ATLaS: Applied Thread-Level Speculation

Version 1.1

Sergio Aldea, Alvaro Estebanez, Diego R. Llanos and Arturo González-Escribano

sergio@infor.uva.es, alvaro@infor.uva.es, diego@infor.uva.es, arturo@infor.uva.es

COMPUTER SCIENCE DEPARTMENT
UNIVERSIDAD DE VALLADOLID, SPAIN

Grupo Trasgo

Universidad de Valladolid

Universidad de Valladolid

Contents

1 Introduction

ATLaS (Applied Thread-Level Speculation) is a compiler and runtime system that supports the speculative execution of C source codes.

ATLaS allows to easily parallelize *for* loops that can not be analyzed at compile time and/or present dependence violations when executed in parallel. The use of this solution do not require any additional training with respect to the one needed to use OpenMP directives. Moreover, this simplifies the task of classifying variables according to its usage. If unsure, the programmer may label a certain variable to be speculative, a decision that guarantees the correct parallel execution of the loop, possibly at the cost of a lower performance. Current state of the implementation correctly execute in parallel loops with runtime dependence violations.

Figure 1 shows an example of (a) a sequential C loop, and (b) its parallelization with OpenMP directives. As can be seen, all variables inside the loop body should be classified as `private` or `shared`. Informally speaking, `private` variables are always defined in a given iteration before their use, while `shared` variables have values that are visible by all threads executing the loop in parallel. In our example, `a[]` is a read-only shared vector, while `v[]` is a shared vector modified at each iteration. See [1, 2, 3, 4] for further details.

Being OpenMP a simple and powerful mechanism for code parallelization, its use has several limitations. First, the classification of all variables inside of the critical region according to their use is a time-consuming, error-prone task. Second, OpenMP does not ensure the execution of the code according to sequential semantics, being the programmer responsible for such task. In our example, in Fig. 1, the programmer is responsible to ensure that each thread modifies a different element of `v[]`. Third, in many cases potentially-parallel regions can not be safely parallelized because their control flow depend on runtime data. Consider the code depicted in Fig.2. Suppose that the value of `k` is not known at compile time. Assuming `b>0`, if the parallel execution of the

```
for (i=0; i<MAX; i++) {
   b = func(i);
   v[i] = b * a[i];
}
```

(a)

```
#pragma openmp parallel for \
 private (i,b) shared (a,v)
for (i=0; i<MAX; i++) {
   b = func(i);
   v[i] = b * a[i];
}
```

(b)

Figure 1: Example of loop parallelization with OpenMP.

```
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```

(a)

```
#pragma openmp parallel for \
 private (i,b) shared (a,k) \
 speculative(v)
for (i=0; i<MAX; i++) {
   b = func(i);
   if (b==k)
      v[i] = v[i-b];
   else
      v[i] = b * a[i];
}
```

(b)

Figure 2: A loop that can not be safely parallelized with current OpenMP clauses (a), and its parallelization with our new `speculative` clause (b).

loop calculates iteration `i` *before* iteration `i-b`, access to `v[i-b]` may return an outdated value, breaking sequential semantics. The only way to guarantee a correct behavior would be to serialize the execution of iterations $i - b$ and i, a difficult task in the general case.

Our proposal consists in augmenting OpenMP with software-based, Thread-Level Speculation (TLS) techniques, to ensure that definitions and uses of `shared` variables are carried out according to sequential semantics. To do so, we define a new `speculative` clause. Variables labeled as `speculative` will be accessed following two simple rules:

- All reads of a `speculative` variable will return the most up-to-date value for this variable. This value can be either generated previously by this thread or by any of its *predecessors* (threads that execute earlier iterations according to sequential semantics). This is called a *forwarding* operation.
- All writes to a `speculative` variable will store the value in a local copy, and will check that no *successor* threads (threads that are executing "future" iterations) have consumed an outdated value of this variable. In this case, the offending thread (and possibly some of its successors) will be stopped and re-started, in order to force them to consume the updated value of the variable. This is called a *squash* operation.

As long as the values of `speculative` variables can be discarded due to a dependence violation, all threads maintain version copies of the `speculative` variables being accessed. When a non-speculative thread successfully finishes the execution of its block of iterations, all changes are committed to the main copy of all `speculative` variables.

We have modified GCC to support the new OpenMP clause `speculative`, and ATLaS is responsible for managing the variable classification done with this and others clauses to compile and run the source code according to the speculative scheme described.

ATLaS is distributed under the GNU General Public License (GNU GPL).

2 Content of the software downloaded

The package contains the following directories:

- **`atlas`**: This is the script that allows us to run ATLaS. The different options to execute this script are described in Sect. 5.1.
- **`doc/`**: This directory contains the `PDF` version of this document.
- **`gcc_updates/`**: This directory contains the source files that modify the compiler GCC to support the new OpenMP `speculative` clause.
- **`specprag/`**: This directory contains the core of ATLaS, all the source code that implements the compiler and runtime modules of ATLaS.

3 System Requirements

ATLaS only requires the GCC compiler, preferably version 4.6.2, to be executed. The rest of packages required are involved with the compilation process of GCC and they will be described in the next section. Therefore, it is necessary to install GCC (the version 4.6.2 is recommended) with the plugin support enabled. To check this, it is needed to execute the following command and see the corresponding output.

