
NAME

Post-link Optimization for Linux on POWER

SYNOPSIS

fdprpro -a *action* [*fdprpro-options*] *program*

DESCRIPTION

The Post-link Optimization for Linux on POWER is a performance-tuning utility for reducing the execution time and the real memory utilization of user-level application programs. The tool optimizes the executable image of a program by collecting information on the program's behavior under a typical workload, and creating a new version of the program that is optimized for that workload. The new program generated by the post-link optimizer typically runs faster and uses less real memory than the original program.

Note: The post-link optimizer applies advanced optimization techniques to a program. Some aggressive optimizations may result in programs that do not behave as expected. It is recommended to test the optimized program at least with the same test suite used to test the original program. The optimized program is not supported as input to the optimizer.

The post-link optimizer builds an optimized executable program in three distinct phases:

- Instrumentation:

  Creates an instrumented executable program and an empty template profile file. You do that by running **fdprpro** with the instrumentation action:

  ```
  $ fdprpro -a instr ... myprog
  ```

  This creates an instrumented file, *myprog.instr* by default, and a profile file, *myprog.nprof* by default.

- Training:

  Runs the instrumented program and updates the profile data.

- Optimization:

  Generates the optimized executable program file, given optimization options. You do that by running **fdprpro** with the optimization action, specifying the same input program, the profile file, plus the desired optimization flags:

  ```
  $ fdprpro -a opt -f myprog.nprof [opts ...] myprog
  ```

  This creates the optimized file for *myprog*, *myprog.fdpr* by default.

Note: The instrumented executable, created in the instrumentation phase and run in the training phase, typically runs several times slower than the original program. Due to the increased execution time required by the instrumented program, the executable should be invoked in such a way as to minimize execution duration, while still fully exercising the desired code areas.

Input files

The input file to **fdprpro** should be an ELF executable or shared library (`.so` file). Both ELF32 and ELF64 are supported.

Important: The executable program should be built with relocation information. Currently, **fdprpro** supports only the GCC compiler and the GNU linker. To leave the relocation information in the executable file, use the linker with the `--emit-relocs` (or `-q`) option. This can be specified in the GCC command by `-Wl,-q`.

Instrumentation and Profiling

Along with the instrumented file, **fdprpro** creates the profile file. The file is then filled with profile information (i.e., counts at various points in the program), while the instrumented program runs with its specified workload.

Important: The instrumented program requires a shared library called `libfdprinst32.so` (or `libfdprinst64.so` for ELF64 programs). Make sure the environment variable LD_LIBRARY_PATH is set to the directory containing these libraries.

Note: The instrumented program expects the profile file to be in the same directory as the instrumented program. To override this, set FDPR_PROF_DIR to the desired directory.

Optimizations

With the basic optimization flag, **-O**, **fdprpro** performs code reordering optimization together with the optimizations of branch prediction, branch folding, code alignment and removal of redundant NOOP instructions.

Higher level of optimizatons (**-O2**, **-O3**, and **-O4**) provide increasingly aggressive function inlining, DFA (data flow analysis) related optimizations, data reordering, and code restructuring (like loop-unrolling). While these optimization flags works well for most applications, optimal performance is typically achieved by carefull selection of specific optimizations for the given user's program.

Options

fdprpro accepts a host of optimization-specific options. In addition, there are a few options that create auxiliary files for debugging purposes (e.g., code disassembly).

Analysis Options:
-[no]aawc, --[no]analyze-assembly-written-csects

> Analyze objects written in Assembly. (When this option is used it must be specified at both the instrumentation and optimization phases).

-acf *analysis-configuration-file*, --analysis-configuration-file *analysis-configuration-file*

> Provide a configuration file of analysis information (advanced option).

-ifl *file*, --ignored-function-list *file*

> Set the ignored function list. The file contains names of functions that should not be instrumentated or optimized.

Instrumentation Options:
-fd *Fdesc*, --file-descriptor *Fdesc*

Set the file descriptor number to be used when opening the profile file. The default of *Fdesc* is set to the maximum-allowed number of open files.

-[no]ri, --[no]register-instrumentation

Instrument the input program file to collect profile information about indirect branches via registers. The default is set to collect the profile information.

-[no]sfp, --[no]save-floating-point-registers

Save floating point registers in instrumented code. The default is set to save floating point registers.

