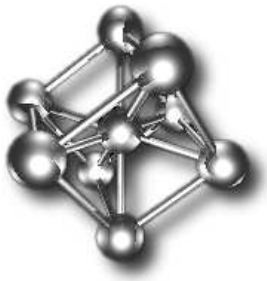
Atomium Record and Playback (ARP) User's and Reference Manual

[unofficial release]

The Atomium Club



Abstract

This document explains use of Atomium Record and Playback (ARP) package functionality used inside Atomium Analysis tools. This document describes the principles behind ARP, how it is handled and what is the data format it uses for storing execution information.

Copyright notice

Disclaimer of warranty

IMEC makes no representations or warranties, either express or implied, by or with respect to anything in this manual, and shall not be liable for any implied warranties of merchantability or fitness for a particular purpose or for any direct, indirect, special, or consequential damages.

Disclaimer for patent infringements

Although IMEC is not aware of any infringement of any patent or other third party proprietary right, to the extent permitted by law IMEC makes no warranties of any kind that the use of the software described in this manual, will not infringe any patent or other third party proprietary right.

Trademarks

Intel is a registered trademark of Intel Corp.
AMD is a registered trademark of Advanced Micro Devices, Inc.
Red Hat Linux is a registered trademark of Red Hat, Inc.
Linux is a registered trademark of Linus Torvalds.
Sun and Solaris are registered trademarks of Sun Microsystems, Inc.
HP and HP-UX are registered trademarks of Hewlett-Packard Company.
Macrovision and FLEX*lm* are registered trademarks of Macrovision Corporation.
Purify is a registered trademark of Rational Software Corporation.

All other products or services mentioned in this manual, other than ATOMIUM, are covered by the trademarks, service marks, or product names as designated by the companies that market those products.

The Atomium Club, IMEC/NES, atomium@imec.be
IMEC vzw – Register of Legal Entities Leuven VAT BE 0425.260.668 – Kapeldreef 75, B-3001 Leuven, Belgium

Contents

1 Introducing Atomium Record & Playback library

Atomium Record & Playback library (ARP) is the toolkit that is used by the users of Atomium tools in order to instrument the application code. ARP is a one of the important steps used during program analysis as a part of Atomium/Analysis . With the help of ARP, Atomium/Analysis supports what is called "kernel profiling". Kernel profiling is the process of obtaining accurate cycle cost estimations for critical parts of an application, typically inner loops, through instrumentation. This information can then be used to analyse the distribution of execution time accross the application, or to steer other optimisation tools.

The kernel profiling process involves the following steps::

1. marking kernels to be profiled;
2. preparing the program for Atomium;
3. adding kernel profiling callbacks;
4. instrumenting the program using Atomium/Analysis;
5. compiling and linking the instrumented program;
6. generating instrumentation results;

The generation of instrumentation results can be done by either running the compiled executable natively or by running it in an instruction set simulator.

Consequently, the process flow for kernel profiling is illustrated in Fig. 1 and Fig. 2.

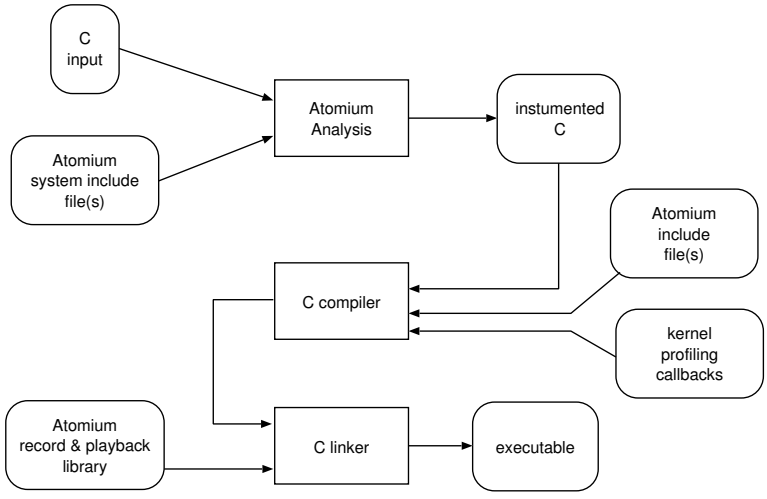


Figure 1: Atomium/Analysis kernel profiling process flow, part 1.

Step 1: Marking kernels to be profiled

It is beyond the capabilities of Atomium/Analysis to automatically detect which parts of an application are time-critical. Therefore, it is up to the user to indicate these critical parts,

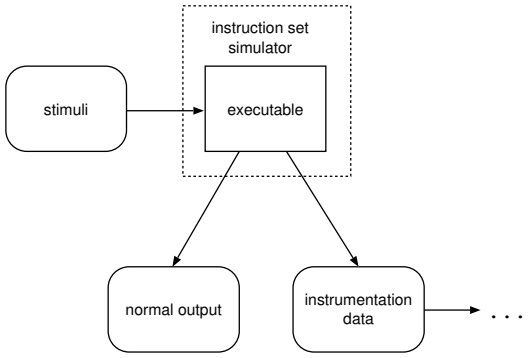


Figure 2: Atomium/Analysis kernel profiling process flow, part 2.

which are referred to as 'kernels'. This can be done in a very simple way by adding standard C labels, using a special naming convention: every part of the code that has a label starting with '`ato_kernel`' is considered to be a kernel for which profiling information must be obtained.

In our running example, we marked the `i`-loop in `reinit.c` as being a kernel. This is all we need to do in order to have Atomium/Analysis recognized this loop as a kernel loop.

There are some restrictions on where kernel labels can be placed, but these are explained in more detail in Section 2 on page 10.

Step 2: Preparing the program for Atomium

For using Atomium tools, one needs to adapt the original program such that it adheres to the Atomium C usage restrictions as they are outlined in Clean-C specifications[1]. There is an Eclipse-based interactive tool support available which can check each Clean-C rule compliance.

Step 3: Adding kernel profiling callbacks Atomium/Analysis

To implement the kernel profiling, Atomium/Analysis instruments the code with calls to the Atomium Record & Playback library, which offers an instrumentation framework that enables profiling of kernel sections. However, it does not perform the actual profiling, because that is highly context dependent. Typically, one wants to obtain cycle cost estimations for kernels by running the code on an instruction set simulator, or possibly even on the hardware itself. In order not to restrict the way the profiling is done, the Atomium Record & Playback library offers a flexible extension mechanism by means of callbacks. These callbacks, which have to be provided by the user, are responsible for performing the actual measurement. The Atomium Record & Playback library takes care of keeping track of the recorded measurements.

At least two instrumentation callbacks have to be provided: one that is executed each time when a kernel is entered, and one that is executed each time that a kernel is left. The callbacks

[1] Jean-Yves Mignolet, Roel Wuyts, "Embedded multiprocessor systems-on-chip programming", IEEE Software, Vol. 26, Issue 3, Pages 34-41, May-June 2009

are called with, among other things, a pointer to a data area in which the measurement results have to be dropped off.

To illustrate this, we add an extra source file containing two callbacks, `callbacks.c`:

```
/*---------------------- callbacks.c -----------------------*/
#include <time.h>
#include <atomium_rp.h>

void enterCallback(const arp_sid sid, void *data, void *context)
{
  /*
   * Save the current clock when we enter a kernel.
   */
  *(long*)data = clock();
}

void leaveCallback(const arp_sid sid, void *data, void* context)
{
  /*
   * Retrieve the current time, and subtract the time that we saved
   * in the data pointer when we entered the kernel. The difference is
   * stored back in the data pointer and will eventually be saved.
   */
  long enterTime = *(long*)data;
  *(long*)data = clock() - enterTime;
}
```

These callbacks simply use the system clock to measure the time spent in a kernel.[2] In the enter callback, which is called when the kernel loop in `process.c` is entered, the current time is taken and stored in the data area provided by the callback. The type of that data area is `long` in this case, but that is fully configurable as illustrated below. When the kernel loop is left again, the leave callback is called. The data pointer still points at the same data area in which the clock time was stored during the enter callback. The current clock time is retrieved again and the difference with the enter time is stored back into the data area. It is this final value which is eventually used to produce the profiling results (in an output file, see further).

Step 4: Instrumenting the program using Atomium/Analysis

Instrumenting an application for ARP is done by invoking Atomium/Analysis on the application's source files, making sure to instruct it what kinds of instrumentation it must perform (kernel instrumentation) [3]. Moreover, the tool has to be told which kernel callback functions have to be used for the profiling, and which data type the callbacks expect (`long` in this case):

```
% anl  --arp                                \
       --arp-enter-callback=enterCallback    \
       --arp-leave-callback=leaveCallback    \
       --arp-data-type=long                  \
       -I include -O outdir src callbacks.c
```

[2] In reality, one would typically retrieve a cycle counter value from the instruction set simulator in which the application is being simulated. Here we use the standard system clock only for illustration purposes.

