# The MET User's Guide

**The Microarchitecture Exploration Toolset**
**IBM Thomas J. Watson Research Center**

**Version 0.1 (3/30/00)**

*Abstract*

The MET *is a collection of tools for supporting fast exploration of processor microarchitecture options for the PowerPC architecture. The entire toolset has being designed emphasizing the need for fast execution, so that execution traces consisting of hundreds of millions of instructions can be analized, and multiple experiments can be performed within a reasonably short turn-around time, on a normally configured workstation.*

*This User's Guide provides basic information on the entire toolset, where to find the tools, and details on using the support tools. Companion documents describe how to execute the existing processor models, how to interpret the results generated by the models, and how to build new models with different microarchitecture features.*

# Index

## 1. Introduction

Contemporary processor microarchitectures are complex. Superscalar and out-of-order instruction issue pipelines are only the beginning of a large domain of design options; other important aspects include features such as branch prediction, cache hierarchy, speculative execution, penalties for mispredictions, to name just a few. These complex features are often counterintuitive, requiring early, accurate, and timely modeling to ensure proper design trade-offs. The evaluation of such design options is further complicated by the varying behavior of programs. Different classes of applications, such as scientific, data-intensive, graphics-intensive, etc., exercise the features of a given processor in different manners. Even a single application may exhibit phases throughout its execution of very different characteristics, such as periods of varying cache misses, varying branch mispredict rate, and so on. As a result, the adequate evaluation of design trade-offs in a microprocessor requires examining a wide variety of inputs, representing the various types of applications of interest, each of a length that captures the varying behavior of the corresponding application. All these factors lead to a large input set for the evaluation.

The tools in use nowadays for the trade-off analysis of design alternatives in a microprocessor are mostly cycle-accurate processor models. These are detailed representations of the organization and behavior of the corresponding processor, as required to adequately exercise and evaluate the features being considered. Unfortunately, the combination of detailed features in the models and large input set required for an adequate evaluation leads to very long simulation time, which imposes limits to the number of experiments possible in a design trade-off exercise. This problem has been addressed in different ways, including the use of sampled execution traces (instead of entire traces), development of synthetic (compact) traces that attempt to capture the behavior of the programs of interest, combination of simulation with probabilistic modeling methods, among others. However, these alternative strategies have not yet been able to provide the same level of accuracy in the evaluation of trade-offs as that available from long simulations.

Research in this area at the IBM Thomas J. Watson laboratory has led to the development of The MET (Microarchitecture Exploration Toolset), a collection of tools for supporting fast exploration of processor microarchitecture options for the PowerPC architecture. The entire tool set has being designed emphasizing the need for fast execution, so that execution traces consisting of hundreds of millions of instructions can be analized, and multiple experiments can be performed within a reasonably short turn-around time, on a normally configured workstation. The MET's features resemble those in other processor simulation environments used in academia, such as SimpleScalar or Microarchitecture Workbench, but differ from them in several key areas:

- support for the PowerPC instruction set architecture;
- higher simulation speed, in the order of 100,000 cycles for a target processor per second;
- innovative mechanism for simulating the execution of instructions in a speculative state (mispredicted instructions), capturing those instructions, and inserting them into the execution traces;
- thorough validation of performance results.

To achieve the fast execution required, the tool set relies extensively on novel microarchitecture modeling techniques and judicious programming syles. These include

- extensive predecoded information, thus avoiding run-time decoding overhead;
- compile-time parameters for processor options, thus avoding run-time interpretation overhead for determining the structure of the processor; and
- programming styles that consider the compiler's optimizing behavior, thus reducing execution penalties in the simulator arising from branches, cache footprint size, and so on.

The tool set is based on trace-driven modeling, wherein traces of instructions executed by programs can be pregenerated by separate tracing facilities or collected dynamically from a workload's execution. The interface to both type of traces is the same. Although static traces do not allow the analysis of the behavior of mispredicted instructions, they offer these advantages

- experiment repeatibility, a feature that might not be guaranteed in execution driven environments if different versions of operating system and.or shared libraries are used across multiple workstations, or if the workload exhibits dependencies on the environment;

- ability to evaluate workloads from PowerPC platforms other than AIX; and
- ability to evaluate programs containing instructions not traceable within our environment

In the case of traces generated dynamically, The MET is capable of analyzing the behavior and performance of single-threaded user programs, in particular the effects of instructions executed by user code and by shared libraries.

