Application note

ST200 VLIW series
ST200 I-cache optimization techniques

Synopsis

This document aims to define a methodology for optimizing I-cache on the ST200. It is most relevant to cores that have a direct mapped I-cache such as the ST231. It is not so applicable to the ST240 core that has a four-way associative I-cache.

This document covers:

Contents

1 Introduction

The I-cache architecture of the ST200 processors family is a 32-Kbyte direct mapped cache with 64-byte lines. The direct mapped cache is the simplest and cheapest type of cache architecture, but as a drawback, it results in higher cache miss rates, compared to more sophisticated cache models, like associative multi-way caches. Furthermore, the ST200 is an instruction-level parallel VLIW processor, thus it has a large instruction bandwidth requirement (inducing an increase in the cache miss rate) and also a large cache miss penalty. For these reasons, it is important to ensure that code running on the ST200 makes effective use of the instruction memory system.

To increase this effectiveness, STMicroelectronics has developed a binary optimization tool (called **binopt**) which includes an I-cache optimization tool. The I-cache optimization phase of **binopt** controls the layout of functions in the binary code to be executed on the ST200. This is achieved by using placement algorithms to reorder the functions in the final executable. The placement algorithms minimize I-cache conflicts by optimizing spatial locality and thus optimize I-cache usage.

The aim of this document is to define a methodology for optimizing I-cache usage on the ST200, which applies to real user application code. Firstly, the context of this application note is defined. Then the different I-cache optimization techniques are stated and described. Finally, the results on a chosen test-case application are presented.

2 Context

The context of this application note is the I-cache optimization of libraries. This was motivated by the fact that key applications running on the ST200, such as audio-video application code, are composed of one main driver (or system), and several modules (or packages), organized as separate individual libraries (a module is a set of source files, compiled and then grouped together in a library).

For instance, in the DVD/ACC audio codec application suite, each library contains the code for a specific audio process (coder, decoder or post-processing) and all these processes are driven by a main system. Supposing there are N libraries (`lib1.a`, `lib2.a`,...,`libN.a`), the classical build commands are the following:

```
# build and creation of the libX.a library
# for process X (X=1,2,..,N)
        st200cc -c libX*.c
        st200ar libX.a libX*.o
# build of the main driver objects
        st200cc -c main*.c
# creation of the final executable
        st200cc -o main.exe main*.o lib*.a
```

The result is the executable `main.exe`, containing the main system objects and the N libraries. Within this context, the code placement in the final executable is not controlled at all, and thus, depending on the process to be run, the I-cache miss cycles can be very high and lead to significant performance loss.

3 I-cache miss issues diagnosis

The purpose of this section is to give the application developer some hints on how to investigate the following issues:

- How to detect that an application is subject to I-cache miss issues,
- How to detect which are the incriminating functions.

To diagnose I-cache issues on application code, the first thing to measure is the I-cache miss cycle count, compared to the total cycles. A high value of the ratio I-cache miss cycles/total cycles is a good indicator of I-cache related issues in the code. These values can be obtained either through the simulator statistics (if simulation is possible), or through the performance monitoring hardware block (by instrumenting the code or using the Board Support Package (BSP) facility).

However, these issues cannot always be completely solved by I-cache optimization techniques, because an I-cache miss can either be a conflict miss or a cold miss (occurring the first time a function is loaded in the cache). In the latter case a reordering of the functions in the binary code will not be able to remove the miss. The number of these non-removable I-cache misses is proportional to the code size of the library to be executed.

In fact, the best figure to track is the number of I-cache conflicts occurring at runtime. This value gives the upper bound of the attainable performance in terms of reducing I-cache miss cycles. If the number of cycles due to conflict misses (one I-cache miss roughly costs 150 cycles) is negligible compared to the total number of cycles, there is no point in trying to optimize function placement. Conversely, if a lot of I-cache conflicts are measured, this indicates inefficient code placement which should be improved by applying I-cache optimization techniques.

The **st200gprof** profiling utility, included in the ST200 Micro Toolset, is a valuable tool for detecting which functions may conflict. For this purpose, the profiling mode of the simulator must be used during execution to generate the gprof profiling I-cache information file `gmon.outICACHE`. This file can then be processed by **st200gprof** to produce a view of the I-cache miss cycles function tree, including the percentage of miss cycles for each function, which may indicate which functions are subject to conflicts (even if there is no information on conflicting function pairs).

4 I-cache optimization techniques

This chapter presents the different steps or methods of I-cache optimization to be applied to minimize the I-cache conflicts in the context defined in the preceding chapters, without increasing the application footprint (or with negligible effects on code size expansion).

4.1 Relocatable linking

Before using the ST200 **binopt** tool, a first attempt at code placement can be achieved by performing a relocatable link of the object files of each library, using the **st200cc** `-r` option, before creating the library. Here is the modified build command sequence:

```
# build and creation of the libX.a library
# for process X (X=1,2,..,N)
        st200cc -c libX*.c
        st200cc -r -nostdlib -o libX.ro libX*.o
        st200ar libX.a libX.ro
# build of the main driver objects
        st200cc -c main*.c
# creation of the final executable
        st200cc -o main.exe main*.o lib*.a
```

The purpose of the intermediate link in relocatable mode (`-r`) is to improve spatial locality by ensuring that all functions in the same module are placed close together.