[3] Atomium/Analysis has capabilities of doing many types of instrumentations e.g. array access counting

In doing so, we instruct the tool to instrument the `*.c` files found in the `src` directory and we also supply the file containing the callbacks, the names of the callback functions, and the data type to be used by the callbacks. The remaining options are the same as in the previous examples.

As an example, the instrumented version of the file `process.c` looks like this:

```
/*
 *  Generated by Atomium/Analysis 4.0.4+ (ndbg) [Rev. 25430]
 *  on Thu Oct  1 12:18:55 2009.
 *
 *  This file is copyrighted and/or licensed under the same
 *  conditions as the file from which it was derived.
 *
 *  You may want to think twice before editing this. But then
 *  again, you may not and you certainly don't have to.
 */

#ifndef __process_c
#define __process_c

#include "process.h"
#include "stdio.h"
#include <atomium_rp.h>

void process(int arr[], int upto, int max)
{
  int i;

  arp_enter(1, 2, 0);
  arp_enter_kernel(1, 1, 0);

  ato_kernel:
  for (i = 0; i < upto; ++i)
  {
    if (arr[i] < max)
    {
      arr[i] += 10;
    }

    printf("%d ", arr[i]);
  }

  arp_leave_kernel(1, 1);
  printf("\n");
  fflush(stdout);
  arp_leave(1, 2);
}

#endif /* __process_c */
```

One can see that Atomium/Analysis has added some calls to the Atomium Record & Playback library. Most notably, some `arp_enter_kernel()` and `arp_leave_kernel()` calls have been added. When the code is compiled and run next, the supplied enter and leave callbacks are called every time an `arp_enter_kernel()` or `arp_leave_kernel()` call is executed, respectively. More information on this can be found in Section 2.

Step 5: Compiling and linking the instrumented program

To compile and link the instrumented code, simply use a C++ compiler, tell it where the ATOMIUM header files are located, and link with the appropriate version of the ATOMIUM RECORD & PLAYBACK library library.

```
% cc outdir/src/*.c outdir/callbacks.c -Ioutdir -Ioutdir/include \
  -I<path-to-atomium-inc-dir> -L<path-to-atomium-lib-dir> -larp -o example
```

Step 6: Generating instrumentation results

At this point, the instrumented program can be run in exactly the same way as the original uninstrumented one. It will, however, next to performing its normal functions, generate an additional data file, which by default will be named `data.arp`:

```
% example
9 10 11 13 14 5 6 7 8 9


Note: The data gathered during this run are stored in the file ‘data.arp’.
```

Currently, there are no tools available for inspecting the obtained kernel profiling information. The information is primarily intended for use by other ATOMIUM tools.

For more information on using kernel profiling, refer to Section 2.

2 Using ATOMIUM/ANALYSIS: kernel profiling

One important functionality of ATOMIUM/ANALYSIS is adding kernel profiling instrumentation to applications, which allows to obtain accurate cycle time estimates for critical parts of the code (called kernels). Typically these kernels are inner loops. These estimates can then be used to steer the optimisations that are performed by other ATOMIUM tools,

As explained in Section 1 on page 5, ATOMIUM/ANALYSIS' kernel profiling process flow consists of six separate steps. Each of these is elaborated on in one of the following subsections.

2.1 Marking kernels to be profiled

As already explained in Section 1, ATOMIUM/ANALYSIS cannot detect critical parts in an application automatically. Therefore, it needs help from the user. The user often has a good idea of what the time-critical parts of an application are. If user is unaware of time-critical parts of an application, *execution time profiling* or other ATOMIUM/ANALYSISinstrumentations (namely *access count profiling*) may help in identifying these parts.

The user can mark time-critical parts of the code, called 'kernels', by adding simple `C` labels using a special naming convention: a label starting with '`ato_kernel`' is assume to identify such a kernel. On top of that, the user can request profiling information about less critical parts of the code, for instance to find out how the execution time is distributed over the application. In order to mark these sections for profiling, the same mechanism is used: adding a `C` label with a special naming convention. In this case, the label has to start with '`ato_profile`'.
An example of such a labeling is shown in the code below:

```
void filter_horizontally(const char image_in[Height][Width], char image_out[Height][Width])
{
 int x, y;
ato_profile_filter_image_hor:
 for (y = 0; y < Height; ++y)
 {
ato_kernel_filter_line_hor:
  for (x = 0; x < Width; ++x)
  {
   char left = x > 0 ?  image_in[y][x-1] :  image_in[y][x];
   char center = image_in[y][x];
   char right = x < Width - 1 ?  image_in[y][x+1] :  image_in[y][x];
   image_out[y][x] = (left + 2*center + right)/4;
  }
 }
}

void filter_vertically(const char image_in[Height][Width], char image_out[Height][Width])
{
 int x, y;
ato_profile_filter_image_vert:
 for (y = 0; y < Height; ++y)
 {
ato_kernel_filter_line_vert:
```

```
   for (x = 0; x < Width; ++x)
   {
    char top = y > 0 ?  image_in[y-1][x] :  image_in[y][x];
    char center = image_in[y][x];
    char bottom = y < Height - 1 ?  image_in[y+1][x] :  image_in[y][x];

    image_out[y][x] = (top + 2*center + bottom)/4;
   }
  }
 }

 int main()
 {
  int t;
  char image_in[Height][Width], image_tmp[Height][Width], image_out[Height][Width];

  for (t = 0; t < nImages; ++t)
  {
   input(image_in); /* Read an input image from some source.  */
   filter_horizontally(image_in, image_tmp);
   filter_vertically(image_tmp, image_out);
   output(image_out); /* Write the output image to some destination.  */
  }
  return 0;
 }
```

The code shows a simple image processing application that reads a series of input images, performs some filtering on them (first horizontally, then vertically), and writes the results to some output.

From a performance perspective, the inner loops that process a complete line of an image each time, are probably the most critical ones. A compiler mapping the code onto a processor will often be able to pipeline these inner loops to make optimal use of the available hardware. If an optimisation tool, such as ATOMIUM/MH, wants to perform any memory optimizations, it should be careful not to apply any transformations that may destroy the performance in these loops, and it should take into account the loops' durations when performing optimisations in their neighborhoud. That is why the inner loops have been labeled with `ato_kernel*` labels. ATOMIUM/ANALYSIS will recognize these labels and insert instrumentation code that allows measuring the duration of the marked loops.

In the example, the outer loops have also been labeled with `ato_profile*` labels, to request instrumentation for these loops too.

In general, not only loops can be labeled with `ato_kernel*` or `ato_profile` labels. Any `C` statement that can be labeled, can be marked as a kernel or a section to be profiled.

There are some restrictions, though, the main one being the fact that *kernel* sections should not be nested, not only statically, but also dynamically. E.g., it is not allowed to call a function from within a kernel section if that function contains a kernel section itself. For general profiled sections, there is no such limitation.

Another restriction is the fact that kernel or profiled sections should not occur inside functions that are or can be called recursively.

In addition, tools that have to use the resulting profiling information may have some limitations on the kind of kernel or profiled section constructs that they support (even though Atomium/Analysis does not have this restriction). In general it should be safe to add kernel or profiled section labels to loops and blocks.

Also, it is possible to mark a group of statements as being a kernel or profiled section by putting them inside a block {...} and labeling the block.

2.2 Preparing the program for Atomium

The first task in the process of using Atomium is to adapt the original program such that it adheres to the Atomium C usage restrictions as they are outlined in Clean-C specifications [4]. There is an Eclipse-based interactive tool support available which can check each Clean-C rule compliance.

For of some usage restriction violations, there may be a choice between changing input to Atomium on the one hand, and editing the code generated by it on the other (occasionally both may be called for). In general, the former is the better solution. This is because the Atomium run time library has been designed for maximal speed and easy automatic instrumentation, thus making it hard(er) to perform the instrumentation task manually.

2.3 Adding (kernel) profiling callbacks

To implement the kernel profiling, Atomium/Analysis makes use of the Atomium Record & Playback library. This library offers a generic instrumentation framework that is not bound to a particular implementation of the cycle count measurement. A description of the Atomium Record & Playback library API can be found in Section 6 on page 32.

A key concept in this library is the use of user-defined callbacks to perform the actual recording of cycle counts. The library takes care of calling the callbacks at the appropriate places *and* keeping track of the recorded values.

At least two callbacks have to be provided in order to be able to collect profiling information: a callback that is called whenever a kernel or profiled section is being entered and a callback that is called whenever such a section is being left again. Those callbacks are called with, among other things, a pointer to a data area in which the callback can store the result of the profiling.