Decoupling the trace generation process from the processor model, while retaining the ability to simulate mispredicted instructions, also has other advantages. For example, the trace generation mechanism allows the simulation of new instructions in the architecture; the mechanism can also be used to profile and tune programs.

The tool set is intended to support microarchitecture exploration prior to the complete definition of a processor implementation. Although the tools perform the analysis of the processor's behavior on a cycle-by-cycle basis, the tools do not model all the details of a processor pipeline but only those that are relevant to the exploratory evaluation.

This report is a Users' Guide for *The MET*. We first provide information on where to find the tools as well as processor models built using the tools which are already available. Related documents are

- *Aria User's Guide*, which describes
- *Turandot User's Guide*, which describes how to execute the processor models, how to interpret the results generated by the models, and finally how to build new models with different features and/or microarchitectures.

Results obtained from using processor models based on *The MET* have been reported elsewhere [xx].

## 2. Overview of the tool set

The MET includes the following components:

- An execution-driven trace generator that can trace user code and shared libraries (but not operating system code), and can also trace mispredicted instructions.
- A trace-driven parameterized processor model.
- A trace-driven tool with built-in models of existing PowerPC cache structures that provides profile-style data as well as cache behavior of a program, at the source code level and the PowerPC instruction level.
- A system-level simulator of the evaluation board for the PowerPC 403GCX embedded processor, also capable of generating instruction execution traces.
- Trace tools for various trace formats.
- A collection of tools for performance analysis and validation.

Microarchitecture exploration also benefits from compiler technology, because new compilation algorithms might impact the performance of programs, or may lead to changes in the processor organizations. Chameleon, an experimental compiler originally developed for a VLIW architecture, has been extended with capabilities for this type of studies, as well as for exploring the potential effects of changes to the instruction set architecture. The extensions include a PowerPC back-end, so code generated from Chameleon can be used as input to the microarchitecture exploration tools described here.

### 2.1 Execution-driven trace generator

*Aria* is an execution-driven trace generator. Aria traces program execution under the supervision of a control program (such as a processor model, a cache model, or even a program profiler), also referred to as the trace consumer. Both tools, and the program being traced, execute in the same address space so that simple procedure calls accomplish the communication between tools. Aria provides an application-programming interface (API) to enable the interaction between the trace consumer and the trace generation process.

At start-up, Aria determines the initial user program state and its starting address, and presents these data to the control program. This program controls the execution path through the user program by providing Aria the starting address of the instructions to be traced and a copy of the program state (all register values). Aria executes instructions starting at the given address, with the provided user program state, until the first branch encountered. Aria returns to the control program the execution trace for the executed instructions, plus the user program state modified by the execution of those instructions.

Aria generates traces by dynamically instrumenting a program at the basic block level. The first time Aria encounters a basic program block, it translates it into a code sequence that executes the block and produces a trace buffer including instruction and address information. Translated blocks are saved, eliminating repeated translation.

Aria can generate two versions of each block of instructions. The taken version corresponds to the block's normal execution; the not-taken version corresponds to a mispredicted path's execution. The control program specifies which version should be executed (and generated if necessary) each time. The not-taken version differs from the taken version as follows:

- load instructions are guarded to ensure they do not introduce spurious segmentation faults;
- store instructions are not executed, to avoid polluting the executing programs' memory state; and
- illegal instructions -data values embedded in the program- are replaced by no-ops.

Any other exception that raises a signal in the not-taken version leads to having the corresponding signal raised.

Aria's translation mechanism does not guarantee that data addresses placed in the trace are exactly those generated by the same program when executed independently. Data segments might be moved to a different region in the

address space during translation, such that data addresses can be offset from the original addresses. The approach tries to minimize such perturbation. Instruction addresses in the trace, however, are correct.

Aria also allows the simulation of new instructions in the architecture. Aria translates a new instruction into a native-instruction sequence that performs the same functionality, but the information in the trace corresponds to the new instruction.

Experimental evidence indicates that the slowdown factor arising from Aria's translation mechanism is around 40 instructions executed for each instruction in the program being traced.

### 2.2 The processor model

*Turandot* is a trace-driven superscalar processor model for the PowerPC architecture. This model is extensively parameterized, so that the same model can address numerous microarchitecture features. A typical configuration consists of components found in contemporary microprocessors but with their size and number increased to support wider instruction issue. The model implements a conventional pipeline but with variable number of stages to enable the effects of longer or shorter pipelines to be explored. For example, the pipeline's decode portion can range from one to four stages. Execution latencies vary depending on operation type; memory and floating-point operations require more stages than integer operations. Turandot models additional cycles for data cache misses and long-latency operations, such as divide and square-root.

