A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP

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

Dpto. Informática, Universidad de Valladolid
Campus Miguel Delibes, 47011 Valladolid, Spain
{sergio,alvaro,diego,arturo}@infor.uva.es

Abstract. In this paper we propose a compile-time system that adds support for Thread-Level Speculation (TLS) into OpenMP. Our solution augments the original user code with calls to a TLS library that handles the speculative parallel execution of a given loop, with the help of a new OpenMP `speculative` clause for variable usage classification. To support it, we have developed a plugin-based compiler pass for GCC that augments the code of the loop. With this approach, we only need one additional code line to speculatively parallelize the code, compared with the tens or hundreds of changes needed (depending on the number of accesses to speculative variables) to manually apply the required transformations. Moreover, the plugin leads to a faster performance than the manual parallelization.

Keywords: Thread-Level Speculation, TLS, OpenMP, Source code generation, GCC plugin.

1 Introduction

The availability of multicore architectures allows users not only to run several applications at the same time, but also to run parallel code. However, the manual development of parallel versions of existent, sequential applications is an extremely difficult task because it needs (a) an in-depth knowledge of the problem to be solved, (b) understanding of the underlying architecture, and (c) knowledge of the parallel programming model to be used. Many parallel languages and parallel extensions to sequential languages have been proposed to exploit the capabilities of modern multicore system. The most successful proposal in the domain of shared memory system is OpenMP [1], a directive-based parallel extension to sequential languages as Fortran, C, or C++, that allows the parallelization of user-defined code regions. OpenMP does not ensure the correct execution of the code according to sequential semantics, making the programmer responsible for such tasks. Possible dependence violations that may occur between iterations during execution need to be addressed by the programmers.

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 234–245, 2014.

On the other hand, automatic parallelization offered by compilers only extracts parallelism from loops when the compiler can assure that there is no risk of a dependence violation at runtime. Only a small fraction of loops falls into this category, leaving many potentially parallel loops unexploited. Thread-Level Speculation (TLS) techniques allow the extraction parallelism from fragments of code that can not be analyzed at compile time, namely, the compiler can not ensure that the loop can be safely run in parallel. TLS can deal with those situations in which dependence violations may occur, leading the parallel loop to correctly finalize its execution. The main problem of these techniques is that the code needs to be manually augmented in order to handle the speculative execution and monitor the possible dependences. Programmers have to modify those accesses to variables that may lead to a dependence violation, also known as *speculative* variables.

In our prior work [2], we proposed the idea of extending OpenMP to allow the user to mark variables as speculative, and a compile-time system that enables the automatic transformation of the code to support its execution by a TLS runtime library. The transformations proposed are transparent to programmers, who do not need to know anything about the TLS parallel model. These key aspects of our proposal solve the problems stated above. Programmers only have to classify variables depending on their accesses, letting our solution perform all the changes needed in the source code. To do so, we have proposed a new OpenMP clause (`speculative`) to handle those variables whose use may lead to any dependence violation.

In this paper we present the development of a GCC plugin-based compiler pass to give support to the new clause `speculative` into GCC OpenMP implementation. This pass transforms the loop with the corresponding `omp parallel for` directive, inserting the runtime TLS calls needed to (a) distribute blocks of iterations among processors, (b) perform speculative loads and stores of `speculative` variables (pointed out using the new clause), and (c) perform partial commits of the correct results calculated so far. The TLS runtime library used [3] is based on the same design principles as the speculative parallelization library developed by Cintra and Llanos [4,5].

Our experimental comparison between manual and automatic transformation of the user code shows that the runtime performance of the code generated by our compilation system is even faster than the performance returned by a manually-transformed code. Besides, the number of lines that should be changed by the programmer to speculatively parallelize a loop is reduced to only one, instead of the significant amount of lines needed in a manual intervention, which depends on the number of accesses to speculative variables inside the loop.