Note: The `-nostdlib` *option, which disables the inclusion of the standard libraries at link time, has to be added in order to prevent multiple definitions at the final link stage.*

Though it is very simple, this first optimization step is crucial to achieve optimal I-cache performance and will always be associated with more complex optimization methods.

4.2 Static I-cache optimization

As stated in the previous section, performing an intermediate link of each module helps to improve the spatial locality of the binary code. Nevertheless, the placement of the functions in each module library is not controlled at all by this method. The use of the binary optimization tools is needed to optimize the function layout, by performing a function reordering before creating each library. This I-cache optimization phase is automatically performed at link-time when using optimization levels `-O2`, `-Os` or `-O3` (the command line option has to be given both at compile and link-time). I-cache optimization can also be explicitly turned on using both `--icache-opt=on` and `--icache-static=on` options at link-time.

By default, a static I-cache optimization is performed, which means that the binary optimizer reorders the functions according to static compiler-estimated frequencies. These frequencies correspond in fact to the weight of each edge of the static call-graph, which is computed by **binopt** and used by the reordering algorithm.

At the final global link phase, the I-cache optimizer must be turned off, to avoid modifying the previously locally optimized function layout in each library. It is turned off by default at optimization levels `-O0` and `-O1`, but can be explicitly turned off using the `--icache-opt=off` option.

The build commands sequence is modified in the following way:

```
# Optimized compilation of the libX*.c sources
# for process X (X=1,2,..,N)
        st200cc -O2 -c libX*.c
# Intermediate link and I-cache optimization
# phase (triggered by the -O2 flag)
        st200cc -O2 -r -nostdlib -o libX.ro libX*.o
# creation of the I-cache optimized libX.a library
# for process X (X=1,2,..,N)
        st200ar libX.a libX.ro
# build of the main driver objects
        st200cc -c main*.c
# creation of the final executable
# (with I-cache optimizations turned off)
        st200cc -o main.exe main*.o lib*.a --icache-opt=off
```

A set of options is provided to help the static I-cache optimizer better estimate the static call-graph of the application. These can be passed to the binary optimization phase through the **st200cc** driver using the `-Wo,[option]` syntax.

In the context of several modules packaged in separate libraries, the use of an I-cache call-graph user configuration file (`.icg` file) may be helpful, as it allows edges to be added to or removed from the static estimated I-cache call-graph. For example, in the case of indirect calls, the simple relocation analysis performed by **binopt** is not able to catch these calls. The corresponding edges can then be added manually into the `.icg` file.

The `.icg` file consists in a list of items:

```
proc1   proc2   freq
```

where `proc1` and `proc2` are function names and `freq` is a floating-point value which is an estimation of the numbers of execution of `proc2` each time `proc1` is executed. The value of `freq` is in the range `[0.0,+inf)`.

Then, the `.icg` file is passed to the binary optimization phase at module link time:

```
# Intermediate link and I-cache optimization
# phase with .icg file use
      st200cc -O2 -r -nostdlib -o libX.ro libX*.o -Wo,--icg,libX.icg
```

*Note: The edges described in the `.icg` are added to the pre-existing call-graph estimated by **binopt** and thus the `.icg` file must not contain all the call-graph edges. To remove an edge, for instance between `proc1` and `proc2`, a line with a frequency value of `0.0` must be added:*

```
# remove the edge between proc1 and proc2
proc1   proc2   0.0
```

Now for a practical example. Suppose you want to optimize the I-cache miss cycles for an application containing the following piece of C code:

```
#define N 10
extern void proc2(void);

void proc1(void (*proc)(void)) {
  int i;
  for (i=0; i<N; i++)
```

```
        (*proc)();
}

void proc3(int i){
  if (i)
    proc1(proc2);
}
```

By default, the binary optimizer only sees the edge between `proc3` and `proc1` (corresponding to the call to `proc1` in `proc3`), putting a frequency of 0.5 (the default frequency value for a single block if construct). But supposing that on average, the `if` condition in `proc3` is true for 90% of the cases, this information can be given to the I-cache optimizer using an `.icg` file.

In the same way, the call of `proc2` from `proc1` is indirect, so it is not taken into account by **binopt**. To fix this, an entry can be added to the `.icg` file.

The `.icg` file for this example could be the following:

```
# add the edge between proc1 and proc2 (frequency of N=10.0)
proc1   proc2   10.0
# refine the frequency estimation of the proc3/proc1 edge,
# overriding the compiler estimation of 0.5
proc3   proc1   0.9
```

4.3 Dynamic I-cache optimization methods

Static I-cache optimization can be inaccurate because it relies on static estimations of frequencies based on heuristics. To improve the accuracy of the function reordering, the use of dynamic I-cache optimization methods is preferred. Basically, this uses run-time data instead of estimations. It implies that the application must be first run to collect this data by profiling, before rebuilding the executable to take advantage of the profiling information collected.