```
$ <gcc_install_dir/>gcc -print-file-name=plugin
<gcc_install_dir/>/lib/gcc/<architecture_and_so>/4.6.2/plugin
```

This path indicates the directory where the needed header files to execute plugins are located. If the plugin support is not enabled, you only will receive the word `plugin` in the output.

The compiler module of our system was developed using version 4.6.2 of GCC. Any version superior to 4.5, in which plugins were introduced, should work. However, we only ensure that the compiler module work with version 4.6.2.

4 Installation Guide

The installation process should be done in two steps. First, you have to download, compile and install GCC. Second, you need to replace original GCC files with source files provided in the ATLaS package. We are going to explain how to install in the Ubuntu operating system the version of GCC used in the development of this plugin, specifically, version 4.6.2, other versions and/or operating systems should also work, but are not tested yet.

4.1 Install and compile GCC

First, you have to download a version of GCC from one of the official mirrors. For example, we downloaded GCC 4.6.2 from the next URL:

`ftp://www.mirrorservice.org/sites/ftp.gnu.org/gnu/gcc/gcc-4.6.2/gcc-4.6.2.tar.bz2`

GCC requires that various tools and packages are available for use in the build procedure. The following packages are required:

- Another GCC compiler. This is necessary to compile the required version of GCC.
- GNAT
- GMP (libgmp3-dev in Ubuntu)
- MPFR (libmpfr-dev in Ubuntu)
- MPC (libmpc-dev in Ubuntu)
- GNU binutils
- libc6-dev[1]
- libtool
- GAWK

A more detailed list of the prerequisites can be found in:

`http://gcc.gnu.org/install/prerequisites.html`

[1]GCC tries to include as much compatibility as possible, so it requires by default the installation of the 32-bit versions of *libc*. Thus, if you have a 64-bit system, install the 32-bit version of the library: libc6-dev-i386. Otherwise, if you do not want to preserve any compatibility, you can add the flag *--disable-multilib* in the configure command).

Once you have installed all the packages listed above, you need to prepare the building directory.

1. Uncompress the file downloaded into a directory called `srcdir`.

```
$ mkdir <gcc_build_dir>
$ mkdir <gcc_build_dir>/srcdir
$ mv gcc-4.6.2.tar.bz2 <gcc_build_dir>/srcdir
$ cd <gcc_build_dir>/srcdir
$ tar -xvjz gcc-4.6.2.tar.bz2
```

2. At the same level than `srcdir`, create a new directory called `objdir`.

```
$ mkdir <gcc_build_dir>/objdir
```

3. Configure the compilation process of GCC with the following command. The `--prefix` option indicates where this version will be installed.