2 Thread-Level Speculation in a Nutshell

Speculative parallelization (SP), also called Thread-Level Speculation (TLS) or Optimistic Parallelization [6] assumes that sequential code can be optimistically executed in parallel, and relies on a runtime monitor to ensure that no dependence violations are produced. A dependence violation appears when a given

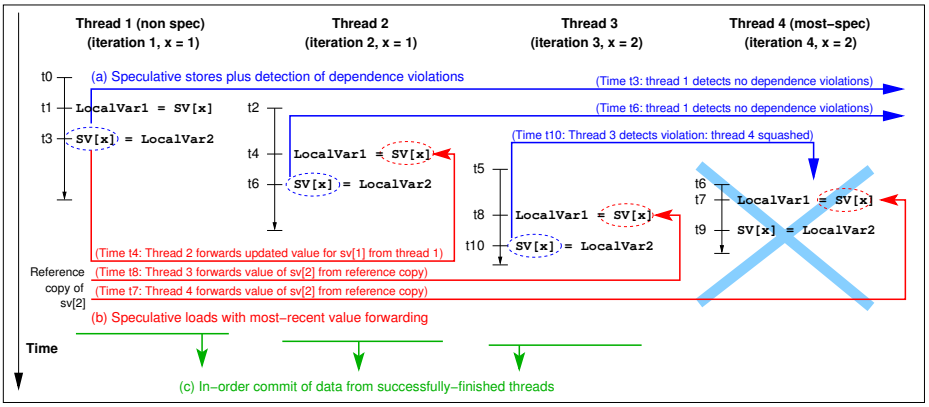

Fig. 1. Example of speculative execution of a loop and summary of operations carried out by a runtime TLS library

thread generates a datum that has already been consumed by a successor in the original sequential order. In this case, the results calculated so far by the successor (called the offending thread) are not valid and should be discarded. Early proposals [7,8] stop the parallel execution and restart the loop serially. Other proposals stop the offending thread and all its successors, re-executing them in parallel [4,9,10,11].

Figure 1 shows an example of thread-level speculation. The figure represents four threads executing four consecutive iterations, and the sequence of events when the loop is executed in parallel. The value of `x` was not known at compile time, so the compiler was not able to ensure that accesses to the `SV` structure do not lead to dependence violations when executing them in parallel. Note that, at runtime, the actual indexes of `SV[x]` are known.

Speculative parallelization works as follows. Each thread maintains a version copy of the entire `SV` vector, called the *speculative data structure*. At compile time, all reads to `SV` are replaced by a function that performs a *speculative load*. This function obtains the most up-to-date value of the element being accessed. This operation is called *forwarding*. If a predecessor (that is, a thread executing an earlier iteration) has already defined or used that element then the value is forwarded (as Thread 2 does in Fig. 1). If not, then the function obtains the value from the main copy of the vector (as Thread 3 does in the figure).

Regarding modifications to the shared structure, all write operations should be replaced at compile time by a *speculative store* function. This function writes the datum in the version copy of the current processor, and ensures that no thread executing a subsequent iteration has already consumed an outdated value for this structure element, a situation called "dependence violation". If such a violation is detected, the offending thread and its successors are stopped and restarted.

If no dependence violation arises for a given thread, it should *commit* all the data stored in its version copy to the main copy of the speculative structure. Note that commits should be done in order, to ensure that the most up-to-date

```
#pragma omp parallel for default(none) private(i, Q, aux) speculative(a)
  for (i = 0; i < MAX; i++) {
      Q = i % (MAX) + 1;
      aux = a[Q-1];

      Q = (4 * aux) % (MAX) + 1;
      a[Q-1] = aux;
  }
```

Fig. 2. Example of FOR loop annotated with the `speculative` clause

values are stored. After performing the commit operation, a thread can receive a new iteration or block of iterations to continue the parallel work.

Finally, the original loop to be speculatively parallelized should be augmented with a scheduling method that assigns to each free thread the following chunk of iterations to be executed. If a thread has successfully finished a chunk, it will receive a brand new chunk not executed yet. Otherwise, the scheduling method may assign to that thread the same chunk whose execution had failed, in order to improve locality and cache reutilization.

In short, at compile time TLS requires that the original code be augmented to perform speculative loads, speculative stores, and in-order commits. In addition, it also requires that the loop structure be rearranged in order to follow the re-execution of squashed operations. Without computational support, this is a task that programmers have to carry out manually. Our plugin solves this limitation, automatically performing all these changes required by the TLS runtime library that gives support. Programmers just need to use the new OpenMP clause we have proposed to point out which variables may lead to a dependence violation.