The processor model lets designers explore multiple policies for removing operations from the issue queues and issuing them for execution. Policies commonly used include:

- An out-of-order policy issues each operation for execution as soon as the required operands and functional units are available, regardless of the order in which the operations were inserted in the issue queues. The issue logic selects the oldest one whenever multiple operations are ready for execution.
- A class-order policy issues the operations belonging to the same class of functional unit in program order, but allows out-of-order issuing among operations belonging to different classes.
- An in-order policy issues the operations in strict program order.

For any issue policy, the issue queues can be separated by operation class or shared among classes.

Other microarchitecture features of the model include

- instruction prefetching from a second level (L2) cache;
- decomposition of complex PowerPC instructions (such as load/store multiple or string operations) into multiple primitive operations;
- two-level translation look-aside buffer;
- clustering of functional units;
- L1/L2 cache bus features such as trailing-edge effects, multicycle transfers, critical-word first transfers;
- modeling of mispredicted instructions or stalling the pipeline until a mispredicted branch is resolved;
- dynamic load-over-store speculation with alias detection and recovery;
- various branch prediction algorithms for predicting branch direction/target.

Turandot also includes a mechanism to understand the behavior of a processor from the perspective of instruction retirement. Whenever an instruction cannot be retired in a given cycle, the cause ("trauma") that lead to such retirement failure is recorded. Histograms of traumas are provided at the end of a simulation session, which allow identifying the sources of performance degradation.

Turandot is the preferred processor modeling within The MET. Turandot takes advantage of the execution-driven features in Aria, but it can also processed instruction execution traces generated outside The MET.

Turandot is a tool for developing an understanding of the limits and potentials of PowerPC-based superscalar processors. Consequently, Turandot does not attempt to accurately model any specific processor implementation; instead, it attempts to model a generalized (perhaps idealized) processor suitable to explore some of the many variables involved. Consequently, the results provided by Turandot should by used to *determine trends* in performance values as features are changed, not as precise performance predictions of a specific configuration. There are prebuilt models that *approximate* the features of existing PowerPC processors, but the degree of approximation is dependent on how closely the features of those processors can be mapped to Turandot's capabilities and parameters. Users of the models should be aware of these limitations, and exercise caution when comparing the results generated by Turandot with those obtained from measurements in actual hardware. Turandot has been validated in detail for some specific processors, in particular a pre-silicon model of Power4.

Turandot supports the exploration of microarchitecture features through extensive parameterization. Parameters include changing the size of the various resources, the number and latency of functional units, the number of pipeline stages, enabling/disabling features, and so on. Complete details on using the processor model are given in the *Turandot Users' Guide*.

### 2.3 A source-level profiling tool

*LeProf*, and its companion post-processing tool *LeProft*, are used to profile programs, providing information regarding the programs' behavior. LeProf is characterized by dynamically instrumenting the program to be profiled, thereby being able to capture the effects of user code and shared libraries.

LeProf gathers extensive data about the instructions executed by the program, the function call behavior, the behavior of branches, and the data cache behavior (i.e., cache misses). Data is gathered at the following granularities:

- for the entire program;
- per function;
- per line of source code (if the program has been compiled using the option "-g");
- per PowerPC instruction.

Moreover, the tools generate the function-call graph, weighted by the frequency of function call invocation.

The tracing/profiling mechanism relies on Aria, the tool for dynamic instrumentation of programs. A program is dynamically instrumented by Aria, generating an execution trace that is used as input to a trace analyzer. The trace analyzer collects data regarding the execution of the program, and generates a file with the results. This file can be used as is to extract information regarding the program, or can be used as input to *LeProft* for generating summary results at various granularities, including source code line level (assuming that the program has been compiled with the -g option; note that the xlc family of compilers permits the use of -g with various levels of optimization, including -O2 and -O3).

### 2.4 An embedded system level simulator

*eOak* is a system-level simulator of the evaluation board for the PowerPC 403GCX embedded processor. It simulates the functionality of processor core (although some core functions are not included) and other components in the board (memories, UART, SPU). The simulation is performed at the functional level, assuming an execution rate of one instruction per cycle and instantaneous I/O.

eOAK is intended as a tool for the generation of instruction execution traces from embedded programs. Since the simulation performed is only functional, eOAK is limited to programs whose behavior is timing-independent. The simulator provides a dbx-like debugging interface, so it can also be used for development of embedded applications without using an actual development board. The performance of the simulator is in the order of 200Kips on a 200MHz 604e workstation.