There are two kinds of dynamic I-cache optimization methods: using data from a gprof file to drive the I-cache optimizer, and using the profiling feedback optimization (PFO) method.

4.4 gprof file-driven I-cache optimization

In this method, the module sources are compiled using the `-pg` option, which instruments the code to collect gprof-formatted profiling information at run-time. The commands for this first build are:

```
# compilation in profiling mode of the libX*.c
# sources for process X (X=1,2,..,N)
        st200cc -O2 -pg -c libX*.c
# Intermediate link in relocatable mode
        st200cc -O2 -r -nostdlib -o libX_pg.ro libX*.o
# creation of the profiling instrumented libX_pg.a
# library for process X (X=1,2,..,N)
        st200ar libX_pg.a libX_pg.ro
# build of the main driver objects
        st200cc -c main*.c
# creation of the final executable including the profiling libraries
        st200cc -pg  -o main_pg.exe main*.o lib*_pg.a
```

The `main_pg.exe` executable, instrumented by the compiler produces profiling information. The next step is to run each process with a significant input data set. Each run produces a file named `gmon.out.000`, which may be renamed to `gmon_X.out` (for process X=1,2,..,N) to attach it to the process that it describes. In the case of several runs of the same process (producing several `gmon_X.out.`*xxx* files) the following commands can be used to sum-up all the collected information in a single `gmon_X.out` file.

Note: This may take a while to execute:

```
# sum-up the profiling information in the default gmon.sum file
st200gprof --sum main_pg.exe gmon_X.out.*
```

Rename the output file `gmon.sum` to `gmon_X.out`.

After having run all processes and produced all `gmon_X.out` files, the last step is to rebuild the application, taking advantage of the profiling information.

This is achieved by passing at module link-time the two options:

- `--icache-profile=gmon_X.out`,
- `--icache-profile-exe=main_pg.exe`.

Note: The optimization level at module compile-time must be the same as the one used for the first compilation, to be consistent with the profiling instrumentation (`-O2` in this case).

```
# optimized compilation of the libX*.c sources
# for process X (X=1,2,..,N)
        st200cc -O2 -c libX*.c
# Intermediate link in relocatable mode with profile file-driven
# information
        st200cc -O2 -r -nostdlib --icache-profile=gmon_X.out \
                --icache-profile-exe=main_pg.exe -o libX.ro libX*.o
# creation of the I-cache optimized libX.a library
# for process X (X=1,2,..,N)
        st200ar libX.a libX.ro
# build of the main driver objects
        st200cc -c main*.c
# creation of the final executable
```

```
# (with I-cache optimizations turned off)
        st200cc -o main.exe main*.o lib*.a --icache-opt=off
```

This method can either be used on the simulator or on real hardware. But if it is possible to run the application code on the simulator, an alternative equivalent method can be applied. Instead of using the `-pg` option of the compiler, the profiling feature of the simulator can be used to produce the `gmon_X.out` file. The run command is as follows:

```
# run X process (with X_args arguments) in ST231 little-endian mode
# using the profiling mode of the simulator
st200xrun -c st200sp -t st231profsimle -e main.exe -a X_args
```

Rename the output file `gmon.out` to `gmon_X.out`.

The advantage of this method, compared to the use of the `-pg` option, is that it is not necessary to re-compile, but only to re-link the modules, using the same options `--icache-profile=gmon_X.out` and `--icache-profile-exe=main.exe`. The drawback is that it does not apply to processes which can only be run on a board.

4.5 Profiling Feedback Optimizations (PFO)

For the gprof file-driven I-cache optimization method, the PFO method is applied in steps:

1. Compile the application with a special option to instrument the code.
2. Run each process to gather feedback data.
3. Re-build the application to benefit from the run-time information collected.

Though the set-up of both methods is very similar, they do not lead to the same results. The main difference lies in the fact that the gprof file-driven I-cache optimization only applies at link-time, whereas the PFO applies at compile-time (recompilation is needed to use the PFO information), not only affecting the functions layout, but also the code-generation phase.

4.5.1 Instrument the code

The first step for performing the PFO is to generate an instrumented executable. This is achieved by adding the `-fb_create<name>` command line option at compile-time. This option must also be passed at final link-time, in order to tell the linker to add the `libinstrC.a` library that contains the routines collecting the feedback information (in particular the frequency counts on control flow).