-spescr *0-127*, --spe-scratch-register *0-127*

Specify a global SPE scratch register, decreasing instrumenation overhead, in order to minimize possibility of local store overflow.

Profile Files Options:
-af *prof_file*, --ascii-profile-file *prof_file*

Set the name of an ASCII profile file containing profile information. There are three different XML entry options: *Simple-..-*, *Cond-..-* and *Reg-..-* for profiling data on regular, conditional or branch via register instructions, respectively.

-aop, --accept-old-profile

Accept the old profile file collected on previous versions of the input program file (requires the **-f** flag).

-f *prof_file*, --profile-file *prof_file*

Set the profile file name. The profile file is created during the instrumentation phase and read during the optimization phase. The profile file is updated each time when you run the instrumented program.

Optimization Options:
-A *alignment*, --align-code *alignment*

Align program so that hot code will be aligned on *alignment*-byte addresses.

-abb ***factor*****, --align-basic-blocks** ***factor***

Align basic blocks that are hotter then the average by given (float) *factor*. This is a lower-level machine-specific alignment compared to **--align-code**. Value of **-1** (the default) disables this option.

-bf, --branch-folding

Eliminate branch to branch instructions.

-bh ***factor*****, --branch-hint** ***factor***

add branch hints to basic blocks that are hotter then the average by given (float) *factor*. This is a SPE specific optimization. Value of **-1** (the default) disables this option.

-bp, --branch-prediction

Set branch prediction bit for conditional branches according to collected profile.

-dce, --dead-code-elimination

Eliminate instructions related to unused local variables within frequently executed functions. This is useful mainly after applying function inlining optimization.

-dp, --data-prefetch

Insert data-cache prefetch instructions to improve data-cache performance.

-ece, --epilog-code-eliminate

Reduce code size by grouping common instructions in function epilogs, into a single unified code.

-hr, --hco-reschedule

Relocate instructions from frequently executed code to rarely executed code areas, when possible.

-hrf ***factor*****, --hco-resched-factor** ***factor***

Set the aggressiveness of the **-hr** optimization option according to a factor value between (0,1), where 0 is the least aggressive factor (applicable only with the **-hr** option).

-i, --inline

Same as **--selective-inline** with **--inline-small-funcs** 12.

-ihf *pct*, --inline-hot-functions *pct*

Inline all function call sites to functions that have a frequency count greater than the given *pct* frequency percentage.

-isf *size*, --inline-small-funcs *size*

Inline all functions that are smaller or equal to the given *size* in bytes.

-kr, --killed-registers

Eliminate stores and restores of registers that are killed (overwritten) after frequently executed function calls.

-lap, --load-address-propagation

Eliminate load instructions of variables' addresses by re-using pre-loaded addresses of adjacent variables.

-las, --load-after-store

Add NOP instructions to place each load instruction further apart following a store instruction that reference the same memory address.

-lro, --link-register-optimization

Eliminate saves and restores of the link register in frequently-executed functions.

-lu *aggressiveness_factor*, --loop-unroll *aggressiveness_factor*

Unroll short loops containing of one to several basic blocks according to an aggressiveness factor between (1,9), where 1 is the least aggressive unrolling option for very hot and short loops.

-lun *unrolling_number*, --loop-unrolling-number *unrolling_number*

Set the number of unrolled iterations in each unrolled loop. The allowed range is between (2,50). Default is set to 2. (applicable only with the **-lu** flag).

-nop, --nop-removal

Remove NOP instructions from reordered code.

-O

Switch on basic optimizations only. Same as **-RC -nop -bp -bf**.

-O2

Switch on less aggressive optimization flags. Same as **-O -hr -pto -isf** 8 **-tlo -kr**.

-O3

Switch on aggressive optimization flags. Same as **-O2 -RD -isf** 12 **-si -dp -lro -las -vro -btcar -lu** 9 **-rt** 0.

-O4

Switch on aggressive optimization flags together with aggressive function inlining. Same as **-O3 -sidf** 50 **-ihf** 20 **-sdp** 9 **-shci** 90 and **-bldcg** (for XCOFF files).

-pbsi, --path-based-selective-inline

Perform selective inlining of dominant hot function calls based on control flow paths leading to hot functions.

-pca, --propagate-constant-area

Relocate the constant variables area to the top of the code section when possible.

-[no]pr, --[no]ptrgl-r11

Perform removal of R11 load instruction in _ptrgl csect.