Typically, the enter callback records the time (in the form of a cycle count) at the moment that the profiled (kernel) section is entered and stores it in this data area, and the leave callback also records the time, but stores the difference with the enter time, i.e., the elapsed time back into the data area. This is depicted graphically in Fig. 3.

The data area is shared between the enter and the leave callback, but *not* among different execution of the profiled section. I.e., there is no guarantee that the provided data area is the same for different executions of the profiled section, so it cannot be use to carry over information from one execution to a next one.[5]

[4] Jean-Yves Mignolet, Roel Wuyts, "Embedded multiprocessor systems-on-chip programming", IEEE Software, Vol. 26, Issue 3, Pages 34-41, May-June 2009

[5] The data area is obviously also not shared between different profiled sections.

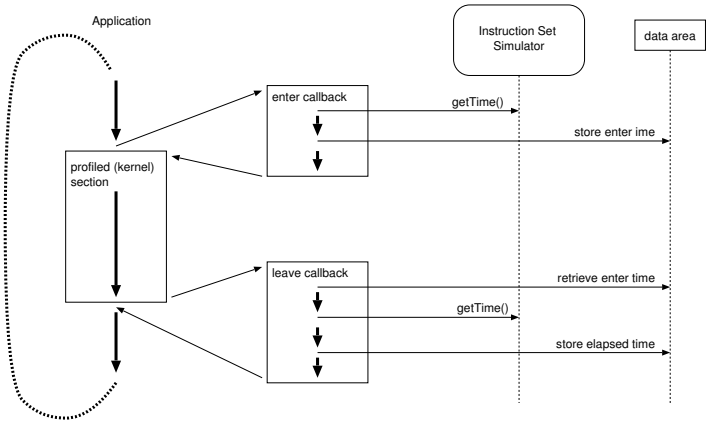


Figure 3: The flow of calling profiled section enter and leave callbacks.

The callbacks are also provided with a pointer to a user-defined context. This context is shared between all executions of all callbacks. It can be used to pass some additional information to the callbacks such as, for instance, a pointer to the instruction set simulator (ISS) in which the compiled code is simulated.

The following code shows an example of such an enter and leave callback:

```
void enterCallback(const arp_sid sid, void *data, void *context)
{
  /* Retrieve the current cycle count from the ISS. */
  Simulator *iss = (Simulator*)context;
  long *dataArea = (long *)data;
  *dataArea = get_current_cycle_count(iss);
}

void leaveCallback(const arp_sid sid, void *data, void *context)
{
  /*
   * Retrieve the current cycle count, and subtract the count that we
   * saved  in the data area when we entered the kernel. The difference is
   * stored back in the data area and will eventually be saved by the
   * ARP library.
   */
  Simulator *iss = (Simulator*)context;
  long *dataArea = (long *)data;

  long enterTime = *dataArea;
  *dataArea = get_current_cycle_count(iss) - enterTime;
}
```

The callbacks receive 3 arguments: a section id number (`sid`), a pointer to the data area for the section, and a context pointer. The section id is a unique number identifying the profiled (kernel) section for which the callback is called. In the example, this information is not used.[6]

The data pointer points at the private data area that we described above. In this case it points at a long integer. The type of the data area is configurable (see below). The third

[6]The section id is rarely needed. It is merely provided for debugging purposes.

argument point at some additional context information, which is also configurable. In this case it points at the instruction set simulator in which we run the compiled code. Below, it is illustrated how this can be configured.

The callbacks implement the typical profiling pattern: the enter time is recorded in the enter callback and the elapsed time is measured and stored in the leave callback.

2.4 Instrumenting the program using ATOMIUM/ANALYSIS

Instrumenting the application is done by invoking ATOMIUM/ANALYSIS on the application's source files, with the proper (kernel) profiling options.

The most important options for this kind of profiling are the following:

- **the desired data type for the profiling data area**
  By default, ATOMIUM/ANALYSIS assumes that the data type is unsigned integer. If another data type is desired, ATOMIUM/ANALYSIS has to be instructed which one to use.
- **the names of the enter and leave callbacks**
  In order to know what callbacks to call, ATOMIUM/ANALYSIS has to be told the names of these functions. It is not strictly necessary to supply these names. If no names are provided, ATOMIUM/ANALYSIS will instrument the code with some dummy placeholder names, which the user can then manually replace by the proper names.
- **a pointer to context information for the callbacks**
  If the callbacks need access to some external information, a pointer to this callback can be supplied at initialization time. ATOMIUM/ANALYSIS can be instructed which pointer to use.
- **the mode in which the application must be run**
  The ATOMIUM RECORD & PLAYBACK library support recording of profiling information in two modes: '`record_all`' and '`record_average`'. In `record_all` mode, every profiling sample is remembered and eventually saved. In `record_average` mode, only the averages are saved. A third mode, '`playback`', is available for playing back the recorded samples. More information about playing back profiling information can be found in Section 4 and Section 6. By default, average data are recorded.
- **the main function of the application**
  In this main function, ATOMIUM/ANALYSIS places the initialization and finalization calls for the ATOMIUM RECORD & PLAYBACK library. By default, it is assumed that `main()` is the main function.

For the example, the following command line options are used:

```
% anl  --arp                                \
       --arp-enter-callback=enterCallback   \
       --arp-leave-callback=leaveCallback   \
       --arp-data-type=long                 \
       --arp-data-context=IIS               \
       --arp-mode=record_average            \
       -I include -O outdir src callbacks.c
```

This instructs the tool to add `arp` instrumentation, which functions to use as callbacks, which data type to use for the profiling data, which context information to pass to the callbacks, and which mode to use. In this case, the `ISS` data context is assumed to be a pointer pointing at the instruction set simulator. The resulting output code looks as follows:

```
void filter_horizontally(const char image_in[200][320], char image_out[200][320])
{
  int x, y;

  arp_enter_profiled(1, 2, 0);
  ato_profile_filter_image_hor:
  for (y = 0; y < Height; ++y)
  {
   arp_enter_kernel(1, 1, 0);
   ato_kernel_filter_line_hor:
   for (x = 0; x < Width; ++x)
   {
    char left = x > 0 ?  image_in[y][x - 1] :  image_in[y][x];
    char center = image_in[y][x];
    char right = x < Width - 1 ?  image_in[y][x + 1] :  image_in[y][x];

    image_out[y][x] = (left + 2 * center + right) / 4;
   }
   arp_leave_kernel(1, 1);
  }
  arp_leave_profiled(1, 2);
}

void filter_vertically(const char image_in[200][320], char image_out[200][320])
{
  int x, y;

  arp_enter_profiled(1, 5, 0);
  ato_profile_filter_image_vert:
  for (y = 0; y < Height; ++y)
  {
   arp_enter_kernel(1, 4, 0);
   ato_kernel_filter_line_vert:
   for (x = 0; x < Width; ++x)
   {
    char top = y > 0 ?  image_in[y - 1][x] :  image_in[y][x];
    char center = image_in[y][x];
    char bottom = y < Height - 1 ?  image_in[y + 1][x] :  image_in[y][x];

    image_out[y][x] = (top + 2 * center + bottom) / 4;
   }
   arp_leave_kernel(1, 4);
  }
  arp_leave_profiled(1, 5);
}

int main(void)
{
  int t;
```

```
  char image_in[200][320], image_out[200][320], image_tmp[200][320];

  arp_add_callbacks(arp_long, enterCallback, leaveCallback, 0, 0, ISS);
  arp_start(1, 8, arp_record_average, 0, 0, 0, 0, 0);
  arp_enter(1, 8, 0);
  for (t = 0; t < nImages; ++t)
  {
   arp_enter(1, 7, 0);
   input(image_in);
   arp_enter(1, 3, 0);
   filter_horizontally(image_in, image_tmp);
   arp_leave(1, 3);
   arp_enter(1, 6, 0);
   filter_vertically(image_tmp, image_out);
   arp_leave(1, 6);
   output(image_out);
   arp_leave(1, 7);
  }
  arp_leave(1, 8);
  arp_finish();
  return 0;
}
```

One can see that Atomium/Analysis has inserted several calls to the Atomium Record & Playback library, which have been put in italics:

- Around every kernel section an `arp_enter_kernel()` and `arp_leave_kernel()` call pair have been added. These calls trigger the calling of the user-provided enter and leave callbacks, respectively, and trigger the recording of the obtained time measurements. The first parameter of these calls is a unique number identifying the thread in which the kernel is executed. For this simple example there is only one thread, so the number is always `1`. The second number is the unique identification number of the section. Every (kernel) section has a unique id, and obviously the enter and leave calls of a section have to use the same number. The enter call has a third parameter, which is only relevant for multi-threaded applications. Therefore, it is not discussed here. It is always `0` for single-threaded applications.

- Around every profiled section, a similar `arp_enter_profiled()` and `arp_leave_profiled()` call pair have been added. The behavior and parameters of these functions are exactly the same as for their kernel counterparts.