3 New OpenMP Clause: `speculative`

The new OpenMP clause we defined [2] is called `speculative`, and it needs to be used as part of a `parallel for` directive. The new clause is used as follows, where `list` contains variables that may lead to any dependence violation:

```
#pragma omp parallel for speculative (list)
  for-loop
```

With this extension, programmers are able to write OpenMP programs as usual, but annotating those variables that could lead to a dependence violation as `speculative`. With this method, programmers do not have to take care of handling these violations, being the speculative engine the responsible of such task. Once a programmer annotates each variable to its type, the plugin augments the code to add support for the TLS runtime library.

Figure 2 shows an example of the use of the proposed clause. Variable `i` is private, since it is the variable that controls the iterations of the FOR loop. Variables `Q` and `aux` are private, because they are always written before being read in the context of an iteration. Finally, variable `a` is speculative, because

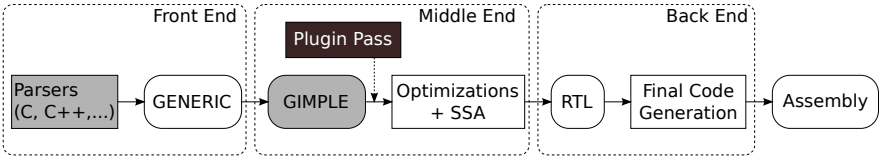

Fig. 3. GCC Compiler Architecture [12,13] simplified. The main OpenMP related components, highlighted in grey, are the C, C++ and Fortran parsers, and the GIMPLE IR level. The black box represents the location of our plugin pass.

accesses to this variable can lead to dependence violations. Eventually, a particular iteration will read from `a` a non-updated value and therefore the execution will be incorrect. As we have seen in Sect. 2, a speculative scheme would allow this loop to finish correctly.

4 Parsing the New `speculative` Clause

Although the plugin mechanism enables us to perform all the changes needed by the TLS runtime library, plugins do not allow the extension of the parsed language. Therefore, adding a new OpenMP clause recognized by GCC requires not only the creation of a plugin, but also modifying the GCC code itself. In order to parse the new clause `speculative`, we have extended the GNU OpenMP (GOMP), an OpenMP implementation for GCC. The main parts of the GCC architecture related within OpenMP are highlighted in grey in Fig. 3. GOMP has four main components [14]: parser, intermediate representation, code generation, and the runtime library called `libGOMP`. In relation to GOMP, we have focused on modifying its parsing phase and the intermediate representation (IR). The generation of new code to support TLS is located in the plugin developed, and mainly this new code consists of calls to the TLS library functions needed for the speculative execution.

The parser identifies OpenMP directives and clauses, and emits the corresponding GENERIC representation. We have modified the C parser and the IR to add support for the new clause `speculative`. First, we have created the GENERIC representation of the new clause like other standard clauses. Then, the compiler has been prepared to recognize and parse the clause as part of the parallel loop construct. When the new clause has been parsed and the IR is generated, our plugin detects the clause and starts all the transformations needed on the code.

5 Plugin-Based Compiler Pass Description

Once the new clause proposed is recognized by GCC, programmers can set the `speculative` variables, and the plugin developed can augment the original code.

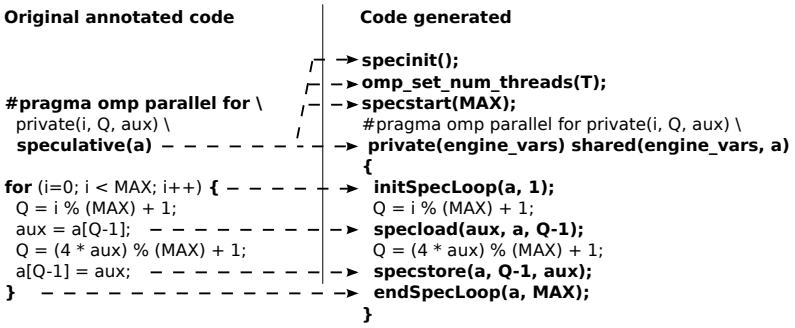

Fig. 4. Code of Fig. 2 annotated and the resulting, transformed pseudo-code. `initSpecLoop()` and `endSpecLoop()` are macros that expand to more code, hidden here for legibility reasons.

The use of plugins provides several advantages, such as faster building of prototypes, easier modifications and contributions, and the use of GCC as a research compiler. Using plugins programmers can load external shared modules, which are inserted as new passes into the compiler. We will take advantage of this feature to develop our plugin and add support to TLS into OpenMP. We have chosen to modify GCC because it is a mainstream mature compiler, and we expect that extending GCC functionalities will have a higher impact. Moreover, as long as GCC supports more than 30 architectures, this increases the compatibility of our proposal.