The modified build commands are:

```
# compilation with profiling feedback instrumentation
# of the libX*.c sources for process X (X=1,2,..,N)
       st200cc -O2 -fb_create fb_data -c libX*.c
# Intermediate link in relocatable mode
       st200cc -O2 -r -nostdlib -o libX_pfo.ro libX*.o
# creation of the pfo instrumented libX_pfo.a library
# for process X (X=1,2,..,N)
       st200ar libX_pfo.a libX_pfo.ro
# build of the main driver objects
       st200cc -c main*.c
# creation of the final executable including the instrumented
libraries
       st200cc -fb_create fb_data -o main_pfo.exe main*.o lib*_pfo.a
```

4.5.2 Collect feedback data

The second step is to gather the feedback data by executing each process, either on the simulator or on the board. During each execution, a frequency data file named <*name*>.instr0.##### is created (in this example <*name*> is fb_data). Unlike the gprof file-driven method, there is no need to create a single summary feedback file in the case of multiple feedback information files for the same process. To associate each data file to its process, it is convenient to create one directory fb_*X*_dir for each process *X* (*X*=1,2,..,N), and to move the data files into the associated directory. Finally, as the instrumented executable runs significantly slower than usual, it is recommended to use the fast mode of the simulator (if the program can execute on the simulator) to speed-up the execution. Here is an example of how to run and collect feedback information for process X with two different sets of arguments (X_args1 and X_args2):

Create the feedback data directory fb_X_dir for process X.

```
# run X process (for both X_args1 and X_args2 arguments) in ST231
# little-endian mode using the fast mode of the simulator
st200xrun -c st200sp -t st231fastsimle -e main_pfo.exe -a X_args1
st200xrun -c st200sp -t st231fastsimle -e main_pfo.exe -a X_args2
```

Put the collected feedback data files in the data directory for process X by copying all fb_data.instr0.##### files to the directory fb_X_dir.

After this step, we have a set of feedback data for each process, placed in the corresponding data directory.

4.5.3 Re-compile

The third and last step is to re-compile all modules adding the -fb <*dir*>/<*name*> option, that tells the compiler to annotate the code, using the information gathered in all feedback files <*name*>.inst0.* in <*dir*>. These annotations occur at a very high level in the compilation back-end process and the compiler uses them in later optimization phases such as if-conversion and instruction scheduling. This also affects the I-cache optimization phase, as the feedback frequencies are also seen by the I-cache optimizer and used to make decisions on code reordering.

Here are the build commands for this last step:

```
# optimized compilation with feedback annotations
# of the libX*.c sources for process X (X=1,2,..,N)
        st200cc -O2 -fb fb_X_dir/fb_data -c libX*.c
# Intermediate link in relocatable mode
# (indirectly using the feedback annotations)
        st200cc -O2 -r -nostdlib -o libX.ro libX*.o
# creation of the I-cache optimized libX.a library
# for process X (X=1,2,..,N)
        st200ar libX.a libX.ro
# build of the main driver objects
        st200cc -c main*.c
# creation of the final executable
# (with I-cache optimizations turned off)
        st200cc -o main.exe main*.o lib*.a --icache-opt=off
```

4.6 Advanced I-cache optimization features

In some cases, it may happen that even using sophisticated I-cache optimization methods like gprof file-driven optimization or PFO is not sufficient to successfully resolve all I-cache miss issues in the context of an application organized in libraries.

This is because the I-cache optimization phase of each library is performed in relocatable mode (`-r` option). In this mode, calls to functions which are external to the module are not taken into account by the I-cache optimizer (such functions are seen as unresolved symbols to be resolved at final link-time). This occurs either in the case of inter-dependencies between modules, or in the context of standard library calls (because of the use of the `-nostdlib` option at module link-time).

The solution in both cases is to make the code of the external functions visible to the **binopt** tool. For inter-dependencies between modules, there is access to the source, so we can duplicate the external functions that are causing the I-cache conflict and embed them in the module, so that they are seen by the I-cache optimizer. In the case of conflicts with standard library functions, there is no choice but to link the module with the standard library containing the conflicting entry points.

However, both of these methods may introduce multiple symbol definitions at final link time. A generic solution is to localize the duplicated symbols in each module library. This can be achieved by creating a text file containing all symbol (function) names that need to be localized, and passing this symbols file to the **st200objcopy** utility.

For example, to localize the function symbols `foo` and `bar` in `libX.a`, the following procedure can be applied:

List the symbols file `sym_list_file` to be passed to **st200objcopy**, as shown in *Figure 1*.

Figure 1. sym_list_file

```
foo
bar
```

```
# make all symbols of sym_list_file local in libX.a
> st200objcopy --localize-symbols sym_list_file libX.a
```

Caution: In the case of C++ code, the mangled symbol names must be passed to **st200objcopy**.

5 Results on real user application code

The test-case application chosen for optimization was the DVD/ACC audio processing test-suite (2004/11/15 CVS snapshot). It is composed of a main system, which drives several audio processes (either decoders, coders or post-processings).

A series of experiments were run on this test application in order to apply the I-cache optimization features and methods described in *Chapter 4: I-cache optimization techniques*.

The following experiments were run:

- `Exp-ref`: reference run without any I-cache optimization,
- `Exp-reloc`: run with libraries built in relocatable mode, but no binary optimization,
- `Exp-static`: run with static I-cache optimization performed on each process library,
- `Exp-pg`: dynamic gprof file-driven I-cache optimization (`-pg`) run,
- `Exp-pfo`: dynamic Profiling Feedback optimization (PFO) run.