-pto, --ptrgl-optimization

Perform optimization of indirect call instructions via registers by replacing them with conditional direct jumps.

-ptosl *limit_size*, --ptrgl-optimization-size-limit *limit_size*

Set the limit of the number of conditional statements generated by **-pto** optimization. Allowed values are between 1..100. Default value set to 3. (applicable only with the **-pto** flag).

-ptoht *heatness_threshold*, --ptrgl-optimization-heatness-threshold *heatness_threshold*

Set the frequency threshold for indirect calls that are to be optimized by **-pto** optimization. Allowed range between 0..1. Default is set to 0.8. (applicable only with **-pto** flag).

-RC, --reorder-code

Perform code reordering.

-rcaf *aggressiveness_factor*, --reorder-code-aggressivenes-factor *aggressiveness_factor*

Set the aggressiveness of code reordering optimization. Allowed values are [0 | 1 | 2], where 0 preserves original code order and 2 is the most aggressive. Default is set to 1. (applicable only with the **-RC** flag).

-rcctf *termination_factor*, --reorder-code-chain-termination-factor *termination_factor*

Set the threshold fraction which determines when to terminate each chain of basic blocks during code reordering. Allowed input range is between 0.0 to 1.0 where 0.0 generates long chains and 1.0 creates single basic block chains. Default is set to 0.05. (applicable only with the **-RC** flag).

-rccrf *reversal_factor*, --reorder-code-condition-reversal-factor *reversal_factor*

Set the threshold fraction which determines when to enable condition reversal for each conditional branch during code reordering. Allowed input range is between 0.0 to 1.0 when 0.0 tries to preserve original condition direction and 1.0 ignores it. Default is set to 0.8 (applicable only with the **-RC** flag).

-RD, --reorder-data

Perform static data reordering.

-rmte, --remove-multiple-toc-entries

Remove multiple TOC entries pointing to the same location in the input program file.

-rt *removal_factor*, --reduce-toc *removal_factor*

Perform removal of TOC entries according to a removal factor between (0,1), where 0 removes non-accessed TOC entries only, and 1 removes all possible TOC entries.

-sdp *aggressiveness_factor*, --stride-data-prefetch *aggressiveness_factor*

Perform data prefetching within frequently executed loops based on stride analysis,

according to an aggressiveness factor between (1,9), where 1 is least aggressive.

-sdpla *iterations_number*, --stride-data-prefetch-look-ahead *iterations_number*

Set the number of iterations for which data is prefetched into the cache ahead of time. Default value is set to 4 iterations. (applicable only with the **-sdp** flag).

-sdpms *stride_min_size*, --stride-data-prefetch-min-size *stride_min_size*

Set the minimal stride size in bytes, for which data will be considered as a candidate for prefetching. Default value is set to 128 bytes. (applicable only with the **-sdp** flag).

-shci *pct*, --selective-hot-code-inline *pct*

Perform selective inlining of functions in order to decrease the total number of execution counts, so that only functions whose hotness is above the given percentage are inlined.

-si, --selective-inline

Perform selective inlining of dominant hot function calls.

-sll *Lib1:Prof1,...,LibN:ProfN*, --static-link-libraries *Lib1:Prof1,...,LibN:ProfN*

Statically link hot code from specified dynamically linked libraries to the input program. The parameter consists of comma-separated list of libraries and their profiles. IMPORTANT: licensing rights of specified libraries should be observed when applying this copying optimization.

-sllht *hotness_threshold*, --static-link-libraries-hotness-threshold *hotness_threshold*

Set hotness threshold for the **--static-link-libraries** optimization. The allowed input range is between 0 (least aggressive) to 1, or **-1**, which does not require profile and selects all code that might be called by the input program from the given libraries. Default is 0.5.

-sidf *percentage_factor*, --selective-inline-dominant-factor *percentage_factor*

Set a dominant factor percentage for selective inline optimization. The allowed range is between (0,100). Default is set to 80 (applicable only with the **-si** and **-pbsi** flags).

-siht *frequency_factor*, --selective-inline-hotness-threshold *frequency_factor*

Set a hotness threshold factor percentage for selective inline optimization to inline all

dominant function calls that have a frequency count greater than the given frequency percentage. Default is set to 100 (applicable only with the **-si -pbsi** flags).

-slbp, --spinlock-branch-prediction

Perform branch prediction bit setting for conditional branches in spinlock code containing l*arx and st*cx instructions (applicable after **-bp** flag).

