATH=$PATH:installation_directory/bin

# change directory where the job was submitted from
cd $PBS_O_WORKDIR
```

```
    aprun -b -n 64 -N 32 sample -g 1 -o my-job-samplefiles/process-%r.smp \
          -r ./my-job arg1 arg2
```

and invoke this script using:

```
$ qsub my-job.pbs
```

Sample files appear in the directory `$PBS_O_WORKDIR/my-job-samplefiles`.

If you only launch one rank, omit **-g 1**, or change it into **-g 0**

A.7. Cray, SLURM, and ALPS

An alternate scheduler in the Cray Linux Environment is SLURM. In this environment, you typically write a batch script and submit it with the **sbatch** command. The script invokes the ALPS tool **aprun** to launch your binary across the cluster.

The same considerations as outlined in Section A.6, “Cray, Torque PBS, and ALPS” apply to parameters for the **aprun** and **sample** command.

Note

Sample file replacement token %r does not expand to a meaningful value on this platform. Please use %h-%p instead.

The proper way to launch a ThreadSpotter™ sampling in a Cray Linux Environment with SLURM and ALPS is to create a batch file:

`my-job.sbatch`:

```
    #!/bin/bash -l
    #
    #SBATCH --job-name="my-job"
    #SBATCH --time=00:05:00
    #SBATCH --nodes=2
    #SBATCH --ntasks-per-node=32
    #SBATCH --mem-per-cpu=1024
    #SBATCH --output=my-job.%j.o
    #SBATCH --error=my-job.%j.e
    #======START=====
    module load slurm

    # set PATH to include the ThreadSpotter™ bin directory
    PATH=$PATH:installation_directory/bin

    echo "The current job ID is $SLURM_JOB_ID"
    echo "Running on $SLURM_NNODES nodes"
    echo "Using $SLURM_NTASKS_PER_NODE tasks per node"
    echo "A total of $SLURM_NPROCS tasks is used"
    aprun -b -n $SLURM_NPROCS sample -g 1 -o my-job-samplefiles/%h-%p.smp \
          -r ./my-job arg1 arg 2
    #=====END====
```

and invoke this script using:

```
$ sbatch my-job.sbatch
```

Sample files appear in directory `my-job-samplefiles`.

A.8. MPI related limitations

There are some limitations in the way ThreadSpotter™ supports MPI. Each rank is sampled separately, and no attempts are made to correlate or synchronize the sampling of different ranks.

Specific limitations include:

- Each rank contacts the license server separately, which for wide jobs could be a bottleneck or a hard limitation.
- Burst mode may not be effective to reduce overhead if ranks communicate with each other.
- Not all MPI versions make the rank number available as an environment variable. This means that ThreadSpotter™ will not be able to produce sample files with names that directly relate to the MPI rank number. Instead, some sequential numbering scheme is used.
- Some MPI environments (e.g., Cray) use optimizations to stage binaries before executing them. Such optimizations need to be disabled when sampling.
- Automatic processor detection at the time of report generation looks at the current processor in the machine where the report generation is performed. In a heterogenous system that processor type does not need to be identical to the processors in the compute nodes.

Appendix B. Cross-Architecture Analysis

B.1. Introduction

ThreadSpotter™ is equipped with a powerful cross-architecture analysis feature. While it can currently only sample applications compiled for the x86 and x86-64 architectures, its analysis is still perfectly applicable to help understanding performance problems in the context of other architectures, such as ARM and PowerPC.

The information that ThreadSpotter™ collects during sampling is related to the memory access patterns of the application. The memory access patterns of an application are usually very similar regardless of the processor architecture and the operating system it is running on.

If you can compile your application to run on x86, you can sample it on x86 and then perform the analysis for the processor architecture on which it normally runs. ThreadSpotter™ includes analysis models for a number non-x86 processors.

Similarly, if you have an application that normally runs on an x86 processor, but not on one of the operating systems supported by ThreadSpotter™, you can still analyze it if you can compile it for one of the supported operating systems.

B.2. Supported Non-x86 Processors

ThreadSpotter™ currently supports analysis for a number PowerPC and ARM processors. For these processors ThreadSpotter™ has an analysis model complete with all the essential information such as cache hierarchy layout, cache sizes and cache line sizes, the availability of prefetch instructions and their effect at different cache levels and whether hardware prefetching is available different at cache levels.

The supported PowerPC processors are:

- Freescale QorIQ P4080
- Freescale QorIQ P4040
- Freescale QorIQ P2020
- Freescale QorIQ P2010
- Freescale QorIQ P1020
- Freescale QorIQ P1011

The supported ARM processors are:

- ARM Cortex A8
- ARM Cortex A9
- ARM Cortex A9 MPCore Single-Core
- ARM Cortex A9 MPCore Dual-Core

- ARM Cortex A9 MPCore Quad-Core

The ARM Cortex processors can be configured with many different combinations of L1 and L2 cache sizes, and with or without L2 cache. The ThreadSpotter™ default models for these processors include a 16 kilobyte L1 data cache and a 128 kilobyte L2 cache.

If this does not match the configuration of the processor you are using, select the closest processor model and the desired cache level, and then specify a different cache size to override the default.

If your processor does not have an L2 cache cache, simply do not perform an L2 analysis. The L1 cache analysis done by ThreadSpotter™ is still valid.

B.3. Considerations for Accurate Cross-Architecture Analysis

To get the best possible results from cross-architecture analysis, there are a couple of things to consider:

- Use the same word size on both architectures.

  Compile the application with the same word size as it normally uses. If the application normally runs on a 32-bit processor, then compile it as a 32-bit x86 binary when sampling it. If the application normally runs on a 64-bit processor, then compile it as a 64-bit x86-64 binary when sampling it.

  Using the same word size on both architectures ensures that data types that depend on the word size, for example, pointers, are the same size on on both architectures. This minimizes the differences in the memory layout of the application's data structures, and ensures that the application's memory access patterns are affected as little as possible by the change of architecture.

- Use the same compiler and options for both architectures.

  If possible, use the same compiler and compilation options when compiling for both architectures. The compiler and compilation options affect the compiler optimizations that are performed on the application, which in turn affect the memory access patterns observed by ThreadSpotter™.

  For example, if different compilers or options are used, the compiler used when compiling for x86 may perform an optimization that fixes a performance problem in the application while the compiler normally used does not fix it. In that case ThreadSpotter™ will not observe the problem when sampling on x86 and not report an issue, even though the problem shows up on the architecture where the application is normally run.