Each experiment contained 26 executions. In each execution, the main driver was given a different set of arguments, such that each audio process was launched separately.

5.1 Preliminary analysis

The preliminary study was to determine:

- which of the 26 processes are subject to I-cache related performance issues,
- the performance gain to be expected using I-cache optimization.

The preliminary study focused on the non I-cache-optimized reference experiment `Exp-ref`, and the number of I-cache miss cycles due to conflicts and the total cycles for each process. This data provided an estimate of the upper bound on the performance gain in cycles (with the approximation of the cost of one I-cache miss on a given ST200 board to 150 cycles):

```
cycles_gain_bound(%) = 100 x (150 x conflicts_nb) / total_cycles
```

Table 1 presents the results of this preliminary analysis.

Table 1. Preliminary analysis

Bench_ID	conflicts	cycles_gain_bound(%)
DE	0	0.00
WMA	1954	2.45
aac	5095	11.12
ac3	2144	13.83
bassmgt	0	0.00
csii	109	1.04
dcremove	0	0.00
ddce	7558	6.65
dts	1621	6.09

Table 1. Preliminary analysis (continued)

Bench_ID	conflicts	cycles_gain_bound(%)
equalizer	0	0.00
kokvc	12	0.54
lpcm	0	0.00
lpcm-2	50	1.04
mix	0	0.00
mlp_decoder	73	0.14
mp2	10256	14.64
mp2e	183	0.14
mpeg1-layer3	5739	12.31
mpeg1-layer3_encoder	49578	19.01
pitchshift	10	0.03
plii	0	0.00
tsxt	58	0.99
resamplex2	1	0.05
resamplex2-2	0	0.00
resamplex2-3	9	0.39
sfc	76	0.31
tsxt	58	0.99

For 18 of the 26 processes, the overall speed-up to be hoped for by applying any code reordering method is 1% or less. Obviously, there is no point trying to optimize the code placement for these processes. For the 8 remaining processes, the potential gain in cycles is significant (it reaches 19% for the Mpeg1-Layer3 encoder).

For this reason, we focus on these 8 processes for which a reasonable speed-up can be expected. In fact, including the other 16 processes would mix up the results and make their analysis more difficult. There may be fluctuations in the I-cache miss results for these processes also, but these have an almost negligible effect on the total cycles. In fact, these fluctuations are just a spurious effect of differences of code alignment between two runs, leading to different numbers of I-cache miss cycles when the code is loaded for the first time (cold misses effects).

The following sections compare the performance results of the five experiments for the 8 key-processes, presenting for each of them a comparative table of total cycles, I-cache miss cycles, and I-cache miss conflicts and the corresponding speed-up.

5.2 Comparison of exp-ref and exp-reloc

The results of the comparison between `exp-ref` and `exp-reloc` are given in *Figure 2*.

Figure 2. Comparison of exp-ref and exp-reloc

```
Reference:      Exp-ref
Test:           Exp-reloc
Cycles:
Benchmark_ID                Reference            Test     Improvement (percent)
-------------------------------------------------------------------------------
WMA                          11942170        11932442         0
aac                           6872679         6297471         9
ac3                           2325365         2040569        14
ddce                         17037586        17028546         0
dts                           3989442         3968949         1
mp2                          10511204         9919147         6
mpeg1-layer3                  6994635         7333064        -5
mpeg1-layer3_encoder         39110908       340029825        15
-------------------------------------------------------------------------------
Sum (8)                      98783989        92523170         7
-------------------------------------------------------------------------------
A.Mean (8)                12347998.62     11565396.25         7
-------------------------------------------------------------------------------
G.Mean (8)                    8840300         8435233         5

I-cache miss Cycles :
---------------------
Benchmark_ID                Reference             Test     Improvement (percent)
-------------------------------------------------------------------------------
WMA                            754528           800763        -6
aac                           1072906           432191       148
ac3                            443067           142332       211
ddce                          1826132          1981312        -8
dts                            360210           211879        70
mp2                           1891491          1648451        15
mpeg1-layer3                  1372876          1585429       -13
mpeg1-layer3_encoder          9613485          4407169       118
-------------------------------------------------------------------------------
Sum (8)                      17334695         11209526        55
-------------------------------------------------------------------------------
A.Mean (8)                 2166836.87       1401190.75        55
-------------------------------------------------------------------------------
G.Mean (8)                    1248142           835821        49

Conflicts :
-----------------------
Benchmark_ID                Reference             Test      Improvement (percent)
-------------------------------------------------------------------------------
WMA                              1954             2486       -21
aac                              5095             1120       355
ac3                              2144              259       728
ddce                             7558             7958        -5
dts                              1621              699       132
mp2                             10256             8341        23
mpeg1-layer3                     5739             7008       -18
mpeg1-layer3_encoder            49578            23981       107
-------------------------------------------------------------------------------
Sum (8)                         83945            51852        62
-------------------------------------------------------------------------------
A.Mean (8)                   10493.12          6481.5        62
-------------------------------------------------------------------------------
G.Mean (8)                       5437             2943        85
```

This shows that just using a relocatable link before creating each library leads to a 5% speed-up in the geometric mean of total cycles and 49% in I-cache miss cycles.