- At relevant places in the code, additional `arp_enter()` and `arp_leave_()` calls have been added. These are inserted to provide the Atomium Record & Playback library with additional run-time context information. In the example, these calls have been added to the `t` loop body in `main()`, around the calls to the filter functions, and in the body of `main()`. The arguments to these function calls are again similar to those of the kernel and profiled section equivalent.

  However, there is an important difference in behaviour: these calls do *not* trigger the enter and leave callbacks. They are only inserted to allow the Atomium Record & Playback library to keep track of the conditions under which a profiled (kernel) section

is called. For instance, if a function containing a kernel section is called from different places, each call is surrounded by an enter/leave pair with a unique section id. This allows the Atomium Record & Playback library to distinguish between different call sites, for instance.[7]

Note that it doesn't matter for the Atomium Record & Playback library whether or not the enter/leave calls are present in the code. The library records all context information that it is provided with. For tools using the recorded information however, it is vital that enough context information is provided. Atomium/Analysis knows where to insert the necessary calls in order to record sufficient context information.

- In the main function, some `arp` initialisation and finalisation calls have been added:
  - `arp_add_callbacks()`: this call make sure that the proper enter and leave callbacks are registered. It also registers the global context information for these callbacks (which is a pointer to the instruction set simulator in this case). The remaining two arguments are used to registers additional callbacks for multi-threaded applications. Those are not described here.
  - `arp_start()`: this call start the actual recording. The first parameter is the maximum number of threads in the application (which is 1 in this case); the second one is the largest section id number in the application (8 in this case). The third argument sets the mode in which the Atomium Record & Playback library is to be used: average recording mode. The remaining arguments can be used to install additional handlers for error handling and dealing with multi-threaded applications. Those are not described here.
  - `arp_finish()`: this call finalises the recording and causes the recorded data to be saved to a file.

For a complete description of the Atomium Record & Playback library API, refer to Section 6.

2.5 Compiling and linking the instrumented program

For this kind of profiling, a regular C compiler can be used for compiling and linking the instrumented program.

```
% cc outdir/src/*.c outdir/callbacks.c -Ioutdir -Ioutdir/include \
  -I<path-to-atomium-inc-dir> -L<path-to-atomium-lib-dir> -larp -o example
```

Users of 64-bit Linux machines need to be aware of the fact that the Atomium Record & Playback library is only provided in a 32-bit version. This implies that on such machines the native compiler and linker, which default to generating 64-bit code, must explicitly be told to produce a 32-bit executable instead by means of the `-m32` or `-melf_i386` options respectively:

```
% cc outdir/src/*.c outdir/callbacks.c -Ioutdir -Ioutdir/include \
  -I<path-to-atomium-inc-dir> -L<path-to-atomium-lib-dir> -m32 -larp -o example
```

[7] One can think of it as a kind of stack trace: the ids of the nested sections leading up to a profiled (kernel) section identify the followed path.

2.6 Generating instrumentation results

A program instrumented for (kernel) section profiling by ATOMIUM/ANALYSIS can be run in exactly the same way as the original uninstrumented one and will in general show exactly the same behaviour, except for these differences:

- In general, the program will run slower and consume more memory. How much slower and how much more memory highly depends on the instrumentation granularity and the recording mode. When kernels are located inside deeply nested loops and the recording mode is set to `record_all`, the impact of the instrumentation may be large. Under normal conditions, especially when average recording mode is used, the impact is limited, though. In extreme cases, the instrumented program may fail because too many data are recorded and memory is exhausted. In those cases, the instrumentation granularity should be reduced or the recording mode should be changed.
- When the program finishes, it generates an additional data file containing the recorded instrumentation data. By default, this file is called '`data.arp`', but another name can be chosen during instrumentation by means of the `--arp-data-filename` command line option.
- As with the access counting instrumentation, data may end up in other places in memory than is the case in the original uninstrumented program. Thisimplies that programs that are buggy, but happen to work by accident, may fail unexpectedly when instrumented.

After instrumentation, the generated data file '`data.arp`' can be used by other ATOMIUM tools that can make use of this information (currently only ATOMIUM/MH).

3 Multithreaded applications

Atomium/Analysis supports analysing multithreaded applications making use of the POSIX threads API. This section explains how to compile such applications and how to interpret the resulting access count reports.

3.1 Instrumenting multithreaded applications

When instrumenting a multithreaded application *for kernel profiling*, additional callbacks have to be provided to enable safe multithreaded execution. At least, a locking and unlocking callback have to be provided that allow the Atomium Record & Playback library to access its internal data structures in a thread-safe way. An example of such callbacks, using a `pthread` mutex, is shown below:

```
static void lockCallback(const void* data, void* context)
{
  pthread_mutex_t *mutex = (pthread_mutex_t *)context;
  pthread_mutex_lock(mutex);
}

static void unlockCallback(const void* data, void* context)
{
  pthread_mutex_t *mutex = (pthread_mutex_t *)context;
  pthread_mutex_unlock(mutex);
}
```

The mutex is passed to the `arp` library at initialisation time (see below). The data pointer passed to these callbacks points at the memory area that has to be protected. In this example, a global lock is used, so the data pointer is not used.

Optionally, extra callbacks can be provided that have to be called whenever a new thread is started or when a thread finishes, in case it would be needed for the profiling implementation to take into account thread spawning and joining. An example is shown below:

```
static void spawnCallback(const arp_tid tid, void** contextp)
{
  Simulator **issp = (Simulator**)contextp;

  /* Clone the simulator from the parent thread */
  *issp = clone_simulator(*issp)
}

static void joinCallback(const arp_tid sid, void** contextp)
{
  Simulator **issp = (Simulator**)contextp;

  /* Clean up the simulator that we created for the child thread.
     There is no need to restore the pointer to the parent thread's simulator
     manually as it is restored automatically by the ARP library */
  cleanup_simulator(*issp);
}
```

The context pointer that is passed to the enter and leave callbacks is related to the one passed to the spawn and join callbacks. Actually, it is the address of the enter/leave context that is passed to the spawn/join callbacks. This allows the spawn/join callbacks to change the context that is passed to the enter/leave callbacks depending on the thread, in case that would be necessary for the profiling implementation.

Atomium/Analysis can be instructed to use these extra callbacks and context pointers via the appropriate command line parameters. For the example, the resulting code will look like this:

```
int main()
{
  ...
  pthread_mutex_t mutex;
  pthread_mutex_init(&mutex, NULL);

  arp_add_callbacks(arp_long, enterCallback, leaveCallback, spawnCallback, joinCallback, &ISS);
  arp_start(1, 8, arp_record_average, 0, 0, lockCallback, unlockCallback, &mutex);
  ...
}
```

Note that the creation and initialisation of the mutex has to be provided by the user. Atomium/Analysis only add the mutex as an argument to `arp_start()` when it told to use this as a locking callback. The mutex must be present in the code already, or must be added afterwards.

3.2 Compiling and linking multithreaded applications

When compiling a multithreaded application *that has been instrumented for kernel profiling*, no special compiler options have to be added (except, of course, the options necessary to link with the pthread library).

If you use C++ compilers, they typically require the user to specify certain command line options in order for the compiler to generate thread safe code. Please consult your compiler's documentation for up to date information.

When using the g++ compiler, it must have been configured with the `--enable-threads` option. This is the case for the default compiler as shipped on Red Hat Linux systems, but users who compile their own compiler need to take care of this themselves.

3.3 Performance

For the Atomium Record & Playback library library, there is no performance penalty when running single-threaded applications. Atomium Record & Playback library library has been implemented such that it does not incur any multithreading related overhead for non-multithreaded applications, or even for multithreaded ones while only a single thread is active.

The impact of multithreading on the ATOMIUM RECORD & PLAYBACK library is much smaller, because the internal `arp` data are separate for each thread and therefore do not require locking.[8]

Please note that running an ATOMIUM RECORD & PLAYBACK library instrumented code may lead to the discovery of race conditions that have always been present, but that by pure coincidence have never been triggered so far.

3.4 Limitations

3.4.1 Cancelling the main thread

The only operation permitted by POSIX threads that is not supported by ATOMIUM/ANALYSIS is that of cancelling the main thread by means of `pthread_cancel()`. In contrast, calling `pthread_exit()` from within the main thread *is* supported.

An attempt to cancel the main thread will result in the following warning message being issued on the application's `stderr` stream:

```
==================================================
Warning: Cancelling the main thread is not supported.
==================================================
```

The cancel request will then be honoured, but from that point onwards results are unpredictable.