  Similarly, the x86 compiler may not perform an optimization that the target compiler normally used performs. In that case ThreadSpotter™ may observe the problem and report an issue, even though fixing the problem is actually unnecessary because the target compiler normally used fixes it for you.

B.4. Sampling the Required Cache Line Size

By default ThreadSpotter™ only records information required for performing the analysis for 64-byte cache lines when sampling an application, 64 bytes being the cache line size of all modern x86 processors. Other architectures may use other cache line sizes, though, the most common are 32 bytes or 128 bytes.

If you want to perform the analysis for a processor that uses a different cache line size than 64 bytes, you must therefore make sure to select the desired cache line size when sampling. Otherwise you will get an error message saying "no samples found" when running the analysis.

B.5. x86-centric Issues

ThreadSpotter™ can generate a couple of issues that are slightly x86-centric. These issues will still be generated when analyzing a non-x86 architecture, but the corresponding solution will be different depending on architecture.

B.5.1. Non-Temporal Data

The non-temporal data issue, see Section 8.10, “Non-Temporal Data”, indicates that there is data that is not going to be reused that is taking up space in the cache that could be better used for other things. On x86 processors non-temporal prefetches can be used to tell the processor to completely evict the data from the cache hierarchy once it is evicted from the L1 cache, thus preventing the it from unnecessarily taking up cache space.

PowerPC and ARM processors do not have any instructions that directly correspond to a non-temporal prefetch, but a similar effect can be achieved by using cache line flushes (called *clean and invalidate* on ARM) to evict the cache lines from the cache once they are no longer needed.

However, accidentally flushing a cache line that will be reused will cause expensive extra cache misses, so cache line flushes have to be used carefully.

B.5.2. Non-Temporal Store Possible

ThreadSpotter™ reports a non-temporal store issue, see Section 8.9, “Non-Temporal Store Possible”, when piece of code completely overwrites entire cache lines. By using non-temporal stores to write to the cache lines, the application can tell the processor to avoid fetching such lines, avoiding the fetch latency and reducing the memory bandwidth used.

On PowerPC, allocating the cache line in the cache using the *dcba* or *dcbz* instructions before overwriting them can be used to achieve a similar effect.

B.6. Considerations for Specific Processors

The ARM Cortex processors contain a *preload engine* that software can use to explicitly load a memory region into the L2 cache. The x86 architecture does not have a corresponding feature, and ThreadSpotter™ has no model for it.

If your application does use the preload engine and you want to model the effect of using it when sampling on x86, insert a loop loading the data into the cache using one of the x86 prefetch instruction where the preload engine would otherwise be used. This will give a similar analysis result.

Appendix C. Supported CPU types

ThreadSpotter™ has a built-in database with CPU model parameters. The following tables list the processors currently included in this database.

The model names generally follow a pattern of: *`codename_cores_thread_cache-size`*. In some cases, some components have been omitted if there is no ambiguity.

Table C.1. AMD

Processor	**Command line interface: use '--cpu amd/*', where * is one of:**
Agena	agena
Albany	albany_128, albany_256
Athens	athens
Barcelona	barcelona
Brisbane	brisbane_1024, brisbane_512
Budapest	budapest
Callisto	callisto, callisto_2_6144
Caspian	caspian_1_512, caspian_2_1024, caspian_2_2048
Champlain	champlain_2_1024, champlain_2_512, champlain_3_512, champlain_4_512
Clawhammer	clawhammer_s754_1024, clawhammer_s754_512, clawhammer_s939_1024, clawhammer_s939_512
Conesus	conesus_1_256, conesus_2_1024, conesus_2_512
Deneb	deneb_4096, deneb_6144
Denmark	denmark
Dublin	dublin_128, dublin_256
Egypt	egypt
Geneva	geneva_1_1024, geneva_1_512, geneva_2_1024
Georgetown	georgetown_128, georgetown_256
Griffin	griffin_1024, griffin_1_512, griffin_2048
Heka	heka
Interlagos	interlagos_12_16384, interlagos_16_16384, interlagos_4_16384, interlagos_8_16384
Istanbul	istanbul
Italy	italy
Keene	keene_256, keene_512
Kuma	kuma
Lancaster	lancaster_1024, lancaster_512
Lima	lima_model111, lima_model127
Lisbon	lisbon_6
Llano	llano_2_1024, llano_2_512, llano_3_1024, llano_4_1024
Magny-Cours	magny_cours_12, magny_cours_8

Processor	**Command line interface: use '--cpu amd/*', where * is one of:**
Manchester	manchester_s939_512, manchester_s939_x2_1024, manchester_s939_x2_512
Manila	manila_4f_128, manila_4f_256, manila_5f_128, manila_5f_256
Neo	neo_111, neo_127
Newark	newark
Newcastle	newcastle_s754_256, newcastle_s754_512, newcastle_s939_512
Oakville	oakville
Odessa	odessa
Ontario	ontario_1_512, ontario_2_512
Orleans	orleans_f2, orleans_f3
Palermo	palermo_s754_d0_128, palermo_s754_d0_256, palermo_s754_e_128, palermo_s754_e_256, palermo_s939_128, palermo_s939_256
Propus	propus
Rana	rana
Regor	regor_1024, regor_2048