Future extensions to this tool include:

- the ability to simulate the execution of processor-specific instructions, such as those in the PowerPC 405 processor;

- the emulation of all the devices on the board (DMA and Ethernet controllers, in particular); and
- the ability to simulate with the RISCwatch monitor/debugger.

### 2.5 Trace generation and analysis tools

The MET also includes a set of tools designed to create instruction execution traces from assembly-like text files; display the contents of trace files; examine traces for consistency, illegal or malformed instructions, address alignment, and other characteristics; generate traces (using Aria) and save them on disk, and so on.

Specific support tools are

| | |
|---|---|
| *disff* | Displays the contents of an fF trace. |
| *asff* | Creates an fF trace from an assembly-like text file. |
| *mkff* | Creates a trace from the execution of a program, using Aria as the tracing tool. |
| *opcount* | Counts the frequency of the operations in an fF trace. |
| *fFlint* | Examines a trace for consistency, illegal or malformed instructions, address alignment, branch alignment, etc. |

### 2.6 Performance validation tools

Performance analysis methods are needed during microarchitecture exploration to define and refine a system design point. A "system" can be just the processor microarchitecture, or it can include a group of processors (and memories) interconnected in some manner. Even within a uniprocessor system, one ideally needs to include the cache/memory subsystem, buses, and other components in a detailed analysis model.

The complexity of processors and systems makes increasingly difficult to ensure that the performance analysis is sufficiently accurate and robust, while being adequately fast. Terefore, testing and validation methods are of increasing importance even in exploratory stages of a design.

The MET includes a collection of tools for performance analysis and validation governed by an overall methodology, collectively known as *PavaRotti*. The toolset covers a range of analysis and validation methods applicable at various stages of design. The focus so far has been on model validation. The approach can be summarized as follows:

- Generation of fundamental micro-architecture performance parameter specifications:
  - Axiomatic formulation from specifications.
  - Automatic generation from existing reference model (if applicable or available).
  - Latency, bandwidth and other configuration parameters.
- Generation of bounds-based performance specifications:
  - Basic block and loop test specifications.
  - Cycle-count, cycles-per-instruction (cpi), cycles-per-flop (cpf), ...
  - Other complex "performance signatures".
  - Automatic and "expert, manual" generation.
  - Bounds mode, cycle-by-cycle timer mode.
- Validation against specifications:
  - Automatic test case generation, model validation.
  - Microvalidation of specifications using "timer-mode" output from tools.
- Cross-validation of alternate models:
  - Evolving timer model against (parts of) a reference model (if available), and vice versa.
  - Regression test suites; tools to compare timelines.
- Rules-based checking of model output timelines

## 3. Overview of the simulation environment

The MET tools can be used in two possible modeling scenarios:

***Static-tracing,*** wherein a trace consumer (such as a processor model or trace analyzer) is fed an existing instruction execution trace in FF52 format.

***Dynamic-tracing,*** wherein a trace consumer (such as a processor model or trace analyzer) is fed an execution trace generated on-the-fly, under control of the trace consumer. The trace may include the execution of predicted paths, even if those paths are not actually taken by the program, so the effects of mispredicted instructions can be taken into account by the trace consumer controlling the generation of the trace.

In the case of using the processor model Turandot as the trace consumer, both modeling scenarios collect summary data regarding the utilization of processor resources. Moreover, both scenarios can also generate a cycle-by-cycle description of the state of the processor, and the flow of instructions through the pipeline.

### 3.1 Static-trace processor modeling

The static-trace processor modeling scenario uses the following components:

- a *Turandot*-based processor model;
- a FF52 instruction execution trace to be fed to the processor model;
- a *pseudo-xcoff* file generated from the FF52 trace, which corresponds to a “binary image” of the program that generated the trace; and
- a file with predecoded information obtained from the *pseudo-xcoff* file.

### 3.2 Dynamic-trace processor modeling

The dynamic-trace processor modeling scenario uses the following components:

- *Aria*, the dynamic trace generator;
- a *Turandot*-based processor model;
- the object (*xcoff*) file of the program to be used in the modeling session, with its corresponding inputs; and
- a file with predecoded information obtained from the *xcoff* file of the program to be used.