3.4.2 Trailling threads at program exit

An application instrumented by ATOMIUM/ANALYSIS will generate its data file either when the main thread returns or executes an `exit()` system call, or when the last remaining thread exits in case the main thread has called `pthread_exit()` earlier on. In case multiple threads are still active when generation of the data file starts, the program as a whole will be forcedly aborted as soon as the data file is ready, and a warning similar to the following is generated on the application's `stderr` stream:

```
Warning: There still is/are 1 other thread(s) hanging around out there.
         It/They would behave very unpredictably from this point onwards.
         Exiting here and now so as to preserve your sanity.
```

The purpose of this is to prevent one of the remaining threads to cause havoc and likely also a core dump due to the ATOMIUM/ANALYSIS data structures already having been closed down.

[8] The ATOMIUM RECORD & PLAYBACK library only uses locking at the time that threads are spawned and joined.

3.4.3 Kernel profiling limitations

The kernel profiling functionality of ATOMIUM/ANALYSIS has several additional restrictions w.r.t. to multithreading:

- When a thread is spawned in section (bounded by a `arp_*enter()/arp_*leave()` pair), the thread must be joined again before that section ends.
- If a function can be called from different threads, that function cannot be instrumented. As a workaround, the function can be duplicated.
- ...

3.5 Platform dependencies

ATOMIUM/ANALYSIS' support for multithreaded applications is subject to the following platform depencencies:

- On Solaris, using Solaris threads is not supported. Using POSIX threads on Solaris is supported, however.
- On HP-UX 11, only the DCE thread implementation is supported, the CMA one is not.

4 Understanding ATOMIUM kernel profiling

This purpose of this section is to briefly explain the principles behind ATOMIUM kernel profiling, the constraints that had to be taking into account during its design, and some of its limitations.

4.1 Requirements

The following requirements and constraints have been taken into account in the design of the kernel profiling:

- The main goal of kernel profiling is to obtain accurate cycle time estimates for so-called 'kernels', typically by means of a platform or instruction set simulator. These estimates should allow optimisation tools such as ATOMIUM/MH to perform more intelligent optimisations. In the future, additional cost estimates may be required, and therefore, the kernel profiling framework should not exclude future extensions.

- Automatic identification of kernels is *not* a requirement: it would be very hard for a tool to decide on the level of granularity to use for the profiling. The definition of what constitutes a 'kernel' is highly context-dependent. In general, a kernel should contain a sufficiently large set of operations on which a compiler can perform its typical (scalar) optimisations like pipelining etc. For instance, some platforms may have a large number of processing units that can operate in parallel and hence the compiler should have a sufficiently large number of operations to map on these processing units in order to use the hardware efficiently. Therefore, the compiler may have to unroll an inner loop in order to keep the hardware busy. On the other hand, a kernel should not be too large, such that high-level optimisation tools (like the ATOMIUM optimisation tools) still have some freedom to apply optimising transformations.

  Since the definition on the desired granularity of a kernel is so context dependent, the user should be able to mark the kernels manually. However, the overhead for the user to mark the code sections that must be profiled, should be minimal. Ideally, the source code should not have to be modified, but small modifications are considered acceptable.

- The instrumentation for the profiling must be source based, i.e., the instrumented application should consist of the original source code, augmented by instrumentation instructions (such as function calls).

  The reason is that the instrumented source code must be compiled for a target platform by an external compiler, because generating target-specific machine code is far beyond the scope of any ATOMIUM tools.

- The code instrumentation should not impose the use of a certain profiling implementation. The framework should offer sufficient flexibility to hook up custom profiling implementations.

- The code instrumentation should affect the cost measurements during profiling runs as little as possible. Since the profiling will typically happen in an instruction set profiler, the instrumentation instructions should be simple and sparse in order not to skew the

measurements too much. This means that the profiling API should be simple too. Ideally, it should consist of function calls with few arguments and the arguments should be simple as possible (E.g.no string arguments as they could affect the measurements).

- Although the main goal of the profiling is to steer optimisation tools, it should also be possible to use the profiling information in a high-level simulation environment even *after* optimising transformations by optimisation tools. In this way, the effectiveness of the optimisations can be verified relatively quickly for different optimisation alternatives (for instance, does the application achieve the estimated performance after parallelisation?). This means that not only recording functionality is needed, but also playback functionality.

- It should be possible to record profiling information at different levels of granularity, ranging from recording each individual sample to recording only averages. Also, it should be possible to distinguish profiling data obtained via different execution paths. For instance, if a kernel is executed from different places, it should be possible to record separate for each place.

4.2 Design decisions

The requirement that the user should be able to mark kernels in a simple way, without having to modify the application code too much, is tackled by the use of C labels with a special naming convention (see Section 2).

The fact that it should be possible to replay profiling information even after (high-level) source code transformations has some implications. In general, it is very hard to relate profiling information that was obtained from one version of an application to another, transformed, version of the application. The kernels in the original code to which the profiling data relate, may be transformed beyond recognition in the transformed version.

In order to make it possible to relate kernels in the original and transformed code, they should be somehow marked in the code and those markings should be able to survive the transformation. One obviously thinks of the user-provided kernel marking labels to accomplish this: those are present in the source code and are likely to survive most high-level transformations.

However, this is not sufficient: they would allow the identification and relating of kernels before and after transformations, but that is not enough. During recording, a lot of context information is saved (for instance, the call trace leading up to a kernel when it is executed), and also this context information must be related to the transformed code.

Therefore, it was decided that any kernel profiling instrumentation added for recording should remain in the application during transformations. This instrumentation is relatively unintrusive and hence it will not prevent the high-level transformations from being applied. This may require special precautions to be taken by the transformation tools in order to keep the instrumentation consistent, but in general, the effort should be limited.

Later on, when the code has been optimized and verified, and the instrumentation is not needed any longer, it can be removed from the (optimised) application again.

In order for this flow to be possible, the instrumentation must only use C constructs, because Atomium in general does only support C.

Note that this flow differs from the (traditional) profiling flow that is used for the access count profiling. During access count profiling, the code is instrumented with C++ constructs and compiled with a C++ compiler. The resulting code can only be used to obtain access count profiling information and cannot be parsed by ATOMIUM again (because it is C++). In the case of kernel profiling, the profiled code can/must be used during the rest of the optimisation trajectory.

The two flows and their relations are depicted in Fig. 4.

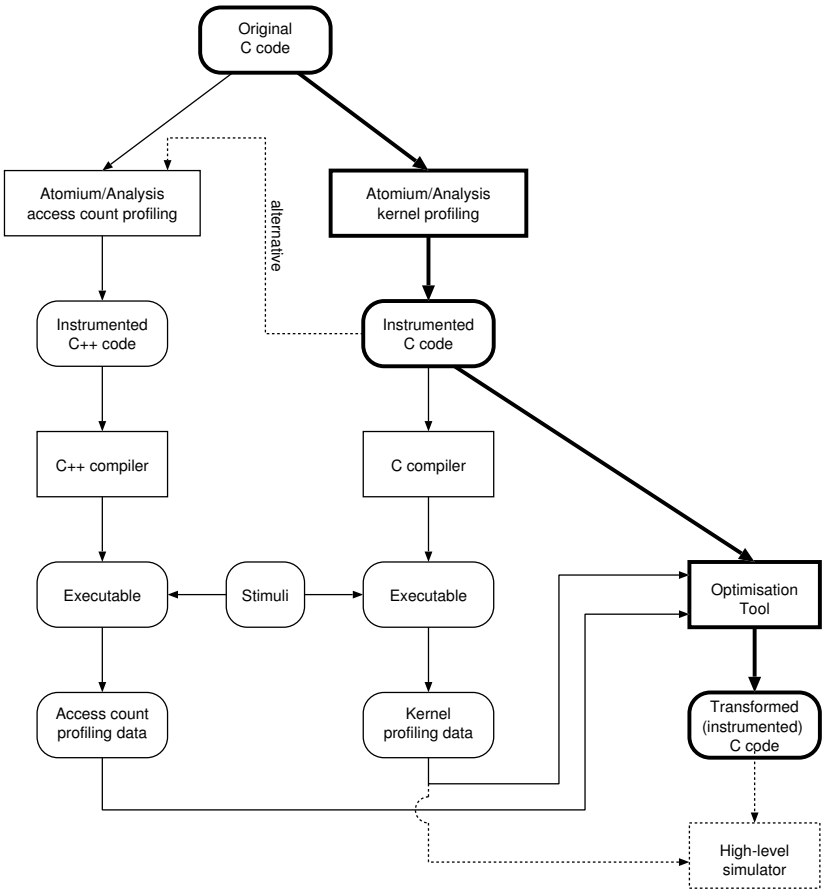


Figure 4: The different ATOMIUM profiling flows and their interaction.

The instrumentation for access count profiling and kernel profiling follow different paths: from the access count profiling path, only the profiling information is used eventually. In contrast, for the kernel profiling flow, the instrumented code is also used during the rest of the flow (indicated in bold).