-sldp, --spinlock-data-prefetch

Perform data prefetching for memory access instructions preceding spinlock code containing l*arx and st*cx instructions.

-so, --stack-optimization

Reduce the stack frame size of functions which are called with a small number of arguments.

-tb, --preserve-traceback-tables

Force the restructuring of traceback tables in reordered code. If **-tb** option is omitted, traceback tables are automatically included only for C++ applications which use the Try & Catch mechanism.

-rtb, --remove-traceback-tables

Remove traceback tables in reordered code.

-tlo, --tocload-optimization

Replace each load instruction that references the TOC with a corresponding add-immediate instruction via the TOC anchor register, when possible.

-vro, --volatile-registers-optimization

Eliminate stores and restores of non-volatile registers in frequently executed functions by using available volatile registers.

Output Options:

-d, --disassemble-text

Print the disassembled text section of the output program into *output_file*.dis_text file.

-dap, --dump-ascii-profile

Dump profile information in ASCII format into *program*.aprof (requires the **-f** flag).

-db, --disassemble-bss

Print the disassembled bss section of the output program into *output_file*.dis_bss file.

-dd, --disassemble-data

Print the disassembled data section of the output program into *output_file*.dis_data file.

-diap, --dump-initial-ascii-profile

Dump initial profile information in ASCII format into *program*.aprof.init (requires the **-f** flag).

-dim, --dump-instruction-mix

Dump instruction mix statistics based on gathered profile information.

-dm, --dump-mapper

Print a map of basic blocks and static variables with their respective new -> old addresses into a *program*.mapper file.

-o *output_file*, --output-file *output_file*

Set the name of the output file. The default instrumented file is *program*.instr. The default optimized file is *program*.fdpr.

-pif, --print-inlined-funcs

Print the list of inlined functions along with their corresponding calling functions, in ASCII format into a *program*.inlined file (requires the **-si** or **-i** or **-isf** flags).

-ppcf, --print-prof-counts-file

Print the profiling counters in ASCII format into a *program*.counts file (requires the **-f** flag).

-simo, --single-input-multiple-outputs

Optimize in parallel into multiple outputs as specified by option sets read from stdin.

-sf, --strip-file

Strip the optimized output file.

-cep, --complement-edge-profile

Complements given partial profile information of basic blocks' frequencies by adding missing basic block-to-basic block edge counts.

-spedir *directory*, --spe-directory *directory*

Set the directory into which SPE executables will be extracted and from which they will be encapsulated.

-enc, --encapsulate

Encapsulate SPE executables present in the PPE input (see **--spe-directory**).

General Options:

-h, --help

Print online usage help.

-m *machine-model*, --machine *machine-model*

Generate code for the specified machine model. Target machine can be one of the following models: power2, power3, ppc405, ppc440, power4, ppc970, power5, power6, ppe, spe, spe_edp. Default is set to no machine.

-q, --quiet

Set quiet output mode, suppressing informational messages.

-st *stat_file*, --statistics *stat_file*

Output statistics information to *stat_file*. If *stat_file* is '-', output goes to standard output. See **--verbose** for the default.

-V, --version

Print version.

-v *level*, --verbose *level*

Set verbose output mode level. When set, various statistics about the target optimized program are printed into file *program*.stat. Allowed level range is between (0,3).

Default is set to 0.

-cell, --cell-supervisor

Integrated PPE/SPE processing. Perform SPE extraction, processing, and encapsulation automatically prior to PPE processing.

Profiling SPE executable files

With PPE executables, a profile file is generated along with the instrumented file. It is then filled with counts while the instrumented file runs. In contrast, with SPE executables, the profile is generated when the instrumented executable runs. A PPE/SPE executable run typically generates a number of profiles, one for each SPE image whose thread is executed. Such profile accumulates the counts of all the threads which execute the corresponding image. A SPE profile is generated by default in the output directory, and is named <spename>.mprof by default (see the **SPE Processing** section below).

Note: If an old profile exists before instrumentation starts, the new data will be accumulated in it. This may be desirable, e.g., if the user wants to combine profiles of number of workloads. Otherwise, the old profile should be manually removed before commencing with instrumentation.

Important: **fdprpro** uses a lock file `/tmp/fdpr_xflck` to synchronize between multiple SPE threads attempting to update a common profile file. The file is created and removed one or more times during an instrumented run. Following some exceptional conditions the file may exist after instrumentation, which might cause problems in following instrumentation runs. It is a good idea to remove this file, if it exists, before commencing with instrumentation.