```
$ cd objdir
$ ../srcdir/configure --enable-shared --enable-threads=posix
  --enable-__cxa_atexit --enable-clocale=gnu
  --enable-languages=c --prefix=/opt/gcc-4.6.2
```

4. Starts the compilation. This process could take several hours.

```
$ make bootstrap
```

5. Install the compiler in the directory indicated in the configuration command.

```
$ make install
```

6. Once this process has finished, you have GCC version 4.6.2 installed in your computer. The next step is add to this version the support for the OpenMP `speculative` clause.

More compilation options and a detailed documentation of this process can be found in:

`http://gcc.gnu.org/install/configure.html`

4.1.1 Known issues

1. If *.info* files cause problems, do not built them. They are not needed here and are broken with the newest versions of *makeinfo*.

```
$ cd srcdir
$ sed -i 's/BUILD_INFO=info/BUILD_INFO=/' gcc/configure
```

2. In addition of installing *32 bit libc dev* package, the installation of *gcc-4.6.2* in Ubuntu 12.04 produces sometimes a known problem that puts the files in a non standard location. This will be solved adding to your *.bashrc*:

```
export LIBRARY_PATH=/usr/lib/$(gcc -print-multiarch)
export C_INCLUDE_PATH=/usr/include/$(gcc -print-multiarch)
export CPLUS_INCLUDE_PATH=/usr/include/$(gcc -print-multiarch)
```

4.2 Add support for the OpenMP `speculative` clause into GCC

Current implementations of OpenMP does not support Thread-Level Speculation (TLS). We have designed a new OpenMP clause, called `speculative`, to support TLS into GCC. For this purpose, it is needed to modify the following files of GCC:

```
$ cd gcc_updates
$ ls *
c-parser.c gimplify.c tree.c tree-nested.c
c-typeck.c omp-low.c tree.h tree-pretty-print.c

c-family:
c-omp.c c-pragma.h
```

Modifying GCC to add this new clause can be done in two simple steps:

1. Copy the files in directory `gcc_updates` into the directory that contains the original source files of GCC.

   ```
   $ cp -a gcc_updates/* <gcc_build_dir>/srcdir/gcc
   ```

2. Recompile GCC, executing the following command in the directory `objdir`.

   ```
   $ cd <gcc_build_dir>/objdir
   $ make
   ```

At this point, a modified version of GCC that supports the new OpenMP clasue is installed in your computer.

5 Using ATLaS with your software

Once you have installed our modified version of GCC, you need to accomplish a few more steps to use ATLaS with your application. You have to follow the next steps to install ATLaS (compiler and runtime modules):

1. Copy (or link) `atlas` script into the directory that contains your source code. This script drivers the compilation of your program with the TLS library.

   ```
   $ cp atlas <your_application_dir>
   ```

2. Create a symbolic link to the directory `specprag` of the installation directory, using the same name than the original. This directory contains the compile and runtime modules of ATLaS.

   ```
   $ ln -s <path_to_specprag> specprag
   ```

3. Update the location of the compiler in *atlas* and *specprag/Makefile* files.

```
$ In atlas: COMPILERDIR=<path to the directory of the gcc-4.6.2 binary>
$ In specprag/Makefile: CC=<{path to the gcc-4.6.2 binary>
```

5.1 Command Options

ATLaS has several options. Mandatory arguments for long options are also required for short options.

`-t, --threads INTEGER` : specifies the number of threads to execute the resulting binary file.

`-b, --block INTEGER` : specifies the size of each block of iterations. Either dynamic or static approaches can be used, see appendix A for more information.

`-p, --maxpointer INTEGER` : specifies the maximum number of elements which are speculative.

`-i, --maxiter INTEGER` : specifies the maximum number of iterations that a speculative loop can execute.

`-m --mask INTEGER` : specifies the size of the mask used.

`-c, --compile "FILE1.C FILE2.C ..."` : specifies the list of source files to be analyzed.

`-f, --flag "FLAG1 FLAG2 ..."` : specifies the flags to compile the code.

`-e, --exec RUNFILE` : specifies the name of the executable resulting of the compilation of the user's program.

`-d, --dump` : enables the dumping at various stages of processing the intermediate language tree to a file.

`-h, --help` : display this help and exit.

`-v, --version` : output version information and exit.

Six of this options are mandatory: `--threads`, `--block`, `--maxpointer`, `--maxiter`, `--mask`, and `--compile`. If you do not specify any of these options, ATLaS generates the following output:

```
1  You must specify the number of threads with '-t' or '--threads'
2  You must specify the size of the block of iterations with '-b' or '--block'
3  You must specify the maximum number of speculative elements with '-p' or
4                                                                  '--maxpointer'
5  You must specify the maximum number of iterations with '-i' or '--maxiter'
6  You must specify the size of the mask with '-m' or '--mask'
7  You must specify a C file with '-c' or '--compile'
8
9  Usage: atlas -t "threads" -b "block" -p "maxpointer" -i "maxiter" -m "mask"
10              -c "file1.c"
11 Example: atlas -t 4 -b 50 -p 10 -i 10000 -m 127 -c "example.c"
```

A template for a correct execution of ATLaS could be the following:

```
$ atlas --threads T --block B --maxpointer P --maxiter I --mask M -c example.c
```

where I is the maximum number of iterations that a speculative loop can execute in the program, T is the number of threads we want to run the program with, B is the size of the block of iterations, P is the maximum number of elements which are speculative, and M is the size of the mask used. These parameters are set by the programmer and they are not very tricky to set up, because they only need to know some easy features of the target loop to set *maxiter* and *maxpointer*. For example, a loop that speculatively reads from, and writes into an array of 1000 elements, with 200 iterations, sets the value of P to 1000, and I to 200. The other three parameters, the number of threads, the block size, and the mask size are variable and programmers can experiment with different values until obtaining the best performance to their programs. The different strategies implemented to schedule a loop are detailed in the appendix A.

5.2 Running Example

The following steps resume the process of parallelizing an application with ATLaS.

1. Analyze the loop to be parallelized, classifying their variables into private, shared or speculative. Any variable that could lead to a dependency violation should be classified as speculative.

2. Add the OpenMP directive `omp parallel for`, with the corresponding clauses `private`, `shared`, and the new `speculative` clause.

3. Add the function `specbegin(N)` before the parallel loop to specify the number of iteration of the loop. This function initializes the structures needed for the runtime module of ATLaS. In following versions of ATLaS, the addition of this function is expected not to be further necessary.

```
#define MAX 100
#define NITER 30000
int i, k, aux, var1, var2;
int array[MAX];

specbegin (NITER);

#pragma omp parallel for default(none) schedule(static) \
  private(i, k, aux) \
  shared(array) \
  speculative(var1, var2)
for ( i = 0 ; i < NITER ; i++ ) {

  if (i == 3000) { k = var2; }
  if (i == 600) { k = var1; }

  for (k = 0; k < array[i % MAX] + NITER; k++) {

    if (k >= 29900) { var2 = k + array[(i + k) % MAX]; }
    if (k <= 200) { var1 = array[i % MAX]; }
    aux = (k + NITER) % 100000;
  }

  if (i == NITER-1) { var2 = var1; }
}
```

   In this case, `var1` and `var2` are speculative because they are read and written, therefore they may produce dependence violations.

4. Execute ATLaS with the right values for each argument. Details for each argument are found in the previous section.

```
$ ./atlas --threads 4 --block 500 --maxpointer 2 --maxiter 30000
          --mask 7 -c synthetic_sequential.c
```

ATLaS produces the following output for the code example shown:

```
****************************************
Analyzing 'main()'...
****************************************
Adding specinit() at the beginning of the function.

OpenMP pragma omp PARALLEL detected in line 69!
  Another kind of clause
  Private Clause: 'i'
  Private Clause: 'k'
  Private Clause: 'aux'
  Shared clause: 'array'
  Speculative clause: 'j'
  Speculative clause: 'l'

Searching FOR directive associated with the PARALLEL directive...
Number of speculative variables = 2
   Variable : j. Type: int.
   Variable : l. Type: int.
Replacing original FOR loop for a speculative version...