Richmond	richmond
Roma	roma_128, roma_256
Sable	sable
San Diego	san_diego_s939_1024, san_diego_s939_512
Santa Ana	santa_ana
Santa Rosa	santa_rosa
Sargas	sargas
Shanghai	shanghai
Sherman	sherman_256, sherman_512
Sledgehammer	sledgehammer
Sonora	sonora_128, sonora_256
Sparta	sparta_256, sparta_512
Suzuka	suzuka
Taylor	taylor
Thuban	thuban_6_6144
Toledo	toledo_X2_s939_1024, toledo_s939_1024, toledo_x2_s939_2048
Toliman	toliman
Trinidad	trinidad
Troy	troy
Tyler	tyler_1024, tyler_512
Valencia	valencia_4_8192, valencia_6_8192, valencia_8_8192
Venice	venice_s754_512, venice_s939_512
Venus	venus_model39, venus_model55
Winchester	winchester_s939_512

Processor	Command line interface: use '--cpu amd/*', where * is one of:
Windsor	windsor_67_1024, windsor_67_2048, windsor_67_512, windsor_75_1024, windsor_75_2048, windsor_75_512
Zacate	zacate_1_512, zacate_2_512
Zambezi	zambezi_4_4096, zambezi_4_8192, zambezi_6_8192, zambezi_8_8192
Zosma	zosma_4_6144
Zurich	zurich_4_4096, zurich_8_8192

Table C.2. ARM

Processor	Command line interface: use '--cpu arm/*', where * is one of:
Cortex A8	cortex_a8
Cortex A9	cortex_a9, cortex_a9_mpcore_1, cortex_a9_mpcore_2, cortex_a9_mpcore_4

Table C.3. Freescale

Processor	Command line interface: use '--cpu freescale/*', where * is one of:
QorIQ P1	qoriq_p1011, qoriq_p1020
QorIQ P2	qoriq_p2010, qoriq_p2020
QorIQ P4	qoriq_p4040, qoriq_p4080

Table C.4. IBM

Processor	Command line interface: use '--cpu ibm/*', where * is one of:
BlueGene/L	bgl
BlueGene/P	bgp
BlueGene/Q	bgq
POWER6	power6
POWER7	power7_4, power7_6, power7_8
Wire-Speed	wirespeed

Table C.5. Intel

Processor	Command line interface: use '--cpu intel/*', where * is one of:
	silverthorne_1_2_512
Arrandale	arrandale_2_2_2048, arrandale_2_2_3072, arrandale_2_4_3072, arrandale_2_4_4096
Bloomfield	bloomfield_4_8_8192
Clarkdale	clarkdale_2_2_3072, clarkdale_2_4_4096
Clarksfield	clarksfield_4_8_6144, clarksfield_4_8_8192
Clovertown	clovertown_4_8192
Conroe	conroe_1_512, conroe_2_1024, conroe_2_2048, conroe_2_4096, conroe_2_512
Dempsey	dempsey_2_2_4, dempsey_2_4
Diamondville	diamondville_1_2_512, diamondville_2_4_1024

Processor	**Command line interface: use '--cpu intel/*', where * is one of:**
Dunnington	dunnington_4_12288, dunnington_4_16384, dunnington_4_8192, dunnington_6_12288, dunnington_6_16384
Gulftown	gulftown_6_12_12288
Harpertown	harpertown_4_12288
Ivy Bridge	ivy_bridge_2_4_3072, ivy_bridge_2_4_4096, ivy_bridge_2_4_6144, ivy_bridge_4_4_6144, ivy_bridge_4_4_8192, ivy_bridge_4_8_6144, ivy_bridge_4_8_8192
Jasper Forest	jasper_forest_1_1_2048, jasper_forest_1_2_2048, jasper_forest_2_2_4096, jasper_forest_2_4_4096, jasper_forest_4_4_8096, jasper_forest_4_8_8096
Kentsfield	kentsfield_4_8192
Lincroft	lincroft_1_2_512
Lynnfield	lynnfield_4_4_8192, lynnfield_4_8_8192
Merom	merom_1_1024, merom_1_512, merom_2_1024, merom_2_2048, merom_2_4096
Nehalem EP	nehalem_ep_2_2_4096, nehalem_ep_2_4_8192, nehalem_ep_4_4 4096, nehalem_ep_4_8_8192
Nehalem EX	nehalem_ex_4_8_12288, nehalem_ex_4_8_18432, nehalem_ex_6_12_12288, nehalem_ex_6_12_18432, nehalem_ex_6_6_18432, nehalem_ex_8_16_18432, nehalem_ex_8_16_24576
Paxville	paxville_2_2, paxville_2_4
Penryn	penryn_1_1024, penryn_1_3072, penryn_2_1024, penryn_2_2048, penryn_2_3072, penryn_2_6144, penryn_4_12288, penryn_4_6144
Pineview	pineview_1_2_512, pineview_2_4_1024
Presler	presler_2_4
Presler HT	presler-ht_2_4
Sandy Bridge	sandy_bridge_1_1_1024, sandy_bridge_1_1_1536, sandy_bridge_2_2_2048, sandy_bridge_2_2_3072, sandy_bridge_2_4_3072, sandy_bridge_2_4_4096, sandy_bridge_4_4_6144, sandy_bridge_4_4_8192, sandy_bridge_4_8_6144, sandy_bridge_4_8_8192
Sandy Bridge E	sandy_bridge_e_4_8_10240, sandy_bridge_e_6_12_12288, sandy_bridge_e_6_12_15360
Sandy Bridge EP	sandy_bridge_ep_2_4_5120, sandy_bridge_ep_4_4_10240, sandy_bridge_ep_4_8_10240, sandy_bridge_ep_6_12_12288, sandy_bridge_ep_6_12_15360, sandy_bridge_ep_8_16_20480
Silverthorne	silverthorne_1_1_512
Smithfield	smithfield_2_2
Stellarton	stellarton_1_2_512
Tigerton	tigerton_2_8196, tigerton_4_4096, tigerton_4_6144, tigerton_4_8192
Tulsa	tulsa_2_2_16, tulsa_2_2_8
Tunnel Creek	tunnel_creek_1_2_512
Westmere EP	westmere_ep_4_4_12288, westmere_ep_4_4_4096, westmere_ep_4_4_8192, westmere_ep_4_8_12288, westmere_ep_6_12_12288

Processor	**Command line interface: use '--cpu intel/*', where * is one of:**
Westmere EX	westmere_ex_10_20_24576, westmere_ex_10_20_30720, westmere_ex_6_12_18432, westmere_ex_8_16_18432, westmere_ex_8_16_24576, westmere_ex_8_8_24576
Wolfdale	wolfdale_2_1024, wolfdale_2_2048, wolfdale_2_3072, wolfdale_2_6144
Woodcrest	woodcrest_2_4096
Yorkfield	yorkfield_4_12288, yorkfield_4_4096, yorkfield_4_6144

Appendix D. Credits

D.1. libelf

GNU Library General Public License, version 2

Copyright (C) 1995 - 2006 Michael Riepe

D.2. libdwarf

GNU Lesser General Public License, version 2.1

Portions copyright (C) 2000,2004 Silicon Graphics, Inc. All Rights Reserved.

Portions copyright (C) 2007 David Anderson.

D.3. libgd-2.0.34

Portions copyright 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002 by Cold Spring Harbor Laboratory. Funded under Grant P41-RR02188 by the National Institutes of Health.

Portions copyright 1996, 1997, 1998, 1999, 2000, 2001, 2002 by Boutell.Com, Inc.

Portions relating to GD2 format copyright 1999, 2000, 2001, 2002 Philip Warner.

Portions relating to PNG copyright 1999, 2000, 2001, 2002 Greg Roelofs.