The new pass is added once the compiler has transformed the code into GIMPLE, and just before GCC does the first pass related to OpenMP (`omplower`). Therefore, our pass is added before `pass_lower_omp` in `passes.c`. In this point, we have the code in a GIMPLE representation, and the FOR-loop marked with the `omp parallel for` directive preserves all the clauses written by the programmer. Therefore, we have the information about which variables are shared, private, and speculative, the latter thanks to the new clause proposed. After this pass, GCC processes speculative variables as shared, while their handling as speculative will be carried out at runtime by the TLS library.

Figure 4 shows a brief example of the transformations made by the plugin. The parser detects the new speculative clause, and the new compiler pass automatically performs all the transformations needed to speculatively parallelize the loop. If the plugin does not find the `speculative` clause on the pragma, the semantic of the loop remains identical to any other standard OpenMP loop. With the list of variables and data structures that should be speculatively updated, the plugin replaces each read of one of these variables or data element with a `specload()` function call. Similarly, all write operations to speculative variables are replaced with a `specstore()` function call. Loads or stores involving other variables do not require additional changes in the code, since all flavors of *private* and *shared* variables keep their respective semantics in the context of a speculative execution. The plugin also adds all the structures and functions needed to run the TLS system that parallelize the code. This process is

completely transparent to programmers, shielding them from the intricacy of the underlying speculative parallelizing model. They only have to label the variables involved in the target loop as *private* or *shared*, as with any other OpenMP program, and label as *speculative* those variables that can lead to any dependence violation.

Once the plugin has transformed the loop, GCC operation continues with the next passes. When the compilation ends, the resulting binary file is prepared to run speculatively.

5.1 Interface with the TLS Runtime Library

The plugin-based compiler has to augment the code with the functions and structures needed for the speculative execution, and defined by the TLS runtime library. The library used [3] is largely based in Cintra's and Llanos' work (see [4,5] for details). The plugin has to replace accesses over speculative variables with `specstore()` or `specload()` functions. This task requires the plugin to detect code lines where a write and/or read is applied, to extract the type of the speculative variable or the particular field of an speculative structure, and to perform the changes needed, including the addition of new variables to handle the temporal values required. The plugin is also able to detect reductions applied on speculative variables, replacing them by the appropriate function calls to the TLS runtime library that handle them.

The TLS runtime library also requires other functions and structures, some of them sketched in Fig. 4, that the plugin has to correctly insert into the code. Regarding the original loop, the plugin replaces the parallelized loop with a new loop that drives the speculative execution. This new loop iterates over the threads, and has the same body as the original, although it is augmented with extra code that ensures the correct distribution of iterations over the threads, and commits the data stored in the speculative variables. The definition of the new loop and the code inserted before the body of the original loop is gathered in the macro `initSpecLoop()` (Fig. 4) for simplicity. The code lines which are required to be inserted after the body of the original are gathered in the macro `endSpecLoop()`.

Besides modifying the target loop and its body, the plugin also adds three functions before the loop. The first one, `specinit()`, initializes the TLS runtime library, and it has to be called once in a program. Therefore, the plugin detects the `main` function of a program, and adds the call to `specinit()` as the first statement. The other two functions required are `specstart()` and `omp_set_num_threads()`, which are always placed before each parallelized loop. `specstart()` initializes the execution of the following parallel loop, while `omp_set_num_threads()` set the number of threads for its parallel execution.

5.2 Handling Complex Statements

The plugin is able to handle all definitions and uses of scalar variables, not only simple assignments. This includes dealing with complex statements, that are

required to maintain the same order in which the multiple speculative loads and stores are executed. The plugin first resolves the loads, creating new temporal variables that take part of the expression that assign a value to the speculative variable. After replacing the loads for the corresponding `specload()`, the plugin handles the store into the speculative variable by placing a `specstore()`. An example of this situation is a writing into a speculative array with a speculative variable as index.

Programmers may write other constructs that the plugin can deal with, such as assignments from one pointer to another, accesses involving directions or the data pointed by the pointer, assignments between entire data structures or only fields of those structures, and speculative variables involved in casting operations.