The kernel profiling data may be used by optimisation tools, but also by an optional high-level simulator following the optimisations (indicated in dotted lines at the bottom), thanks to the fact that the instrumentation remains present in the code.

Note that it is not strictly necessary to start the access counting profiling from the original C code. Alternatively, access counting profiling can be applied to the code that was instrumented for kernel profiling, as this code is still C code and does not affect the access counts. This is

indidated by the alternative dotted arrow at the top of the figure.

4.3 Implementation

Since it should be possible to record each individual sample and to play the recorded data back, some additional context information is needed whenever a kernel or a profiled section is being executed. Part of this information consists of the call path leading up to the kernel at the moment that it is executed.

For instance, consider the following code:

```
void foo(int start, int stop)
{
  int x, y;
  for (y = start; y < stop; ++y)
  {
arp_kernel:
    for (x = 0; x < width; ++x)
    {
      ...
    }
  }
}

int main()
{
  int t;

  for (t = 0; t < N; ++t)
  {
    foo(0, 100);
    foo(100, 200);
  }

  ...

  if (<some condition>)
  {
    foo(0, 30);
  }
}
```

In this code, we have a kernel (marked by an `ato_kernel` label) in function `foo()`. This function is called from different places, though, with varying argument values.

If we would only instrument the kernel loop, we would not be able to make a distinction between the different paths leading to the execution of the kernel, i.e., we would not know which cycle measurements for the kernel loop would belong to which calls in the `t` loop or that in the condition. In order to be able to do this, we must add extra instrumentation to keep track of the execution path. This is done the code below:

```
void foo(int start, int stop)
{
```

```
  int x, y;
  arp_enter(1, 2, 0);
  for (y = start; y < stop; ++y)
  {
arp_kernel:
    arp_enter_kernel(1, 1, 0);
    for (x = 0; x < width; ++x)
    {
      ...
    }
    arp_leave_kernel(1, 1);
  }
  arp_leave(1, 2, 0);
}

int main()
{
  int t;

  for (t = 0; t < N; ++t)
  {
    arp_enter(1, 5, 0);

    arp_enter(1, 3, 0);
    foo(0, 100);
    arp_leave(1, 3);

    arp_enter(1, 4, 0);
    foo(100, 200);
    arp_leave(1, 4);

    arp_leave(1, 5);
  }

  ...

  if (<some condition>)
  {
    arp_enter(1, 6, 0);
    foo(0, 30);
    arp_leave(1, 6);
  }
}
```

As can be seen, not only an `ato_enter_kernel()/ato_leave_kernel()` call pair was added around the kernel loop, but also `ato_enter()/ato_leave()` pairs have been added at strategic places in the code.

Most importantly, every call to `foo()` has been surrounded by an enter/leave pair with a unique section id (the second argument in the calls). When the profiling framework keeps track of these additional sections (and their ids), it can make a distinction between the different call paths that lead to the execution of the kernel.

Secondly, in every loop body surrounding (at run-time) an execution to the kernel loop, such as loops `t` and `y` in this case, an additional section bounded by an enter/leave pair was created. This allows the `arp` library to keep track of each execution of each loop (surrounding

the kernel) separately, which is required for accurate playback in case individual samples are recorded.

Internally, the `arp` library uses a counter for every section. This counter is normally incremented each time that the section is entered, except when the context surrounding the section has changed. In that case, the counter is reset. A change of context can be caused by two reasons: either the section is entered via another call path, or the counter of the surrounding section was reset (because the context of the surrounding section has changed).

In this way, the `arp` library can identify a unique counter tuple with every execution of a kernel or profiled section. The following example illustrates this:

```
arp_enter(1, 3, 0);
for (...)
{
  arp_enter(1, 2, 0);
  for (...)
  {
    arp_enter_kernel(1, 1, 0);
    <kernel code>
    arp_leave_kernel(1, 1);
  }
  arp_leave(1, 2);
}
arp_leave(1, 3);
```

Fig. 5 represents the execution of this piece of pseudo-code graphically.

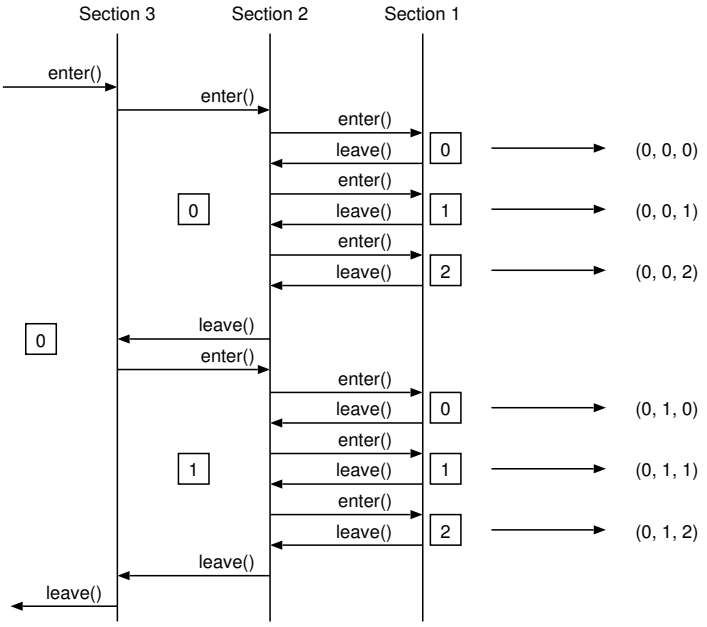


Figure 5: Internal section counters maintained by the `arp` library

With every section, a counter is associated (the values of which are shown in boxes). Each time that a section is entered, its counter is incremented, unless its context has changed. As

a result, a unique counter tuple (shown a the right) can be associated with every execution of the kernel section. When these counter tuples (representing a combined call and iteration trace) are saved together with the instrumentation data, it is possible to play the results back and retrieve the proper data for every execution of the kernel.

For the latter to work, it is vital that the counter tuples can be generated in exactly the same way as when the profiling data were collected. This is one of the reasons why the profiling calls must remain in the code during transformations by high-level optimisation tools. In that way, the section ids can survive transformations.

4.4 Parallisation issues

Now that some details on the internal operation of the `arp` library have been explained, it is time to explain the 'mysterious' third parameter that is used in the `arp_enter*()` calls: this number is actually the start value to be used for the section's internal counter whenever the counter is reset. Under normal circumstances, one can leave this value at zero, as is done in previous examples in this text.

However, when an application is parallelised, it is possible that certain loops are split over different threads, each of which take care of executing a number of iterations. For instance, the following code:

```
int main()
{
  for (...)
  {
    arp_enter(1, 2, 0);
    for (x = 0; x < N; ++x)
    {
      arp_enter_kernel(1, 1, 0);
      <kernel code>
      arp_leave_kernel(1, 1);
    }
    arp_leave(1, 2);
  }
}
```

may be parallellised, as part of a high-level optimisation, as follows:

```
void thread1()
{
  arp_spawned(1, 2);
  for (x = 0; x < N/2; ++x)
  {
    arp_enter_kernel(2, 1, 0);
    <kernel code>
    arp_leave_kernel(2, 1);
  }
  arp_join(2);
}

void thread2()
```

```
{
  arp_spawned(1, 3);
  for (x = N/2; x < N; ++x)
  {
    arp_enter_kernel(3, 1, 0);
    <kernel code>
    arp_leave_kernel(3, 1);
  }
  arp_join(3);
}

int main()
{
  for (...)
  {
    arp_enter(1, 2, 0);
    spawn(thread1);
    spawn(thread2);

    join(thread1);
    join(thread2);
    arp_leave(1, 2);
  }
}
```

Note that `arp_spawned()` and `arp_join()` calls have been inserted to notify the `arp` library about the thread creation and termination, and that each thread gets its unique thread id (the first argument in the enter and leave calls).

When this code is used to playback recorded data from the original sequential version, there is a problem, though: the two threads generate the same counter tuples for the kernel section. The reason is that both threads reset their counter for the kernel section to zero each time when the `x` loop is entered. Hence, while for the original loop the counter ran from 0 to N, the counter only runs from 0 to N/2 for both threads. As a consequence, both threads will play back only the first half of the recorded samples.

Actually, what is needed here is that the second thread plays back the second half of the samples. This can be achieved by changing the start value for the section iteration in the second thread. In the example, the line `arp_enter_kernel(3, 1, 0);` has to be replaced by `arp_enter_kernel(3, 1, N/2);`. In this way, the section counter tuples for both threads will be unique and cover all possible iterations.

Note that this updating of the start value for a section iterator is not specific to kernel sections: it needs to be done for any kind of section for which te iterations are divided over several threads.