Processing PPE/SPE executable files

The hybrid PPE/SPE executable file poses a veritable challenge to **fdprpro**. By default **fdprpro** processes the executable file, depending on the target machine, PPE or SPE. If this is a PPE file, its embedded SPE's are ignored. Two modes are available in order to fully process the PPE/SPE hybrid file: *integrated mode*, and *standalone mode*.

Integrated mode

Integrated mode hides the details of SPE processing. It provides convenient interface for

performing the full PPE/SPE processing at the expense of lesser flexibilty. To fully process a PPE/SPE file, simply add the **-cell** (or **--cell-supervisor**) to the standard **fdprpro** command. For example:

```
$ fdprpro -cell -a instr myprog -o myprog.instr
```

And, for optimization:

```
$ fdprpro -cell -a opt myprog -f myprog.nprof -o myprog.fdpr
```

Two additional options might be useful here, **-spedir**, and **-spefdir**. The first specifies the directory into which SPE files are extracted, where they are processed, and from where they are encapsulated back into the PPE file. If this option is not specified, a temporary directory is created and is deleted if **fdprpro** ends successfully. The **-spefdir** is usefull in the optimization phase and specifies the directory where SPE profiles can be found (by default they are expected in the same directory where the input program resides).

Standalone mode

In integrated mode the same optimization options are used when processing the PPE file and when processing each of the SPE files. Full flexibility is available in standalone mode, where the user specifies the explicit commands needed to extract the SPE files, process them, and then encapsulate and process the PPE file:

- Extraction

  SPE images are extracted from the input program and placed as executable files in the specified directory:

  ```
  $ fdprpro -a extract -spedir mydir myprog
  ```

- SPE processing

  The SPE images are processed one by one. All the output files should be placed in a distinct directory **by their original name**:

  ```
  $ fdpdpro -a <action> mydir/<spe1> [-f <prof1>] [opts ...] -o outdir/<spe1>
  $ fdpdpro -a <action> mydir/<spe2> [-f <prof2>] [opts ...] -o outdir/<spe2>
  ...
  ```

  The *action* is either `instr` or `opt`. The profile file is specified, as with the PPE case, by the **-f** option. If not specified, it is assumed to be *outdir*/<*spe*.mprof>.

  Note: FDPR_PROF_DIR environment variable cannot be used for overridinng profile directory (see section **Instrumentation and Profiling** above).

- Encapsulation and PPE processing

  Finally the SPE files are encapsulated as a part of the actual PPE processing:

  ```
  $ fdprpro -a <action> --encapsulate -spedir outdir [opts ...] myprog
  ```

  Note that this time -spedir is used to specify the output SPE directory.

ASCII profile

By default the profile generated by **fdprpro** is in some internal binary format. To allow external tools to generate the profile, an ASCII profile is also supported (see **--ascii-profile-file**).

The format of the ASCII profile file is:

```
<Simple> address execCount </Simple>
<Cond> address execCount fallthruCount </Cond>
<Reg> address execCount fallthruCount regIndex
type1 value1 execCount1
type2 value2 execCount2
...
typeN valueN execCountN
</Reg>
```

The profile file is set of the Profile entries - Simple, Cond and Reg. The types in <Reg> entries are Abs - for Absolute Values, Text - for Text addresses, Data - for Data addresses. There are no other ``tags'' defined, there must not be white spaces between the tags` letters, no comments. Addresses and Values can be in decimal or in hex form (starting with 0x).

For example -

```
<Simple> 0x100000240 10 </Simple>
<Simple> 0x100000250 20 </Simple>
<Cond> 0x100000260 20  10 </Cond>
<Simple> 0x100000270 20 </Simple>
<Reg>  0x100000260 20  10 17
Abs 23 5
Text  0x100000300 5
Data  0x200000400 10
</Reg>
```

The order of the profile entries is not important, although for better readability they should be sorted according to address. The ASCII profile file (extension .aprof) should contains entries for code executed at least once. The code with execCount = 0 should not be included (it is not forbidden but will not provide any information to fdpr). Generally it is sufficient to provide one profile entry for each executed basic block. The address of that profile entry should be any address within the basic block. Since **fdprpro**'s internal basic block partitioning is not always known, several profile entries may be provided for a single basic block up to the maximum of one profile entry for each instruction. When several profile entries are provided for a single basic block and they contain conflicting information (e.g., different execCount), **fdprpro** will produce a warning starting with ``Conflicting profiling'' ... and ignore the later conflicting information.