5.3 Using the Plugin to Compile the User Code

From the point of view of programmers, to speculatively parallelize a source code with our system they only have to add an OpenMP parallel loop directive and set a few parameters to the compiler. First, programmers should add the OpenMP directive in the target loop, and classify its variables according to their usage in `private` and its variants, `shared`, `speculative`.

Second, to compile the program, programmers should indicate the size of the block of iterations that will be issued for speculative execution, as well as the number of threads they want to launch. We have developed a wrapper script that launches the compilation of the plugin plus the speculative engine, and it is run as follows:

```
$ atlas -threads T -block B -c example.c
```

Just by using the speculative clause, a programmer can speculatively parallelize a code, while the rest of transformations needed are transparently performed by the plug-in and the compiler.

6 Validation

In order to check the correctness of our plugin and the code that it generates, we have developed a battery of regression tests. These regression tests include more than 50 loops with one or more speculative variables, scalar variables, pointers, elements from multidimensional arrays, or elements from data structures. They also cover situations with speculative variables that have different types, and loops executing a number of iterations that are variable and defined in runtime. These regression tests are developed with the aim of covering possible situations that we can find in a source code, allowing us to check the correction of the plugin before addressing real applications. One of these tests is shown in Fig. 5, where we check the correct operation of the plugin with speculative accesses over variables with different sizes, and speculative accesses to data structures, including assignments between entire structures.

We have also tested the plugin with real-word applications that are not parallelizable at compile time due to several data dependencies, requiring runtime

```
1:  int i, j, array[MAX], array2[MAX];
2:  struct card{ int field; };
3:  struct card p1 = {3}, p2 = {99999}, p3 = {11111};
4:  char aux_char = 'a';
5:  double aux_double = 3.435;
      ...
6:  #pragma omp parallel for default (none) private(i,j) shared(array1, p2) \
7:    speculative(p1, p3, aux_char, aux_double, array2)
8:  for ( i = 0 ; i < NITER ; i++ ) {
9:    for ( j = 0 ; j < NITER ; j++ ) {
10:     if (i <= 1000) p1.field = array[i % 4] + j;
11:     else array2[i % 4] = p1.field;

12:     if (i > 2000) aux_char = i %20 + 48 + aux_char % 48;
13:     else aux_char = i % 20 + array[i % 4] % 10 + 48;

14:     if (i > 1500) aux_double = array[i % 4] / (i+1) + aux_double;
15:     else array2[i % 4] = (int) (aux_double / i*j) + (array2[(i+j) % 4] + i*j) % 1234545;

16:     if (i*j > 10000) p1 = p2; else p3 = p1;
17:   }
18: }
```

Fig. 5. Example of the kind of situations that the plugin can deal with

speculative parallelization. These applications are the 2-dimensional Convex Hull problem (2D-Hull) [15], the Delaunay Triangulation using the Jump-and-Walk strategy [16], the 2-dimensional Minimun Enclosing Circle (2D-MEC) problem [17], and a C implementation of TREE [18]. The plugin is able to speculatively parallelize the target loops in these benchmarks correctly.

7 Relative Performance and Programmability

Automatic parallelization moves the workload from the programmer to the compiler. This is a great deal if the performance achieved by the automatic approach is as good as the obtained by the manual one. In Table 1 we summarizes the relative performance of both automatic and manual approaches. Note that the numbers are not the speedups obtained, but their relative comparison. The experimental results show that the automatic transformation leads to a faster code than the one obtained by manually replacing accesses to speculative variables with function calls. The reason is that the manual transformation of the source code may prevent the application of certain compiler optimizations. In contrast, our automatic transformation system works with the GIMPLE intermediate representation, after the first phases of the compiler have been triggered. The performance achieved by the applications parallelized using the `speculative` clause is 24% faster than the performance scored by the manual parallelization on geometric average. The maximum speedup achieved in each application is shown in Table 1. Data have been obtained running each experiment three times, and then obtaining the average. Experiments were carried out on a 64-processor server.

Regarding programmability, using the proposed clause dramatically reduces the number of lines required in comparison with the former, manual way of

Table 1. Number of lines required in both automatic and manual approaches, their relative performance, and the maximum speedup achieved for each application, where 'p' indicates the number of processors. 2D-Hull and MEC are executed with a 10M-points dataset, Delaunay with a 1M-points dataset, and TREE with a dataset of 4096 nodes.