## 4. Where to find *The MET*

Prebuilt models based on Turandot, the source code necessary for creating new models, and the collection of support tools, can be found at the MET directory in the Watson DFS cell

`/.../watson.ibm.com/fs/projects/MET/Turandot`

The directory structure at this place contains the following subdirectories:

| | |
|---|---|
| `/Source` | Source code for Turandot and other related tools. |
| `/bin` | Executable files for Aria and the various support tools used in conjunction with Turandot. |
| `/Models` | prebuilt versions of Turandot for some generic processor configurations, and versions corresponding to approximations to some existing PowerPC processors (604e,...). |

...

...

...

## 5. Users' guide to the trace generation and analysis tools

As stated earlier, The MET includes a set of tools designed to create instruction execution traces from assembly-like text files; display the contents of trace files; examine traces for consistency, illegal or malformed instructions, address alignment, and other characteristics; generate traces (using Aria) and save them on disk, and so on. This section gives brief instructions for using them.

### 5.1 disff

*disff* disassembles an fF format trace (also known as tt6 or tr6) into ASCII text. disff reads the trace from <stdin> and writes the text version of the trace to stdout. Usage is:

```
disff < <fF_trace>
```

An example of the output is as follows:

```
0x00001000
0x00001000: 0x7c85302e lwzx      r4,r5,r6       0x00020000
0x00001004: 0x84860000 lwzu      r4,0(r6)       0x00030000
0x00001008: 0x48004000 b         0x1000         0x00005008
0x00005008: 0x38630007 addi      r3,r3,7
0x0000500c: 0x7c074000 cmp       cr0,0,r7,r8
0x00005010: 0x4c04fa02 crand     cb0,cb4,cb31
0x00005014: 0x7c803d2a stswx     r4,r0,r7       0x00040000 12
0x00005018: 0x4c800021 bclrl     0x4,cb0        0x00010000
```

### 5.2 asff, asmff

*asff* converts a text file into an fF format trace (also known as tt6 or tr6 trace). *asmff* is a preprocessor which invokes *asff* and allows the use of some extended mnemonics in the text file. The command line for using asff (and also asmff) is as follows:

```
asff [-h] [-t] [-i <text_input_file>] [-o <binary_output_file]
```

wherein the optional arguments have the following function:

| | |
|---|---|
| -h | prints out help; |
| -t | prints to <stderr> a human-readable version of the trace being produced; |
| -i | text input file, defaults to <stdin> |
| -o | output file containing trace, defaults to <stdout> |

The text file used as input to asff accepts the following grammar:

```
file       ::= ADDR records
records    ::= | string_op | memory_op | branch_op | other_op
string_op  ::= insn ADDR LEN
memory_op  ::= insn ADDR
branch_op  ::= insn ADDR
other_op   ::= insn
insn       ::= OPCODE ARGS
```

`ADDR` and `LEN` can be given in any of the number formats recognized in the C language; that is:

- a string prefixed with 0x is a hexadecimal (base 16) number; e.g. 0x0f83
- a string prefixed with 0 is an octal (base 8) number; e.g. 07351
- other strings are decimal (base 10) numbers.

`OPCODE` is a string denoting the opcode of the instruction, whereas `ARGS` is a string containing all the arguments of the instruction. `ARGS` must not contain white-space.

An example input file is as follows:

```
0x1000
lwzx r4,r5,r6       0x2000
lwzu r4,0(r6)       0x3000
b 0x1000            0x5008
addi r3,r3,7
cmp  cr0,0,r7,r8
stswx r4,r0,r7      0x4000 4
crand cb0,cb4,cb31
bclrl 4,cb0         0x10000
```

The input grammar is characterized by the following features:

- the displacement for a branch instruction is in words, not bytes;
- compare instructions expect the second argument to be 0 or 1, denoting 32 or 64-bit compare operations, respectively;
- registers must be specified as names, not plain numbers, including condition register fields (eg, r7, fp5, cr0);
- condition bits, used in instructions such as crand or bc, must be specified as names (eg., cb0, cb11)

asmff recognizes the following *extended opcodes* defined in the PowerPC architecture:

nop — translates to `ori r0,r0,0`
`illop` — translates to `dsixes r0`

```
mfxer mflr mfctr
mfdsisr mfdar mfdec mfsdr1
mfsrr0 mfsrr1 mfsear mfpvr
mtxer mtlr mtctr
mtdsisr mtdar mtdec mtsdr1
mtsrr0 mtsrr1 mtear mttbl mttbu
bdz bdza bdzl bdzla
bdnz bdnza bdnzl bdnzla
bdzlr bdzlrl bdnzlr bdnzlrl
blr bctr blrl bctrl
```