Reading from speculative variable:  k = l;
Reading from speculative variable:  k = j;
Writing into speculative variable:  l = D.3239 + k;
Writing into speculative variable:  j = array[D.3242];
Reading from speculative variable:  l = j;
Writing into speculative variable:  l = D.3324;
Adding engine's functions pre-loop

Plugin finished in function 'main()'.
```

5. ATLaS generates a binary file functionally equivalent to the original application whose execution will run in parallel using the number of threads specified in the compilation.

   The execution of this file prints in first place a resume of the values selected for each option of ATLaS.

```
$ ./synthetic_sequential-auto
************************************************************
Speculative code executed with the following parameters:
Threads = 4
Maximum number of iterations = 30000
Block size = 500
Mask size = 7
Initial capacity reserved to pointers in a matrix = 2
************************************************************
...(application output)
```

A Scheduling strategies

When specifying sizes to the iterations chunks chosen to be speculatively executed, either a static, or a dynamic strategy can be taken. The steps to follow when the static scheduling is desired are well described in the previous installation guide: users only have to specify the positive number of iterations through the parameter `-b` when compiling. Dynamic scheduling is a more complex way

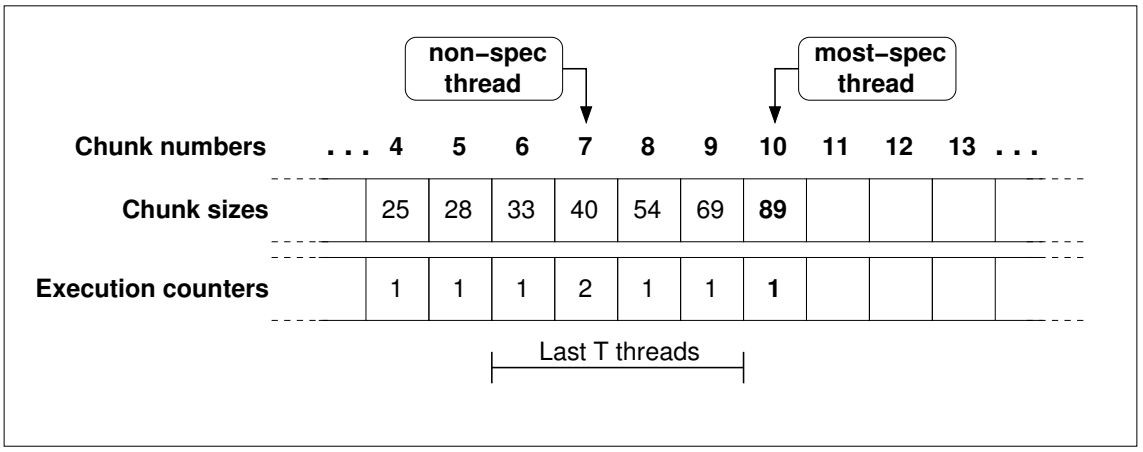

Figure 3: Just-In-Time dynamic scheduling, conservative approach.

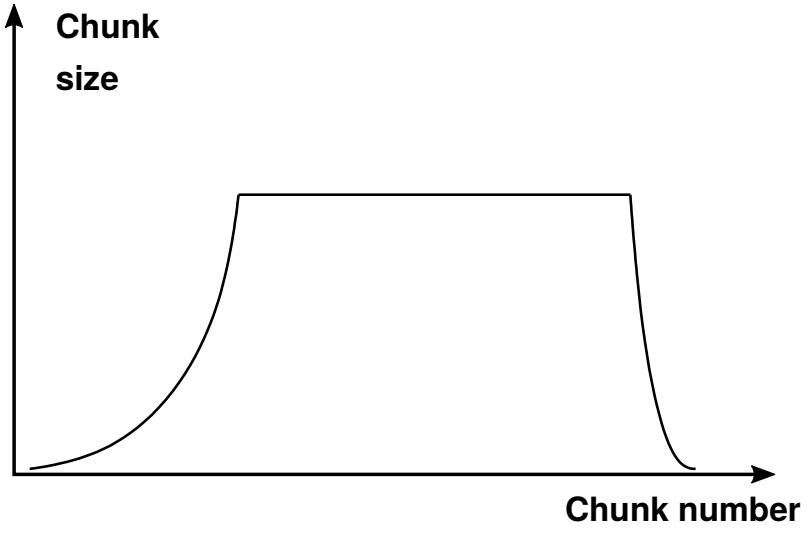

Figure 4: Meseta scheduling.

to fix the number of iterations because it varies dynamically depending on several factors such as dependence violations produced or the place that a given iteration occupies in the whole set. The main advantage of static solutions is that no calculations have to be done in order to obtain the next chunk size, however they have to be carefully tuned through experimentation so that a good performance can be achieved. Figure 3 shows an example of this kind of scheduling. Below are detailed the dynamic scheduling approaches implemented in ATLaS.

- **Meseta scheduling.** This approach follows the assumption that dependence violations are located in the initial iterations of loops. Therefore, it dynamically increase sizes given according to the number of iteration to execute until achieving a fixed limit when sizes are static. In addition, when iterations to execute are closely to the end of the loop, sizes are decreased in order to leverage as much as possible the threads in execution. Figure 4 shows the evolution of the size of the chunk of iterations given in an execution. More details of this strategy can be found in [6].

- **Just-In-Time scheduling.** This strategy takes into account the number of re-executions

$C = \left\lceil \frac{\ln(i) \cdot \ln(N)}{\overline{e}(t)} \right\rceil$	$C = \left\lceil \frac{\ln(i)^2 \cdot \ln(N)}{\overline{e}(t)} \right\rceil$
(a)	(b)

Figure 5: Functions that define Just-In-Time scheduling.