	# of lines		Relat. perfor. by # of proc.					Maximum
Application	**Auto**	**Man.**	**8**	**16**	**32**	**48**	**64**	**Speedup**
2D-Hull	1	139	1.301	1.288	1.404	1.287	1.205	**12.97** (56p)
Delaunay	1	191	1.261	1.255	1.212	1.106	1.122	**3.11** (32p)
2D-MEC	1	50	1.335	1.369	1.416	1.285	1.410	**2.63** (24p)
TREE	1	42	1.125	1.106	1.077	1.198	1.218	**6.47** (40p)
Geom. Mean	1	86	1.253	1.251	1.269	1.217	1.234	**5.12**

parallelizing a code using the TLS library. Parallelizing a code with the proposed `speculative` clause only requires one line of code –the modified OpenMP pragma–, while parallelizing the same code manually requires tens to thousands new lines, depending on the number of accesses to speculative variables.

Such reduction in the number of required lines is not the only advantage. Parallelizing the code with the plugin only requires classifying the variables within the loop according to their usage, whereas the manual alternative is not only a hard, error-prone task, but also a deep knowledge of the TLS library.

8 Related Work

As far as we know, there are not proposals to extend OpenMP to support software-based TLS. Instead, in the literature there are some approaches that extend OpenMP to support Transactional Memory (TM) [19], and hardware speculation, such as the pragma implemented in the IBM C/C++ compiler for Blue Gene/Q [20]. Early works propose the use of pragma directives, OpenMP-based [21] or not [22], to enable speculative parallelism at a hardware level. However, these proposals do not define any particular new OpenMP directive.

More recently, proposals are focused on TM. Proposals such as [23,24,25] extend OpenMP to support TM, providing new directives and clauses in order to mark and wrap critical sections A similar proposal is Soc-TM [26], but focused on TM programming for embedded systems.

Although some of these proposals implement the code generation required, as far as we know, there are not any specific work that proposes or implements OpenMP extensions to support Thread-Level Speculation. This empty hole is what we aim to fill with this paper, proposing a new OpenMP clause, and a plugin-based compiler pass that supports the TLS runtime library [3] based on the technique that Cintra and Llanos' speculative engine [4,5] implements.

Other research groups have also experimented with the GCC (since version 4.5) plugin mechanism. Among them, some plugins are designed to make the development of GCC plugins easier than with the standard procedure, such as

GCC Melt [27], MilePost GCC [28], or a GCC Python plugin [29]. We decided to develop our transformation system as a GCC plugin in order to avoid dependencies to third-party, not-so-mature systems.

9 Conclusions

We present a compile-time system that automatically adds the code needed to handle the speculatively parallel execution of a loop, and uses a new OpenMP clause (`speculative`) to find those variables that may lead to a dependence violation. We have used the plugin mechanism provided by GCC to support the new OpenMP clause. Using this clause, programmers can point out the speculative variables, and they do not need to know anything about the speculative parallelization model. In order to parallelize a code, programmers are only required to add one line (the OpenMP pragma plus the `speculative` clause), instead of the significant amount of lines required by the manual parallelization, which depends on the number of accesses to speculative variables. Moreover, the performance of the generated codes is even faster that the manually parallelized codes.

We expect that implementing this new clause in a mainstream compiler, together with the automation of the whole process of the speculative parallelization, will help Thread-Level Speculation to be mature enough for its inclusion in mainstream compilers.

Acknowledgments. This research is partly supported by the Castilla-Leon Regional Government (VA172A12-2, PIRTU); Ministerio de Industria, Spain (CENIT OCEANLIDER); MICINN (Spain) and the European Union FEDER (MOGECOPP project TIN2011-25639, CAPAP-H3 network TIN2010-12011-E, CAPAP-H4 network TIN2011-15734-E).

References

1. Chandra, R., Menon, R., et al.: Parallel Programming in OpenMP, 1st edn. Morgan Kaufmann (October 2000)
2. Aldea, S., Llanos, D.R., González-Escribano, A.: Support for thread-level speculation into OpenMP. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 275–278. Springer, Heidelberg (2012)
3. Estebanez, A., Llanos, D.R., Gonzalez-Escribano, A.: New Data Structures to Handle Speculative Parallelization at Runtime. In: Proceedings of HLPP 2014 (2014)
4. Cintra, M., Llanos, D.R.: Toward efficient and robust software speculative parallelization on multiprocessors. In: Proceedings of PPoPP 2003 , pp. 13–24 (June 2003)
5. Cintra, M., Llanos, D.R.: Design space exploration of a software speculative parallelization scheme. IEEE Trans. Parallel Distrib. Syst. 16(6), 562–576 (2005)
6. Kulkarni, M., Pingali, K., et al.: Optimistic parallelism requires abstractions. In: Proceedings of PLDI 2007, pp. 211–222 (2007)
7. Gupta, M., Nim, R.: Techniques for speculative run-time parallelization of loops. In: Proc. of the 1998 ACM/IEEE Conference on Supercomputing, pp. 1–12 (1998)