### 5.3 mkff

*mkff* creates trace in the fF trace format from the execution of a program, by using the trace generator tool Aria. Usage is:

```
mkff [mkff_options] program [program_options]
```

wherein `mkff_options` are:

| | |
|---|---|
| -h | prints out help; |
| -q | surpresses all extraneous output; |
| -o file | output is written to the specified file, defaults to standard output; |
| -s S | start tracing after approximately S instructions; |
| -e E | stop tracing after approximately E instructions; |
| -i M N | trace (approximately) M instructions, then skip (approximately) N instructions; |
| -z | the output is a gzip-style compressed file. |

If both -i and -s are specified, then the first M instructions are traced after skipping approximately S instructions. If both -i and -e are specified, at most (approximately) E instructions are traced.

*mkff* is actually a shell script which combines three programs:

- the trace generator (*Aria*)
- the trace writer (*amkff*)
- the program being traced

Aria loads both amkff and the program into the same address space. After that, it tranfers control to the main routine of amkff. The execution then proceeds as follows:

- amkff calls Aria for trace of a  block of instructions;
- Aria executes some instructions of the program, generating a “micro-trace”' which is then returned to amkff;
- amkff produces the equivalent fF/tt6/tr6 format output for the micro-trace.

### 5.4 opcount

*opcount* counts the frequency of opcodes in the execution of a program. Usage is:

```
opcount [opcount_options] program [program_options]
```

wherein the `opcount_options` are:

| | |
|---|---|
| -h | prints out help; |
| -q | surpresses all extraneous output; |
| -o file | output is written to file; defaults to standard output. |

opcount is actually a shell script which combines three programs:

- the trace generator (*Aria*);
- the counter of instructions (*aopcount*); and
- the program being traced.

Aria loads both aopcount and the program into the same address space. After that it tranfers control to the main routine of aopcount. The execution then proceeds as follows:

- aopcount calls aria for trace of a  block of instructions;
- Aria executes some instructions of the program, generating  a ``micro-trace'' which is then returned to aopcount;
- aopcount looks at the instructions in the micro-trace, and increments the count for the opcode of each instruction.

### 5.5 LeProf

*LeProf*, and its companion post-processing tool *LeProft*, are used to profile programs, providing information regarding the behavior of a program. In addition to the profile of instructions executed, the tool gathers information regarding the data cache behavior of a program by simulating the directory of a data cache memory. The cache configurations available match the characteristics of many existing PowerPC processors, such as 603, 603e, 604, 604e, POWER, P2SC, and POWER2. Usage of this tools is as follows:

```
leprof [-h] [-q] [-c <config>] [-o <output_file>] [-p (1|2|3|4|5)*]
          [--] <program> [<args>...]
```

wherein the options are:

| | | |
|---|---|---|
| -h | prints out help; | |
| -q | run quietly; | |
| -o | output file, defaults to <ymon.out>; | |
| -c <config> | cache configuration to simulate, wherein <config> is one of the following options: | |
| | 603, 603e, 604 | 16K, 4-way associative, 32 byte line; |
| | 604e | 32K, 4-way associative, 32 byte line; |
| | POWER,RIOS | 64K, 4-way associative, 128 byte line; |
| | P2SC | 128K, 4-way associative, 128 byte line; |
| | POWER2,RIOS2 | 256K, 4-way associative, 256 byte line; |
| -p | statistics to be collected, with the following options: | |
| | 1 | overall summary; |
| | 2 | per-function summary; |

| | |
|---|---|
| 3 | per-instruction information for those with source line information (program must have been compiled with -g option); |
| 4 | per-instruction information for all instructions in object file; |
| 5 | function call graph. |

LeProf requires that the program being profiled must not be world-readable; if it is, leprof will temporarily turn off world-readabilty.

For example, finding the “hot spots” in the program foo.c is achieved as follows:

- compile the program using the -g flag (required to gather data at source code line level)

```
xlc -O2 -g -o foo foo.c
```

- execute the program

```
leprof -o foo.stats foo inputs
```

- find the source code line from which the most instructions are executed

```
leproft total foo.stats
```