Human-readable output

In addition to the optimized or instrumented program, **fdprpro** produces human readable output.

1. Standard output. The text that goes to standard output includes the signon message, progress information and signoff message. The progress information displays the passage of **fdprpro** along the different phases of processing, as follows:

```
    FDPR-Pro <version>
    fdprpro -a opt -O3 li.linux.gcc32.base
    > reading_exe ...
    > adjusting_exe ...
    > analyzing ...
    > building_program_infrastructure ...
...
    > updating_executable ...
    > writing_executable ...
    bye.
```

If the **--quiet** option is specified, no output is produced here.

2. Standard error. As usual, warnings and errors messages are written to the standard error file. Note that **fdprpro** exists after the first error.

3. Statistics file. If the **--verbose <level**> option is selected, various kinds of statistics about the program will be written to the statistics file, *output_file*.stat. The file consists of a list of tables, typically in a form of <attribute> <value> per line. The amount of information is determined by *level*. The following is an example, corresponding to the above invocation:

```
    options. group              active_options
    options. optimization       -bf -bp -dp -hr -hrf 0.10 -kr -las -lro -lu 9 -isf 12 -nop -pr -RC -RD -rt 0.00 -si -tlo -vro
    options. output             -o 1.base

    global.use_try_and_catch:           0
    global.profile_info:          not_available

    file.input:                 li.linux.gcc32.base
    file.output:                1.base
    file.statistics:            1.base.stat

    analysis.csects:                347
    analysis.functions:             343
    analysis.constants:              13
    analysis.basic_blocks:         5360
    analysis.function_descriptors:      0
    analysis.branch_tables:            10
    analysis.branch_table_entries:     374
    analysis.unknown_basic_units:       17
    analysis.traceback_tables:          0
...
```

Note, the options specified in the optimization group are the actual ones enabled by the -O3 option.

Importing code from shared libraries

Typically **fdprpro** optimizes a single target module (an executable file or a shared library), without considering the cross-module flow of the program. The **--static-link-libaries** option allows **fdprpro** to go beyond the boundary of the target module and import hot code (i.e., heavily used) from other modules to which it is dynamically linked. These modules are referred below as *SLL libraries*.

For example, to import hot code from `mylib.so` using its profile `mylib.so.prof`, to `myprog`, use the following command:

```
$ fdprpro -sll mylib.so:mylib.so.prof -O3 -o myprog.fdpr -f myprog.prof myprog
```

For better performance results, it is highly recommended that users collect the profiles of the specified SLL libraries with the same workload as the one used for training the target program.

IMPORTANT: If an SLL library is later upgraded, the optimization must be rerun with the upgraded library to keep the correspondence valid between that library and the target module.

IMPORTANT: It is the responsibility of the user to ensure that code copying from SLL libraries is compliant with the usage license of these libraries.

FILES

***installed_dir*/bin/fdpr**

The **FDPR-Pro** wrapper script (by default *installed_dir* is **/opt/ibm/fdprpro**).

***installed_dir*/bin/fdprpro**

The actual **FDPR-Pro** executable (binary) program.

***installed_dir*/lib/libfdprinst32.so**

The shared library used during profiling for ELF32 executable files.

***installed_dir*/lib/libfdprinst64.so**

The shared library used during profiling for ELF64 executable files.

***output_file*.dis_text**

The disassembly file of program text, produced by the **--disassemble-text** option.

***output_file*.dis_data**

The disassembly file of program data, produced by the **--disassemble-data** option.

***output_file*.dis_bss**

The disassembly file of program data, produced by the **--disassemble-bss** option.

***output_file*.mapper**

The map of basic block and static variables. See the **--dump-mapper** option.

***output_file*.aprof_init**

The initial profile information in ASCII format. See the **--dump-initial-ascii-profile** option.

***output_file*.aprof**

The ASCII-formatted profile file. See the **--dump-ascii-profile** option.

***output_file*.autoerr_log**

In case of error, the file contains information related to the error. Please send it with the bug report to fdpr@il.ibm.com.

***output_file*.stat**

If **--verbose <level>** is specified the file will contain certain statistics about the target program or about the optimization process.