Under normal circumstances, the user does not have to worry about these parallellisation issues, as the (future) optimisation tools that introduce these transformations will automatically take care of updating the start indices (and thread ids).

5 Known problems

Like all software, ATOMIUM RECORD & PLAYBACK library has some known problems.[9] These are listed in this section, together with some suggested work arounds.

5.1 `setjmp()` and `longjmp()`

Problem: While ATOMIUM/ANALYSIS has no problem processing code that contains calls to `setjmp()` and `longjmp()`, the instrumented application is likely to crash if `longjmp()` actually is activated. Even if it does not crash, the resulting kernel profiling report is very likely to be incorrect. This problem is caused by the fact that `longjmp()` is sort of incompatible with the use of C++ in general and in the ATOMIUM run time library in particular.

Symptom: See above.

Fix: In general, there is no way to work around this kind of problem, other than avoiding the use of `setjmp()` and `longjmp()`.

5.2 Applying `pthread_cancel()` to the main thread

Problem: ATOMIUM RECORD & PLAYBACK library does not support applying `pthread_cancel()` to an application's main thread. This is to a large extent due to the fact that this operation, although expressly permitted by the POSIX threads specification, is at present still non-portable in real life and in addition appears to be unsafe on many operating systems.

Symptom: When an instrumented application tries to cancel its main thread, it will first issue a message on the `stderr` stream indicating that doing so is not supported. After this, the thread is cancelled as requested, but there is no guarantee whatsoever that the remaining threads will continue to operate normally, nor that the access count report will be both generated and correct.

Note that, in contrast to the above, calling `pthread_exit()` from within the main thread *is* supported by ATOMIUM RECORD & PLAYBACK library .

Fix: There is no fix.

[9] And certainly some as yet unknown ones... 😉

6 The Atomium Record & Playback library

The Atomium Record & Playback library offers a flexible instrumentation framework that can be used to obtain accurate execution (cycle) times or critical parts of an application through simulation. The library does not perform any profiling itself. This is left to the user, who has to hook up the necessary profiling callbacks.

Atomium/Analysis knows how to instrument an application for kernel profiling with calls to the Atomium Record & Playback library. After instrumentation, adding the profiling callbacks, compiling and linking with the library, the resulting executable exhibits the same behaviour as a compiled version of the original application, with the exception that it also produces profiling information (see Section 2). This section describes the Atomium Record & Playback library application programming interface (API) and the various modes of operation that are available.

6.1 The Atomium Record & Playback library API

The Atomium Record & Playback library API is defined in the include file `atomium_rp.h`, which can be found in the standard Atomium include directory. It contains several sections.

6.1.1 Type definitions

A first set of simple type definitions of the Atomium Record & Playback library is shown below:

```
typedef unsigned short arp_tid;  /* Thread ids */
typedef unsigned short arp_sid;  /* Section ids */
typedef int            arp_lidx; /* Loop index */

typedef enum _arp_data_type
             { arp_char,  arp_uchar,
               arp_short, arp_ushort,
               arp_int,   arp_uint,
               arp_long,  arp_ulong,
               arp_llong, arp_ullong,
               arp_float, arp_double,
               arp_ldouble } arp_data_type;

typedef enum _arp_mode
             { arp_record_average,
               arp_record_all,
               arp_record_histogram,
               arp_playback } arp_mode;

typedef enum _arp_file_format
             { arp_ascii,
               arp_binary } arp_file_format;

typedef enum _arp_error_t
             { arp_warning, arp_error_fatal, arp_error_non_fatal,
               arp_error_syntax, arp_error_io } arp_error_t;
```

These are explained in more detail below, when they are used in other typedefs and function prototypes.

A second set of typedefs defines the signature of the callbacks that the user can/must provide to the Atomium Record & Playback library:

```
typedef void (*arp_enter_callback) (const arp_sid, void* data, void* context);
typedef void (*arp_leave_callback) (const arp_sid, void* data, void* context);

typedef void (*arp_lock_callback)  (const void*, void*);
typedef void (*arp_unlock_callback)(const void*, void*);

typedef void (*arp_spawn_callback) (const arp_tid, void** contextp);
typedef void (*arp_join_callback)  (const arp_tid, void** contextp);

typedef void (*arp_error_callback) (const arp_error_t, const char* msg, void* context);
```

The first two types define the type of the callback functions that are called whenever a profiled or kernel section is entered or left, respectively (see Section 2). The first argument is the unique identification number of the section (type `arp_sid`, defined above), the second one is a pointer to the data area in which the profiling data are stored, and the third pointer points at some context information provided at initialisation time (see below).

The second pair of callback types defines the signatures of a locking and unlocking functions, which have to be provided when a multithreaded application has to be instrumented (see Section 3). They receive a pointer to a data area in memory that has to be protected from simultaneous access by multiple thread, and also pointer to an optional locking context that can be provided at initialisation time (like a mutex in the example in Section 3).

The third pair of callback types defines the signatures of a thread spawn and join callback. If provided, the spawn callback is called whenever a new thread is created, and the join callback is called when a thread is about to finish. Both of them receive as argument the (unique) id number of the (child) thread that was spawned or that is about to be joined, and a *pointer to the data context pointer* provided at initialisation time (the pointer that is also passed to the enter and leave callbacks). These callbacks can update the data context pointer, if necessary (see Section 3 for an example). Note that a spawn an join callback are optional, even for multithreaded applications.

Finally, there is the signature definition for an optional error handler. Whenever the Atomium Record & Playback library library detects an error or a potentially erroneous situation, it will issue an error message and exit or issue a warning message, respectively, by default. This may not be the desired behaviour when the application is run in an instruction set simulator, for instance. Therefore, it is possible to install an error handler to intercept these warnings and errors. When installed, and an error occurs, this handler is called with 3 arguments: the type of the error (on of the `arp_error_t` types defined above), a pointer to a message describing the warning or error, and a context pointer that can be provided at initialisation time (which may be different from the data context pointer used in the enter, leave, spawn and join callback).

6.1.2 The record and playback API

The most important functions in the Atomium Record & Playback library are the functions that have to be called during the instrumentation:

```
extern void     arp_enter         (const arp_tid, const arp_sid, const arp_lidx);
extern void     arp_enter_kernel  (const arp_tid, const arp_sid, const arp_lidx);
extern void     arp_enter_profiled(const arp_tid, const arp_sid, const arp_lidx);
extern void     arp_leave         (const arp_tid, const arp_sid);
extern void     arp_leave_kernel  (const arp_tid, const arp_sid);
extern void     arp_leave_profiled(const arp_tid, const arp_sid);

extern void     arp_spawned       (const arp_tid, const arp_tid);
extern void     arp_join          (const arp_tid);
```

The first 6 functions, when called, signal the Atomium Record & Playback library that a non-profiled section, a (profiled) kernel section, or a general profiled section is about to be entered or left, respectively (see Section 2. Every enter call has to be matched by a corresponding leave call, in the proper order: when sections are nested, the leave call for an inner section must be executed before the leave call of the surrounding section, *under all possible conditions*. The latter means that for every enter call, there may be several corresponding leave call sites, if the section can be left via different paths. For instance, if a section corresponds to a function body, an enter call is typically inserted at the entrance of the function, but if the function contains multiple *return* statements, each of these must be preceeded by a matching leave call.

The enter functions have to be called with three parameters: a unique id number identifying the thread in which the section is entered, a unique id number identifying the section, and a start index. The latter is only relevant for for multithreaded applications using data-parallelism. More information on how this parameter is used in provided in Subsection 4.4.

The leave functions have to be called with only the thread id number and the section id number.

The next set of functions have to be called to initialise, start, and finalise the recording (or playback):

```
extern void arp_add_callbacks(const arp_data_type,
                              arp_enter_callback,
                              arp_leave_callback,
                              arp_spawn_callback,
                              arp_join_callback,
                              void* context);
extern void  arp_start       (const arp_tid,       /* Maximum thread id */
                              const arp_sid,       /* Maximum section id */
                              const arp_mode,
                              arp_error_callback,
                              void* error_context,
                              arp_lock_callback,
                              arp_unlock_callback,
                              void* lock_context);
extern void     arp_finish   ();
```

The function to add callbacks has to be called first: it installs the enter and leave callbacks (which are called whenever a profiled (kernel) section is entered or left), and the optional spawn and leave callbacks. If no spawn and leave callbacks are needed, NULL-pointers can be passed instead. The last argument is a pointer to context information that the enter, leave, spawn, and join callbacks may need. If no context information is needed, a NULL-pointer can be passed here too.

Note that it is possible to add several (independent) sets of callbacks: when multiple callbacks set are added, they are called in the order that they have been registered. Each set of callbacks can have its own data type and does not share its data storage areas with other callback sets.