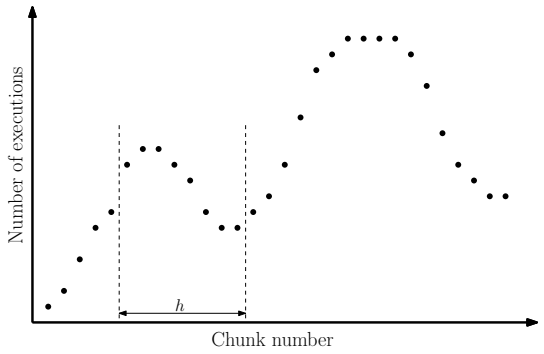

Figure 6: Moody scheduling approach.

performed in the previous threads, the number of iterations of the loop, and the number of the first iteration to be executed in order to obtain the size of the next chunk of iterations. This approach has two flavours, one more optimistic which let sizes to high faster, and another which increase slower the sizes given. Functions showed in the figure 5 details this description. More details of this strategy can be found in [7].

- **Moody scheduling.** This approach tries to obtain the best size of chunk each time mainly taking into account the number of times that the last chunks have been squashed and re-executed due to dependence violations, the average number of executions of the last chunks, and the tendency of these re-executions, i.e., whether previous executions tend to produce dependence violations or not. This solution also requires programmers to define some parameters to set how optimistically decrease, or increase, sizes of iterations. Moreover, users have to define the average number of re-executions considered acceptable, the number of previous threads to take into account in the calculations, and the size of the first iteration to initialize the scheduler. These parameters can be modified in the file *dynScheduler.h*, located in *specprag*. Figure 6 and table 1 describe how this solution works. More details of this strategy can be found in [5].

Both Just-in-Time and Moody scheduling have also two approaches to develop their task. If no dependences arose during the parallel execution, the size of the following chunk would be calculated only once, that is, just before issuing its execution. Otherwise, if the execution of the chunk fails, it gives the runtime system an opportunity to adjust its calculation by calling the scheduling function with updated runtime information. This leads to two different ways to use the scheduling function:

	$meanH \approx 1$	$meanH \approx accMeanH$	$meanH \approx maxMeanH$	$meanH > maxMeanH$
$d \to -1$	$\uparrow$	$\nearrow$	$=$	1
$d \approx 0$	$\nearrow$	$=$	$\searrow$	1
$d \to 1$	$=$	$\searrow$	1	1

Table 1: Changes on the following chunk sized according to the tendency d and the mean of re-executions $meanH$ parameters. The label $maxMeanH$ refers to the highest number of executions acceptable, and $accMeanH$ refers to a number of acceptable re-executions.

1. To calculate the size of the following chunk only the first time this particular chunk will be issued. Subsequent re-executions will keep the same size.

2. To re-calculate the size of the following chunk each time the chunk is scheduled. This solution is called adaptive scheduling.

The advantage of adaptive over dynamic scheduling is that the first calculation of the chunk size may rely on incomplete information, since some or all of the previous chunks are still being executed, and therefore they may suffer additional squashes. Adaptive scheduling will always reconsider the situation using updated data. Naturally, this comes at the cost of additional calls to the scheduling function.