Furthermore, a dramatic improvement in I-cache miss cycles is observed for some processes. For instance 211% for the ac3 decoder, 148% for the aac decoder, and 118% for the Mpeg1-Layer3 encoder.

Nevertheless, some regressions are also encountered, but they remain reasonable: at most 13% in I-cache miss cycles (5% in total cycles) for the Mpeg1-Layer3 decoder. These regressions can be explained by the fact that, in the reference experiment without any specific I-cache optimization, the code placement is arbitrary, boosting some processes at the expense of the others.

5.3 Comparison of exp-reloc and exp-static

The results of the comparison between `exp-reloc` and `exp-static` are given in *Figure 3* and *Figure 4*.

Figure 3. Comparison of exp-reloc and exp-static page 1 of 2

```
Reference:      Exp-reloc
Test:           Exp-static
Cycles :
--------

Benchmark_ID                Reference            Test      Improvement (percent)
-------------------------------------------------------------------------------
WMA                          11932442        11957632         0
aac                           6297471         6338792        -1
ac3                           2040569         2024126         1
ddce                         17028546        16309685         4
dts                           3968949         4021699        -1
mp2                           9919147         9226267         8
mpeg1-layer3                  7333064         6805125         8
mpeg1-layer3_encoder         34002982        35497476        -4
-------------------------------------------------------------------------------
Sum (8)                      92523170        92180802         0
-------------------------------------------------------------------------------
A.Mean (8)                11565396.25     11522600.25         0
-------------------------------------------------------------------------------
G.Mean (8)                    8435233         8295651         2

I-cache miss Cycles :
---------------------

Benchmark_ID                Reference            Test      Improvement (percent)
-------------------------------------------------------------------------------
WMA                            800763          780023         3
aac                            432191          458098        -6
ac3                            142332          104291        36
ddce                          1981312         1287176        54
dts                            211879          321049       -34
mp2                           1648451          823650       100
mpeg1-layer3                  1585429         1184931        34
mpeg1-layer3_encoder          4407169         5955625       -26
-------------------------------------------------------------------------------
Sum (8)                      11209526        10914843         3
-------------------------------------------------------------------------------
A.Mean (8)                1401190.75      1364355.37         3
-------------------------------------------------------------------------------
G.Mean (8)                     835821          739581        13
```

Figure 4. Comparison of exp-reloc and exp-static page 2 of 2

```
Conflicts :
---------------------

Benchmark_ID                Reference            Test      Improvement (percent)
-------------------------------------------------------------------------------
WMA                             2486            2051        21
aac                             1120            1745       -36
ac3                              259             183        42
ddce                            7958            4850        64
dts                              699            1298       -46
mp2                             8341            3759       122
mpeg1-layer3                    7008            4341        61
mpeg1-layer3_encoder           23981           32343       -26
-------------------------------------------------------------------------------Sum
(8)                           51852           50570         3
-------------------------------------------------------------------------------
A.Mean (8)                   6481.5         6321.25          3
-------------------------------------------------------------------------------
G.Mean (8)                      2943            2613        13
```

These results show that enabling the static I-cache optimizer at module link-time allows a further improvement of 2% in total cycles (geometric mean), with a 13% global improvement in the geometric mean of the I-cache miss-cycles.

But even though some modules have very high I-cache miss improvement (100% for the mp2 decoder or 54% for the ac3 encoder (ddce)), an increase in I-cache miss cycles is observed for 3 of them (reaching 34% for the dts decoder and 26% for the Mpeg1-Layer3 encoder).

These fluctuations highlight the fact that, in this example, static I-cache optimization is not accurate enough to significantly improve the I-cache miss cycles numbers for all process libraries. However, compared to the non-optimized run, the ST200 static I-cache optimization method still allows a performance gain of 7% in geometric mean for the main set of processes.

5.4 Comparison of exp-static and exp-pg

In *Figure 5* and *Figure 6*, the gprof file-driven I-cache optimization method (`exp-pg`) is compared to the static I-cache optimization.