Portions relating to gdttf.c copyright 1999, 2000, 2001, 2002 John Ellson (ellson@lucent.com).

Portions relating to gdft.c copyright 2001, 2002 John Ellson (ellson@lucent.com).

Portions copyright 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007 Pierre-Alain Joye (pierre@libgd.org).

Portions relating to JPEG and to color quantization copyright 2000, 2001, 2002, Doug Becker and copyright (C) 1994, 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, Thomas G. Lane. This software is based in part on the work of the Independent JPEG Group. See the file README-JPEG.TXT for more information.

Portions relating to WBMP copyright 2000, 2001, 2002 Maurice Szmurlo and Johan Van den Brande.

Permission has been granted to copy, distribute and modify gd in any context without fee, including a commercial application, provided that this notice is present in user-accessible supporting documentation.

This does not affect your ownership of the derived work itself, and the intent is to assure proper credit for the authors of gd, not to interfere with your productive use of gd. If you have questions, ask. "Derived works" includes all programs that utilize the library. Credit must be given in user-accessible documentation.

This software is provided "AS IS." The copyright holders disclaim all warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to this code and accompanying documentation.

Although their code does not appear in gd, the authors wish to thank David Koblas, David Rowley, and Hutchison Avenue Software Corporation for their prior contributions.

D.4. OpenSSL

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)

D.5. klibc

klibc version 1.4 by H. Peter Anvin

Command Reference

Table of Contents

Name

threadspotter — GUI for ThreadSpotter™ sampling and report generation

Synopsis

`threadspotter` [-h, --help] [--version] [--tmpdir *`TMPDIR`*] [-- *`PROGRAM`* [*`ARGUMENT`*...]]

Description

threadspotter is a simple graphical user interface for sampling applications and generating reports with ThreadSpotter™.

The same graphical user interface can be used to:

- Start an application and sample it, optionally create a report and optionally display the report.
- Attach to an already running application, sample it, optionally create a report and optionally display the report, or
- Generate a report from an existing sample file, and optionally display the new report.
- Display and existing report.

The program allows setting the same parameters as the corresponding command line programs, **sample**, **report** and **view**.

The target application is executed in a new 'xterm' window. After completing the sampling, please close the xterm window.

Options

`-h, --help`	Print help message.
`--version`	Print version information.
`--tmpdir` *`TMPDIR`*	Name the directory where all temporary files will be stored. This directory can alternatively be specified by environment variables `THREADSPOTTER_TMPDIR` or `TMPDIR`. The command line option takes precendence over the environment variables.
`PROGRAM`	Use PROGRAM as the default program to launch.
`ARGUMENT``...`	Use ARGUMENTs as the default arguments when launching a program.

Exit Status

0	Successful program execution.
>0	An error occured.

Environment

`RW_LICENSE_FILE`	Environment variable pointing to the license file. Should only be used to override default license locations.

`BROWSER`	Web browser to use to display the report.

Files

`$HOME/.threadspotter/license`	Directory containing per-user license files. ThreadSpotter looks for license files here if no system wide license file can be found.

Name

sample — sample the memory access pattern of a process and generate a sample file

Synopsis

`sample` [-h] [--version] [-o *`FILENAME`*] [-s *`PERIOD`*] [-l *`BYTES`* [,*`BYTES`*...]] [-n] [-f] [-d *`DELAY`*]
[-t *`DURATION`*] [-c *`COUNT`*]
[-b *`RUNTIME`*] [-q *`QUALITY`*] [--safe-stack] [--tmpdir *`TMPDIR`*] [--preallocation *`SIZE`*]
[--use-target-stderr] [--unsupported-attach] [-g *`GENERATION`*]
[{ --start-at-function *`FUNCTION`* | --start-at-address *`ADDRESS`* } [--start-at-ignore *`COUNT`*]]
[{ --stop-at-function *`FUNCTION`* | --stop-at-address *`ADDRESS`* } [--stop-at-ignore *`COUNT`*]]
{ -p *`PID`* | -r *`BINARY`* [*`ARGUMENT`*...] }

Description

`sample` is the tool that starts and monitors the fingerprinting operation of the target application. It can either start a new target process, or it can attach to an already running process. After the execution is complete, or after the sample tool has detached from the target process, it will post-process the fingerprint information and optionally reduce the fingerprint volume to a standard size.

Options

`-h`
: Show help message and exit.

`--version`
: Print version information.

`-o` *`FILENAME`*
: Sample file name, sample.smp by default.

`FILENAME` can contain certain substitution sequences, which are useful to automatically derive unique filenames. This feature is particularly useful when sampling parallel jobs (e.g., MPI) to create unique filenames based on properties of each node.

The following table lists the special sequences that ThreadSpotter™ recognizes:

Table 9. Filename substitutions

Sequence	Substitution