A.1 How to use these mechanisms

As seen previously, using static scheduling only requires to set the size with the parameter -b. The dynamic flavour of scheduling can be used in a similar way through the use of negative numbers. Thus, the following numbers define each mechanism implemented so far.

- -1: To use the Meseta scheduling.
- -2: To use the dynamic standard version of the Just-in-Time scheduling with the function which increases slower.
- -3: To use the dynamic adaptive version of the Just-in-Time scheduling with the function which increases slower.
- -4: To use the dynamic standard version of the Just-in-Time scheduling with the function which increases faster.
- -5: To use the dynamic adaptive version of the Just-in-Time scheduling with the function which increases faster.
- -6: To use the dynamic standard version of the Moody scheduling.
- -7: To use the dynamic adaptive version of the Moody scheduling.

B Papers related with ATLaS

- **An OpenMP extension that supports thread-level speculation**. Sergio Aldea, Alvaro Estebanez, Diego R. Llanos, Arturo González-Escribano. IEEE Transactions on Parallel and Distributed Systems, 2015.

- **New Data Structures to Handle Speculative Parallelization at Runtime**. Alvaro Estebanez, Diego R. Llanos, Arturo González-Escribano. International Journal of Parallel Programming, 2015.
- **Compile-Time Support for Thread-Level Speculation**. Sergio Aldea. PhD thesis, Valladolid, Spain. July, 2014.
- **Support for Thread-Level Speculation into OpenMP**. Sergio Aldea, Diego R. Llanos, Arturo González-Escribano. Proceedings of the 8th International Workshop on OpenMP (IWOMP 2012), Rome, Italy, June 11-13, 2012. Springer, Lecture Notes in Computer Science, 2012, Volume 7312, ISBN 978-3-642-30960-1, pages 275-278
- **Extending OpenMP to support speculative execution**. Sergio Aldea, Diego R. Llanos, Arturo Gonzalez-Escribano. HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures. 2012.

References

[1] OpenMP specification, version 4.0. http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf.

[2] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, San Francisco, California, USA, 1 edition, October 2000.

[3] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[4] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, March 1998.

[5] Alvaro Estebanez, Diego R. Llanos, David Orden, and Belen Palop. Moody scheduling for speculative parallelization. In *Proceedings of the 21th International Conference on Parallel Processing*, Euro-par'15, Berlin, Heidelberg, 2015. Springer-Verlag.

[6] Diego R. Llanos, David Orden, and Belén Palop. Meseta: A new scheduling strategy for speculative parallelization of randomized incremental algorithms. In *Proc. 2005 ICPP Workshops (HPSEC-05)*, pages 121–128, Oslo, Norway, June 2005. ISBN 0-7695-2381-1, IEEE Press.

[7] Diego R. Llanos, David Orden, and Belen Palop. Just-in-time scheduling for loop-based speculative parallelization. In *PDP 2008*, pages 334–342, 2008.