Figure 5. Comparison of exp-static and exp-pg page 1 of 2

```
Reference:      Exp-static
Test:           Exp-pg

Cycles :
--------

Benchmark_ID                Reference            Test      Improvement (percent)
-------------------------------------------------------------------------------
WMA                          11957632        11918671         0
aac                           6338792         6251118         1
ac3                           2024126         1994004         2
ddce                         16309685        16392840        -1
dts                           4021699         3956163         2
mp2                           9226267         9045333         2
mpeg1-layer3                  6805125         6896797        -1
mpeg1-layer3_encoder         35497476        33781396         5
-------------------------------------------------------------------------------
Sum (8)                      92180802        90236322         2
-------------------------------------------------------------------------------
A.Mean (8)                11522600.25     11279540.25         2
-------------------------------------------------------------------------------
G.Mean (8)                    8295651         8193108         1

I-cache miss Cycles :
---------------------

Benchmark_ID                Reference            Test      Improvement (percent)
-------------------------------------------------------------------------------
WMA                            780023          779295         0
aac                            458098          390755        17
ac3                            104291           82395        27
ddce                          1287176         1285075         0
dts                            321049          250877        28
mp2                            823650          509903        62
mpeg1-layer3                  1184931         1138043         4
mpeg1-layer3_encoder          5955625         4281857        39
-------------------------------------------------------------------------------
Sum (8)                      10914843         8718200        25
-------------------------------------------------------------------------------
A.Mean (8)                 1364355.37         1089775        25
-------------------------------------------------------------------------------
G.Mean (8)                    739581          613616        21
```

Figure 6. Comparison of exp-static and exp-pg page 2 of 2

```
Conflicts :
---------------------

Benchmark_ID                 Reference            Test      Improvement (percent)
-------------------------------------------------------------------------------
WMA                               2051             963       113
aac                               1745            1398        25
ac3                                183              23       696
ddce                              4850            4599         5
dts                               1298             695        87
mp2                               3759            1459       158
mpeg1-layer3                      4341            5018       -13
mpeg1-layer3_encoder             32343           23684        37
-------------------------------------------------------------------------------
Sum (8)                           50570           37839        34
-------------------------------------------------------------------------------
A.Mean (8)                      6321.25         4729.87        34
-------------------------------------------------------------------------------
G.Mean (8)                        2613            1427        83
```

These results show that although the overall number of I-cache miss cycles decreases by about 21%, the use of the gprof file-driven dynamic I-cache optimization only leads to an additional improvement of about 1% in total cycles.

By looking more closely at the I-cache miss numbers, it can be seen, however, that most of the processes benefit from the use of profile data in terms of I-cache miss cycles improvement, with notable speed-up peaks of 62% for the mp2 decoder and 39% for the Mpeg1-Layer3 encoder. Nevertheless, the processes for which an important decrease in I-cache miss conflicts numbers is observed (ac3, dts, mp2) already have small numbers, so that the effect in terms of overall cycles is not very large. The only exception is for the Mpeg1-Layer3 encoder, where a 5% additional speed-up is achieved.

5.5 Comparison of exp-static and exp-pfo

The results of the comparison between `exp-static` and `exp-pfo`, to highlight the effect of the PFO method are given in *Figure 7* and *Figure 8*.

Figure 7. Comparison of exp-static and exp-pfo page 1 of 2

```
Reference:      Exp-static
Test:           Exp-pfo

Cycles :
--------

Benchmark_ID                Reference            Test      Improvement (percent)
-------------------------------------------------------------------------------
WMA                          11957632        11790783         1
aac                           6338792         6056219         5
ac3                           2024126         1974770         2
ddce                         16309685        16371656         0
dts                           4021699         3843128         5
mp2                           9226267         9810667        -6
mpeg1-layer3                  6805125         7085381        -4
mpeg1-layer3_encoder         35497476        33339259         6
-------------------------------------------------------------------------------
Sum (8)                      92180802        90271863         2
-------------------------------------------------------------------------------
A.Mean (8)                11522600.25     11283982.87         2
-------------------------------------------------------------------------------
G.Mean (8)                    8295651         8206009         1

I-cache miss Cycles :
---------------------

Benchmark_ID                Reference            Test      Improvement (percent)
-------------------------------------------------------------------------------
WMA                            780023          662168        18
aac                            458098          359008        28
ac3                            104291           80536        29
ddce                          1287176         1256961         2
dts                            321049          118205       172
mp2                            823650          375140       120
mpeg1-layer3                  1184931         1244956        -5
mpeg1-layer3_encoder          5955625         3974401        50
-------------------------------------------------------------------------------
Sum (8)                      10914843         8071375        35
-------------------------------------------------------------------------------
A.Mean (8)                 1364355.37      1008921.87        35
-------------------------------------------------------------------------------
G.Mean (8)                     739581          519194        42
```

Figure 8. Comparison of exp-static and exp-pfo page 2 of 2

```
Conflicts :
---------------------

Benchmark_ID            Reference       Test            Improvement (percent)
-------------------------------------------------------------------------------
WMA                          2051            1447        42
aac                          1745             899        94
ac3                           183              67       173
ddce                         4850            4341        12
dts                          1298             132       883
mp2                          3759             832       352
mpeg1-layer3                 4341            4929       -12
mpeg1-layer3_encoder        32343           22398        44
-------------------------------------------------------------------------------
Sum (8)                     50570           35045        44
-------------------------------------------------------------------------------
A.Mean (8)                6321.25         4380.62        44
-------------------------------------------------------------------------------
G.Mean (8)                   2613            1210       116
```

The use of the profiling feedback optimization method has a similar impact on the overall performance compared to the gprof file-driven I-cache optimization method, only gaining 1% in total cycles, but with a significant speed-up of 42% in I-cache miss cycles and 116% in conflicts (both in geometric mean).