The start function, when called, starts the actual recording (or playback). The arguments to be provided are: the highest possible thread id number used in the application; the highest possible section id number; the operation mode (which selects one of the various recording modes or playback mode; see below), an optional error handler and context; and a locking and unlocking callback and context. The latter callbacks and context are only required for multithreaded applications (see Section 3).

The finish function stops the recording (or playback) and saves the recorded data (when in recording mode).

The following functions can be used to control the name and format[10] of the file to which the profiling data are saved. If no name is set, the default name '`data.arp`'. If a filename and/or a file format is provided, this must be done before the start function is called.

```
extern void arp_set_file_name         (const char*);
extern void arp_set_output_file_format(const arp_file_format);
```

6.2 Operation modes

The Atomium Record & Playback library library's primary objective is to allow the recording of accurate profiling data which can be used by optimisation tools such as Atomium/MH to steer their optimizations. Therefore, the library currently provides two recording modes: `record_all` mode, which keeps track of all recorded values for all section traces, and `record_average` mode, which only keeps track of the average recorded value for every section trace. In the future, additional recording modes may be provided (such as a histogramming mode).

The library can also be used to play recorded data back (`playback` mode). Playing data back is performed in a similar way as recording. The only difference is the fact that when an enter callback is called, the data area that is passed to it contains the previously recorded data and any value written to this area is passed to the matching leave callback but futher ignored.

The playback mode is useful for verifying the certain optimisations such as parallellisations. It will be supported by future Atomium tools.

6.3 Compilation modes

At compilation time, the `arp` API can be tuned towards the way in which the profiling information is recorded.

[10] Currently only one file format is implemented, but this may be extended in the future.

By default, all functions in the API behave as described above. This mode is intended for the situation where the resulting executable is run natively on a desktop PC, for instance, where the Atomium Record & Playback library library can be accessed without limitations.

When the compiled executable is to be run in an instruction set simulator, it may not be technically possible to install callbacks in the normal way from within the application code. Typically, the simulator will install those callbacks itself, e.g., to avoid that the callbacks would have to be simulated too (which could hence influence the measurements). To solve this problem, the API can be partly remapped by means of a C preprocessor definition: when the application is compiled with the `-DARP_SIM` option, the `arp_add_callbacks()` and `arp_start()` functions are mapped onto the following to functions, respectively:

```
extern void arpsim_add_callbacks();
extern void arpsim_start(const arp_tid, const arp_sid, const arp_mode);
```

All callback and context arguments are dropped in these calls. However, the Atomium Record & Playback library does not provide an implementation of these functions! It is up to the simulator to provide an implementation that is accessible by the application. This implementation will then typically call the original `arp` function, *with* the appropriate callbacks and contexts.

Finally, it is also possible to compile instrumented code as if the code was not instrumented at all: by adding the `-DARP_NONE` flag to the compiler command line, all `arp` function calls are effectively removed from the code (through preprocessor magic).

6.4 The Query API

The query API of the `arp` library allows an external application to load and query saved profiling results. As explained in Section 4, recording and playback is centered around section call and iteration traces: all recorded data are associated with a certain section call trace (the list of ids of active sections when the kernel or profiled section is entered), optionally extended with section counter values (in case each individual sample is recorded).

A first set of functions in the query API can be use to load profiling data from a file, to retrieve some (meta-) information about the recorded data, to optionally switch to another mode, and to free the loaded data:

```
extern void                arp_load_data              (const char *file_name,
                                                       arp_error_callback cb,
                                                       void *context);
extern const arp_mode      arp_get_mode               ();
extern void                arp_switch_mode            (const arp_mode mode);
extern void                arp_free_data              ();

extern const unsigned      arp_get_data_mult          ();
extern const arp_data_type arp_get_data_type          (const unsigned dataIndex);
```

Note that an error handler can be provided when data are loaded, similar to when data are recorded or played back.

Switching between recording modes obviously has some limitations: for instance, if the data were recorded in average mode (`record_average`), it is not possible to restore the individual samples by switching to `record_all` mode. The other direction is always possible though: when the data file contains all individual samples, switching to `record_average` mode converts the samples to averages. This may be useful for tools that can only work with average information, for instance.

The data multiplicity of a trace set is the number of callback sets that were loaded at the time of recording. Usually, this will be 1. Since each callback set can have its own data type, it is possible to retrieve the data type for each set, given the index of the set.[11]

Once data have been loaded, the `arp` query API provides the necessary functionality to inspect the recorded traces. A trace is defined as an opaque data type `arp_trace`, which can be passed to certain functions to retrieve information about the trace. The trace related functions are the following:

```
typedef void *arp_trace;

extern const unsigned      arp_get_nTraces            ();

extern const arp_trace     arp_get_trace              (const unsigned traceNo);
extern void                arp_free_trace             (const arp_trace trace);

extern const unsigned      arp_trace_get_count        (const arp_trace trace);
extern const unsigned      arp_trace_get_stack_length (const arp_trace trace);
extern const arp_sid       arp_trace_get_stack_pos    (const arp_trace trace,
                                                       const unsigned depth);
extern const arp_lidx      arp_trace_get_stack_index  (const arp_trace trace,
                                                       const unsigned depth);
extern const void*         arp_trace_get_value        (const arp_trace trace,
                                                       const unsigned dataIdx);
```

The first one returns the number of traces that are available in the data set. Note that this number may change when the mode of the set is altered: an average mode will in general carry fewer traces than a sample mode.

Next, the traces can be retrieved one by one by calling `arp_get_trace()` with the appropriate index. Note that every retrieved trace must eventually be freed again using `arp_free_trace()` to avoid memory leaks.

`arp_trace_get_count()` returns the number of samples that the trace represents. In sample mode (`record_all`), every trace corresponds to a single sample and hence the count will always equal 1.

`arp_trace_get_stack_length()` returns the number of sections that are present in the trace, i.e., the number of nested sections that were active when the corresponding kernel or profiled section was entered. `arp_trace_get_stack_pos()` can be used to retrieve the ids of each of these sections.

In case the traces are collected in `record_all` mode, the traces also contain information about the section counters at the time of the measurement. These counters values can be retrieved by means of `arp_trace_get_stack_index()`.

[11] All indexes in the query interface are zero-based.

Finally, the measured sample data can be retrieved by means of `arp_trace_get_value()`. In case there were multiple callback sets, the index for each set can used to retrieve the corresponding values. If there is only one set, which is usually the case, the index is always zero.

6.5 Data format of ARP database

Output file generated by executing an ARP-instrumented application can grow very big in size depending on number of kernel iterations e.g. for MPEG4 encoder application running 300 frames, the database can grow to the order of 1GB. This size can also cause execution delays (during database access) while running the applicationin *Playback* mode. Hence to create a database with an optimal size, ARP follows an (internally) predefined data format. Further, publicly available compression tools such as *gzip* are used to reduce disk space needed to store the output file.

Following example shows *data.arp* output file generated by running Atomium Record & Playback library in *Record* mode for recording all kernel executions in all iterations (*arp_record_all* mode) . The example is followed by a brief explanation of the data format used by Atomium Record & Playback library.

```
# Atomium record and playback data - format version 1.1.
0 1 1 5
1033259
8 0 148@0 147@0 146@0 145@0 144@0 143@0 142@0 141@0: 20239254;
16 8 140@0 139@0 138@0 130@0 129@0 30@0 29@0 28@0: 148;
18 16 19@0 18@0: 105;
15 13 40@0 39@0: 182;
16 15 31@0: 162;
```

- First line prints Atomium Record & Playback library version information.
- Second line (0 1 1 5) prints
  - File format: 0 = ascii, 1 = binary
  - ARP mode: 0 = average[12], 1 = all, 2 = histogram[13], 3 = playback
  - Number of callbacks
  - Type of data for each of the callbacks : char/short/int/... 5 = *unsigned int*
- Third line (1033259) prints number of ARP records in this file
- Forth line onwards are the actucal ARP records dumped. Each record identifies uniquely an instance of a kernel execution during the application run. First column in each record tells about depth of the kernel in terms of ARP section-stack built e.g. "8 0 148@0 147@0..." says that this kernel is 8-stack deep and it is the kernel inside section

[12] For *average* mode, only average execution time per kernel is recorded. This means that the data format is little different in that case, mainly that each iteration count is not present.

[13] *histogram* mode is not yet implemented.

141 which is inside section 142 and so on till section 148. *@0* means the number of iteration. Last figure (*20239254* in this case) gives the cycle count for this kernel iteration. *0* followed by *8* in the second column indicates how many columns from the previous record can be reused to access this record. e.g. Fifth line *"16 8 ..."* means that out of 16 ARP sections, first 8 are the same as the previous one. Note that this stack reuse counts in the traces denotes number of reused stack entries with *unchanged iteration counts* of the previous record. This reuse of section numbers in the data format is devised in order to reduce the disk space needed to store the database.