%r	If executed in an MPI environment this is replaced with the **MPI rank** number, if known. On other systems it is equivalent to **%h-%p** (see below). Example: x-%r.smp becomes x-17.smp (MPI), or x-myhost-123.smp (otherwise)
%h	Replaced with the current hostname of the machine. Example: x-%h.smp becomes x-myhost.smp
%p	Replaced with a unique number within the current host. Note: It is not related to the Pid of the target process. Example: x-%p.smp becomes x-123.smp

Sequence	Substitution
%u	Expands to "*.number*" in such a way that the resulting filename becomes unique. Numbers are tried sequentially, starting with 0. As a special case, the number ".0" is elided. Example: x%u.smp becomes x.smp (first file), x.1.smp (next file) etc.
%U	Expands to "*.number*" in such a way that the resulting filename becomes unique. Numbers are tried sequentially until a unique number is found. Example: x%U.smp becomes x.0.smp (first file), x.1.smp (next file), etc.
%{*ENV*}	Replaced with the content of the environment variable `ENV` Example: x-%{USER}.smp becomes x-demo.smp
%%	Replaced with a single **%**. This is sometimes useful on Windows when encoding commands in a batch file. Example: x%%.smp becomes x%.smp

The following table lists the environment variables that ThreadSpotter™ tries in sequence to figure out the MPI rank number for the %r substitution.

Table 10. %r substitutions

Environment variable[a]	Used for MPI type:
PMI_RANK	MPICH 2 and derivatives
OMPI_COMM_WORLD_RANK	OpenMPI 1.3
OMPI_MCA_ns_nds_vpid	OpenMPI 1.2 and derivatives
PMI_ID	SLURM PMI
SLURM_PROCID	SLURM
LAMRANK	LAM
MPI_RANKID	HP MPI for Linux
MP_CHILD	IBM PE
MP_RANK	Sun CT
MPIRUN_RANK	MVAPICH 1.1

[a] The inclusion of an environment variable in this list does not automatically imply that ThreadSpotter™ is certified for that platform.

`-s` *`PERIOD`*	Sample period in number of memory accesses, default 100.
`-l` *`BYTES`* `[,`*`BYTES`*`...]`	Line sizes to sample in bytes. Default is 64 byte line size.
`-n`	Don't decimate the sample file.

`-f`	Use fixed sample period. Default is to use automatic sample period adjustment.
`-d` *`DELAY`*	Start sampling after DELAY seconds.
`-t` *`DURATION`*	Detach after sampling for DURATION seconds.
`-c` *`COUNT`*	Collect COUNT samples. Default is 50000.
`-b` *`RUNTIME`*	Enable burst sampling. RUNTIME should be the estimated run time of the application in minutes.
`-q` *`QUALITY`*	Adjusts the quality parameter for bursting. Default quality level is 'normal'. fast — Decreases sampling overhead by sacrificing accuracy. Accuracy should normally be good for caches of 8 MB or smaller, and is generally acceptable for 16 MB caches. normal — The normal accuracy level, which is default, provides a good compromise between accuracy and performance. detailed — Increases sampling overhead by roughly a factor 2. Improves accuracy particularly when analyzing applications with small fetch ratios running on systems with large caches (32 MB or larger). This setting also allows more long range blocking and fusion opportunities to be found.
`--safe-stack`	Disable the sampler's use of the application stack. May be needed for applications with non-standard stack handling.
`--tmpdir` *`TMPDIR`*	Name the directory where all temporary files will be stored. This directory can alternatively be specified by environment variables `THREADSPOTTER_TMPDIR` or `TMPDIR` on Linux or `TMP` on Windows. The command line option takes precendence over the environment variables.
`--preallocation` *`SIZE`*	Reduces the amount of virtual address space reserved to the sampler to SIZE megabytes. This is only applicable to 64-bit installations and only needed on systems enforcing virtual memory limits. Minimum preallocation size is 128 MB, default is 2048 MB.
`--use-target-stderr`	Allow the sampler to log warnings and errors on the target application's stderr. Such messages are normally shown on the sampler control application's console instead.
`--unsupported-attach`	Relaxes the sampler permission checks to allow attaching to other users' processes when running as root. Use of this option may cause security issues, particularly on systems with multiple simultaneous users, and is not recommended.
`-g` *`GENERATION`*	Whether to sample the invoked process (=0, default) or its children (=1) etc. In some MPI environments, like SGI and Cray, the target

	application is bootstrapped using an intermediate process which should not be sampled.
`--start-at-function` *`FUNCTION`*	Start sampling when the execution reaches the specified function.
`--start-at-address` *`ADDRESS`*	Start sampling when the execution reaches the specified address.
`--start-at-ignore` *`COUNT`*	Ignore the first COUNT times the sampling start location (set with **--start-at-function** or **--start-at-address**) is reached.
`--stop-at-function` *`FUNCTION`*	Stop sampling when the execution reaches the specified function.
`--stop-at-address` *`ADDRESS`*	Stop sampling when the execution reaches the specified address.
`--stop-at-ignore` *`COUNT`*	Ignore the first COUNT times the sampling stop location (set with **--stop-at-function** or **--stop-at-address**) is reached.
`-p` *`PID`*	Sample the already running process with the specified pid.
`-r` *`BINARY`* `[`*`ARGUMENT`*`...]`	Start BINARY with ARGUMENTs.

Examples

Example 4. Starting an application in the sampler

Acquire fingerprint information from target process 'ls', run with '-l' command line parameter:

```
$ sample -r ls -l
```

Example 5. Attaching to a running process

Acquire fingerprint information from target process with pid 3344 and naming the output file:

```
$ sample -p 3344 -o fingerprint.smp
```

Example 6. Burst sampling a long running application

Long running applications generally benefit from burst sampling. The following example will burst sample a hypothetical benchmark that is normally expected to run for 15 minutes:

```
$ sample -b 15 -r ./benchmark
```

Example 7. Using a template name for output file

Sampling in an MPI environment requires that sample file names are unique. The following example will launch a number of instances of the sampler across an MPI environment, assigning a unique name to each sample file.

```
$ mpirun -np 16 sample -o experiment1/file-%r.smp -r ./my-mpi-application
```

Exit Status

0	Successful program execution.
1	Usage error.
2	Sampling failed.
3	Sample file post processing failed.

Environment

`RW_LICENSE_FILE`	Environment variable pointing to the license file. Should only be used to override default license locations.

Files

`$HOME/.threadspotter/license`	Directory containing per-user license files. ThreadSpotter looks for license files here if no system wide license file can be found.

Name

report — generate a report from a sample file

Synopsis

`report` [-h, --help] [--version] [--verbose] -i, --in-file *`FILENAME`* [-t, --title *`TITLE`*] [-o, --report *`NAME`*] [-s, --source-dir *`DIRECTORY`*...] [-D, --debug-dir *`DIRECTORY`*...] [-b, --binary *`BINARY`*...] [--debug-symbol-level *`LEVEL`*] [--c++filt*`[=FILTER]`*]
[-p, --percent *`PERCENT`*] [-d, --depth *`FRAMES`*] [--new-hwpf-algorithm]
[--cpu *`CPU`*] [--number-of-cpus *`NUM`*] [--level *`LEVEL`*]
[-c, --cache-size *`SIZE`*] [-l, --line-size *`SIZE`*] [-r, --replacement *`POLICY`*] [-n, --number-of-caches *`NUM`*]

Description

report generates a report based on an existing sample file.

Options

`-h, --help`	Print help message.
`--version`	Print version information.
`--verbose`	Show which filenames are tried when searching for debug information and source code.
`-i, --in-file` *`FILENAME`*	Specifies the input file.
`-t, --title` *`TITLE`*	Title of report.
`-o, --report` *`NAME`*	Name of the generated report file, defaults to `report.tsr`.
`-s, --source-dir` *`DIRECTORY`*	Additional directories to look for source code in.
`-D, --debug-dir` *`DIRECTORY`*	Additional directories to look for external debug information in. The report tool will by default look in the system global debug directory (`/usr/lib/debug`), the `.debug` directory in the same directory as the binary and the same directory as the binary.
`-b, --binary` *`BINARY`*	Specifies an additional binary containing debug information to use if the sampled binary with the same file name can not be found.
`--debug-symbol-level`	(experimental) Balance debug symbol detail and processing speed.

0	no debug symbols
1	line number
2	line number and public symbols
3	full debug info (default)

`--c++-filt[=FILTER]`	Specify an external symbol demangler program to translate the symbols for presentation. Useful for c++ code compiled with the 'stabs' debugging format. If `--c++filt` is given without a program, the external program **c++filt** is used. If `--c++filt=FILTER` is specified, use the program **FILTER**. Default is to not translate symbols.
`-p, --percent PERCENT`	Percent of total fetches, upgrades or write-backs required for advice to be reported. The default is 1.
`-d, --depth FRAMES`	Stack depth to use for separating issues caused by different calls to the same function. Default 1. Use 0 to merge all different call paths into a function for analysis.
`--new-hwpf-algorithm`	Use a new experimental version of the hardware prefetch analysis algorithm, that more diligently captures complex patterns. It consumes more memory and takes longer for some input sets than the default algorithm.
`--cpu CPU`	Selects the processor model to use in the analysis. *`CPU`* is specified as *`vendor-id`*/*`cpu-id`*. Default is to 'auto'. The following special processor models are defined: help — Lists available processor models. auto — Auto-detects the processor model of the computer the report is being generated on.
`--number-of-cpus NUM`	Number of physical processors to include in the analysis. Each physical processor may have multiple logical processors (cores/threads). The special value '0' may be used to indicate that auto-detection should be used, which is also the default.
`--level LEVEL`	Selects the cache level to analyze. The number of available cache levels depend on the selected processor model. Default is to analyze the highest cache level.
`-c, --cache-size SIZE`	Overrides the cache size specified in the processor model.
`-l, --line-size SIZE`	Overrides the cache line size specified in the processor model. Must be power of two.
`-r, --replacement POLICY`	Cache replacement policy. Must be 'random' or 'lru'. The default is 'random'.
`-n, --number-of-caches NUM`	Total number of caches to assign threads to. Should match the number of caches of the desired cache level for the intended processor/architecture. Default: Determined by the processor model, cache level and the number of physical processors. The special value '0' may be used to assign one private cache to each thread in the application.

Examples

Example 8. Analyzing sample files using autodetected CPU models

Perform an analysis of `sample.smp` for the currently running processor on cache level 2:

```
$ report --level 2 -i sample.smp
```

The report tool will create a report file named `report.tsr` by default.

Example 9. Specifying a CPU model

If you are running a different processor than you are analyzing for, you may specify the **--cpu *model*** option.

First, use **--cpu help** to get a list of available CPUs.

```
$ report --cpu help
```

Find the processor you want to perform analysis for, for example the Intel Quad-Core Xeon E5345 which has the model name 'clowertown_4_8'.

Use the manufacturer name together with the model name like this when calling **report**:

```
$ report --cpu intel/clovertown_4_8 -i sample.smp
```

Exit Status

0	Successful program execution.
>0	An error occured.

Environment

`RW_LICENSE_FILE`	Environment variable pointing to the license file. Should only be used to override default license locations.

Files

`$HOME/.threadspotter/license`	Directory containing per-user license files. ThreadSpotter looks for license files here if no system wide license file can be found.

Name

view — start a report viewer

Synopsis

`view` [-h, --help] [-version] -i, --in-file *`FILENAME`* [--port *`PORT`*] [--timeout *`SECONDS`*] [--no-browser]

Description

view diplays an existing report. It runs as a web server serving the report as web pages on a port on the local host. By default it also starts a web browser displaying the report.

view will try to start the default web browser on the system. If you want it to use a different browser you can specify it using the `BROWSER` environment variable.

Options

`-h, --help`	Print help message.
`--version`	Print version information.
`-i, --in-file` *`FILENAME`*	Specifies the input report file.
`--port` *`PORT`*	Launch the web server on the specified port instead of selecting an arbitrary free port.
`--timeout` *`SECONDS`*	Terminate the web server when no activity has been seen from the web browser viewing the report for the specified number of seconds. The default is 10 seconds. Try increasing the timeout if the web server quits unexpectedly while you are viewing the report.
`--no-browser`	Do not launch a web browser viewing the report, just start the web server serving it.

Exit Status

0	Successful program execution.
>0	An error occured.

Environment

`RW_LICENSE_FILE`	Environment variable pointing to the license file. Should only be used to override default license locations.
`BROWSER`	Web browser to use to display the report.

Files

`$HOME/.threadspotter/license`	Directory containing per-user license files. ThreadSpotter looks for license files here if no system wide license file can be found.

Name

license — Install a license file

Synopsis

`license` [-h] [--version] {[--file *`FILENAME`*] [[--server *`port@host`*]...]} [--system]

Description

`license` is the tool that is used to install licenses for use by ThreadSpotter™.

Options

`-h`	Show help message and exit.
`--version`	Print version information.
`--file` *`FILENAME`*	Install the named license file. This can either be a node locked license file for the current host, or a license file for a server. The license file will be installed for the user, unless `[--system]` is specified.
`[[--server` *`PORT@HOST`*`]...]`	Create a license file that identifies the named license server. Several redundant license servers can be specified. The server information is written as a license file. The license file will be installed for the user, unless `[--system]` is specified.
`[--system]`	Install the license file for all users in the ThreadSpotter™ installation directory.

Examples

Example 10. Installing a license file for the current user

Install a local license file for the current user. The license file can either be a node-locked file, or a copy of the license file installed on a license server machine.

```
$ license --file license.lic
```

Example 11. Installing a reference to three license servers

Install a system wide license file pointing to three machines running redundant FlexNet Publisher license servers on the default port.

```
$ license --system --server @10.0.0.2 --server @10.0.0.3 --server @10.0.0.4
```

Exit Status

0	Successful program execution.

1	Usage error.

Files

`$HOME/.threadspotter/license/license.lic`	Location of per-user license files. ThreadSpotter looks for license files here if no system wide license file can be found.
`/opt/threadspotter/license/license.lic`	Location of system wide license file.