Looking in detail at the total cycles figures, two processes show regressions (4% for the Mpeg1-Layer3 decoder and 6% for the mp2 decoder). This explains why the overall gain is only 1%.

It is important to bear in mind that the PFO method is performed at compile-time, and thus affects not only the binary optimization phase, but also the code generation. This explains why the overall performance change does not completely mirror the change in I-cache miss cycles.

In summary, the PFO experience still produces the best results in terms of performance. Compared to the reference run (without any I-cache optimization), for the main set of applications it achieves a speed-up of 140% in I-cache miss-cycles (geometric mean) which leads to a gain of more than 8% in total cycles. For the whole set of processes, the overall speed-up in total cycles is more than 5%.

5.6 Special features usage

This section focuses on the AAC encoder process, because it exposes specific I-cache miss cycle issues, which cannot easily be fixed by performing any of the global binary optimization methods presented in the previous sections. Because it needed to be treated differently, this process was not part of the 26 runs previously analyzed.

Compared to the reference run without any I-cache optimization, a static I-cache optimized run of the AAC encoder process produces the results shown in *Figure 9*.

Figure 9. AAC encoder results

```
Cycles :
--------

Benchmark_ID            Reference       Test            Improvement (percent)
-------------------------------------------------------------------------------
aace                    22130124        30628473            -28

I-cache miss Cycles :
---------------------

Benchmark_ID            Reference       Test            Improvement (percent)
-------------------------------------------------------------------------------
aace                    4112370        12606457            -67
```

A dramatic increase of almost 70% of I-cache miss cycles is observed: in fact, after static I-cache optimization more than 40% of the total cycles are due to I-cache misses.

The analysis of the I-cache miss profile graph (obtained by using the **st200gprof** toolset utility) for the AAC encoder process clearly illustrates that the predominating reason for this issue is a conflict between one application function and some functions from the low-level floating-point runtime (part of the `libgcc.a` runtime library).

This is one of the main points discussed in the *Section 4.6: Advanced I-cache optimization features on page 12*, namely the occurrence of I-cache miss conflicts due to the existence of standard library calls in the application code. But in this specific case, it is even more difficult, because these calls are not explicit, but compiler generated (for floating-point computations, the compiler generates the calls to the low-level runtime operators).

As described in *Section 4.6*, the solution to this issue is to add the `libgcc.a` library to the relocatable link command of the AAC encoder process, so that the problematic library entry points are resolved by that link, and are therefore visible to the binary optimization tool. These symbols can have multiple definitions (because they may be used in other modules or even in the main system driver), so it is necessary to localize them in the library using the following command:

`st200objcopy --localize-symbols <sym_`*`list_file`*`> <lib_`*`file`*`>`

A simple way to derive <`sym_`*`list_file`*> is to use the listing of the duplicate symbol errors, coming from the linker.

After having followed this procedure, the new run of the AAC encoder process with static I-cache optimization gives the following updated results as shown in *Figure 10*, compared to the reference non I-cache-optimized run.

Figure 10. AAC encoder results after libgcc.a library is added

```
Cycles :
--------

Benchmark_ID            Reference       Test            Improvement (percent)
-------------------------------------------------------------------------------
aace                    22130124        19216986            15

I-cache miss Cycles :
---------------------

Benchmark_ID            Reference       Test            Improvement (percent)
-------------------------------------------------------------------------------
aace                    4112370         1107886             271
```

The resolution method completely fixes the issue. The I-cache miss cycles are reduced to a very reasonable level of less than 6% of the total cycles (instead of 41%) which leads to a performance gain of 15% compared to the non-optimized process. The drawback is that there is an increase in the code size of the library, which in this particular case is negligible, but in other cases may be more significant.

6 Conclusion

This application note has presented different general optimization techniques to reduce the ST200 I-cache miss cycles in the context of an application organized as multiple libraries.

All these techniques were applied to the DVD/ACC audio codecs benchmark and produce a significant performance gain of more than 8% in total cycles for the main set of processes (5% for the overall set of processes). Almost 80% of this speed-up is achieved using static I-cache optimization. The remaining 20% is achieved using performance-feedback techniques.

7 Revision history

Table 2. Document revision history

Date	Revision	Changes
01-Jul-2009	E	Throughout: removed all references to ST220, which is no longer supported.
3-Dec-2007	D	Supports the ST200 R6.1 toolset. Updated the *Synopsis* to clarify the relevance of this application note with respect to the target processor core.
11-Sept-2007	C	Updated to support the ST200 R6.0 toolset. Updated *Section 4.4: gprof file-driven I-cache optimization* to replace **st200run** command line with **st200xrun** command line. Updated *Section 4.5.2: Collect feedback data on page 11* to replace **st200run** command line with **st200xrun** command line.
9-Nov-2006	B	Moved to new template. No technical changes.
24-Jun-2005	A	Initial release.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com