8. Rauchwerger, L., Padua, D.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In: Proceedings of PLDI 1995, pp. 218–232 (1995)
9. Dang, F.H., Yu, H., Rauchwerger, L.: The R-LRPD test: Speculative parallelization of partially parallel loops. In: Proceedings of 16th IPDPS, pp. 20–29 (2002)
10. Xekalakis, P., Ioannou, N., Cintra, M.: Combining thread level speculation helper threads and runahead execution. In: Proceedings of ICS 2009, pp. 410–420 (2009)
11. Gao, L., Li, L., et al.: SEED: A statically greedy and dynamically adaptive approach for speculative loop execution. IEEE Trans. Comput. 62(5), 1004–1016 (2013)
12. GNU Project: GCC internals (2013), `http://gcc.gnu.org/onlinedocs/gccint/`
13. Novillo, D.: GCC an architectural overview, current status, and future directions. In: Proceedings of the Linux Symposium, Tokyo, Japan, pp. 185–200 (September 2006)
14. Novillo, D.: OpenMP and automatic parallelization in GCC. In: Proceedings of the 2006 GCC Developers' Summit, Ottawa, Canada (2006)
15. Clarkson, K.L., Mehlhorn, K., Seidel, R.: Four results on randomized incremental constructions. Comput. Geom. Theory Appl. 3(4), 185–212 (1993)
16. Devroye, L., Mücke, E.P., Zhu, B.: A note on point location in Delaunay triangulations of random points. Algorithmica 22, 477–482 (1998)
17. Welzl, E.: Smallest enclosing disks (balls and ellipsoids). In: Maurer, H. (ed.) New Results and New Trends in Computer Science. LNCS, vol. 555, pp. 359–370. Springer, Heidelberg (1991)
18. Barnes, J.E.: TREE. Institute for Astronomy. University of Hawaii (1997), `ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/`
19. Larus, J., Kozyrakis, C.: Transactional memory. Commun. ACM 51(7), 80–88 (2008)
20. IBM: Thread-level speculative execution for C/C++. IBM XL C/C++ for Blue Gene, Tech. report (2012)
21. Packirisamy, V., Barathvajasankar, H.: OpenMP in multicore architectures. University of Minnesota, Tech. Rep (2005)
22. Martínez, J.F., Torrellas, J.: Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In: Proceedings of ASPLOS 2002, pp. 18–29 (2002)
23. Baek, W., Minh, C.C., et al.: The OpenTM transactional application programming interface. In: Proceedings of 16th ISCA, pp. 376–387. IEEE Computer Society (2007)
24. Milovanović, M., Ferrer, R., Unsal, O.S., Cristal, A., Martorell, X., Ayguadé, E., Labarta, J., Valero, M.: Transactional memory and OpenMP. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 37–53. Springer, Heidelberg (2008)
25. Wong, M., Bihari, B.L., de Supinski, B.R., Wu, P., Michael, M., Liu, Y., Chen, W.: A case for including transactions in OpenMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 149–160. Springer, Heidelberg (2010)
26. Ferri, C., Marongiu, A., et al.: SoC-TM: Integrated HW/SW support for transactional memory programming on embedded MPSoCs. In: Proceedings of CODES+ISSS 2011, pp. 39–48. ACM Press (2011)
27. Starynkevitch, B.: MELT: A translated domain specific language embedded in the GCC compiler. In: Proceedings of IFIP DSL 2011, pp. 118–142 (2011)
28. Fursin, G., Kashnikov, Y., et al.: Milepost GCC: machine learning enabled self-tuning compiler. Int'l. Journal of Parallel Programming 39(3), 296–327 (2011)
29. Malcolm, D.: GCC python plugin v0.12.(2013), `https://fedorahosted.org/gcc-python-plugin/` (last visit: May 2014)