An explicit parallelism study based on thread-level speculation

José L. Aguilar
Universidad de Los Andes, CEMISID
Mérida, Venezuela, 5101
aguilar@ula.ve

and

Kahlil Campero
Universidad de Los Andes, CEMISID
Mérida, Venezuela, 5101

Abstract

Developments in parallel architectures are an important branch in computer science. The success of such architectures derives from their inherent ability to improve the program performances. However, their ability to improve the performance on programs depends on the parallelism extraction strategies, which are always limited by the logic of each sequential program. Speculation is the only known alternative to overcome these constraints and increase the parallelism. In this paper, we study the explicit speculative parallelism using a library of thread-level speculation. We present the design of this library and study different speculative models: speculation of decision structures, speculation of loops, and speculation of critical sections. Our study evaluates different cases taken from SPEC CPU 2000, allowing acceleration of about 1.8x in multicore architectures (four core) with coarse-grained multithreaded.

Keywords: Speculative Parallelism, Parallel Programming, Thread-level Speculation.

1 Introduction

The new computational architectures organize additional processing cores on the same chip, which characterize multicore processors and chip multiprocessors. They allow run instructions in the cores concurrently from different threads (multithreads processors) [5, 6, 8, 9, 12]. The programs performance on these processors depends on the ability of these programs to generate multiple threads to run simultaneously. The multicore processors offer a lot of possibilities for parallel programs based on multithreads processing [1, 2, 3, 9, 13].

For that, we need techniques to decompose programs in a multi-threaded execution, which could then take advantage of parallel processing. This process of decomposition of a computation into several parts that could run simultaneously can be carried out automatically and implicitly (at level of the hardware, compiler or operating system) or manually and explicitly (by the programmers, with tools). Regardless of the chosen approach, the use of parallelism will be limited by sequential processing requirements inherent to each program.

Under this circumstance, speculation can be considered an important technological alternative, since it is a way to overcome these constraints and increase the parallelism that can be extracted from any program [1, 3, 4, 8, 9, 12]. Various speculative techniques have been proposed, which have been implemented in hardware, compiler and software. However, these techniques have not been widely adopted in the computer industry. Part of the reasons is the lack of consensus about how to use speculation.

In this paper we study the explicit speculative parallelism problem using a library of thread-level speculation. This library allows the programmers implement speculative techniques in their programs. Section 2 presents the main theoretical aspects of our work. Section 3 defines the state of art in this domain. Section 4 present in detail the library, the speculative models implemented; and finally, section 5 contains some experiments and analysis the results.

2 Theoretical Aspects

2.1 Multicore platforms

A generic multicore processor consists of multiple processing cores, each one with their own caches, both data and instructions. Each core also possesses another cache, called translation lookaside buffer, which is used to improve the speed of the virtual address translation [3]. The different cores share caches in a higher levels. Multicore

processors are Multiple Instructions and Multiple Data (MIMD) architectures with shared memory, all in a single chip.

The shared memory architecture is not always the most efficient multi-core platform on a chip. As the number of cores increases, the management of the shared memory is become complex and the performance deteriorates. Therefore, we can opt for ordering the cores in a distributed memory architecture in a chip. In this case it is called a multiprocessor chip [1, 13]. Apart of the variations in the deployment of the cores on the chip, various technologies have been proposed in order to diversify the possible simultaneous processing in each core. The instruction pipelining is one of the fundamental technologies. This type of parallelism is called instruction level parallelism (ILP), but it incurs in dependency violations. Normally, in order to avoid these violations, the more common is stopping an instruction, while another instruction produces the results which it depends. This degrades the processor performance, and in order to reduce the impact of dependencies, various techniques have been proposed: predictive and speculative [1, 3, 6, 8, 9]. Alternatively, we can exploit the idle time to the processing of instructions from additional threads of execution, this parallelism is called thread-level parallelism (TLP).
Some of the technologies that exploit TLP in processors are [13]:

- The classical pipelining, where all instructions to execute are extracted from a single thread, resulting in an inefficient system that wastes execution cycles.
- The coarse-grained multithreaded, it proposes that when there are high latency instructions, the system process instructions from another active thread, allowing some overlap of latency. The implementation of this form typically has a high cost due to context switching between threads.
- The fine-grained multithreaded or overlap multithreaded, it allows the system simultaneously execute instructions from multiple threads, but with the support of the microprocessor which minimizes the cost of context switching from one thread to another.
- The simultaneous multithreaded (SMT) combines the fine grained multithreaded with superscalar technology. The result is a process where at each cycle; the instructions are extracted from different threads and assigned to multiple cores.

A multicore processor of simultaneous multithreaded divides the threads between the different cores, each one executing simultaneous multithreaded. Moreover, at certain points in the execution of a thread, the operating system may decide migrate from one core to another, in order to improve the use of the processing cores. The Simultaneous multithreaded technology exceeds the efficiency and acceleration of the other approaches presented here [13]. In fact, they can quadruple the efficiency of a basic superscalar and duplicate of a fine-grained multithreaded. For this reason we study this technology in this paper.

2.2 Models and techniques for speculative parallelism

Speculation can be defined as "an execution based on an assumption whose validation will be done later" [1]. If this assumption proves to be false the computation is aborted and the program returns to its original state, whereas if it is valid the computation results are accepted and the program continues. Particularly, the speculative parallelism can be considered as the property of a program by which can be decomposed, using assumptions that will be checked later, in areas suitable to take place independently and simultaneously. The techniques used to take advantage of this parallelism should ensure correct results for the programs, regardless of the assumptions under which it decomposes. Some of these techniques in its thread-level versions are [1, 2, 3, 4, 8, 9, 12]:

- *Speculation of the simple decision structures (SSDS)* is the simultaneous execution of two conditional branches of a program and part of the code that precedes the assessment of the condition, under the assumption that implementation can keep the sequential consistency of the program.
- *Speculation of multiple decision structures (SMDS)*: is the parallel execution of different parts of a program, where each one assumes that a given data will have a specific value.
- *Loops Speculation (EL):* the simultaneous computation of several iterations of a loop.
- *Speculation of critical sections (SCS)* is the simultaneous processing of mutual exclusion sections.
- **Loading ahead speculative**: is the thread execution so that it makes an advance load of data to the cache used by the program.
- *Pre-speculative execution*: is a thread of a program that, under various assumptions, such as the values of some data or branch, executes some instructions of the program.

Each technique has particular consistency constraints, which implies a specific management of the threads execution that each one uses. The implementation of this management does not correspond to the speculative techniques, is a different degree of abstraction, called speculative models [3, 4, 8, 9, 12]. In this sense, speculative models are the base of the speculative techniques, and they guarantee the technical consistencies. Cintra et al propose three basic speculative models [3, 4]: the thread level speculation (TLS), the helper threads (HT), and the runahead execution (RA). Cintra et al [9] define the TLS as a model where threads corresponding to sequential parts

of code are speculatively executed in parallel, under the assumption that these do not violate sequential semantics, including data and control dependencies. In the HT model, additional threads are created; these threads are running in parallel to the program, in order to contribute indirectly to improving its efficiency, reducing the cost of high latency operations that it incurs. To this end, they may bring to the cache data that could be requested by the program, or run early operations of the program. The results are not sent directly to the program, but indirectly.

3 State of art

In the case of multicore processors, we are not only interested in extracting the ILP, but also the parallelism to a higher granularity, enabling the use of SMT technology, as well as additional cores on the chip [12, 13]. This granularity refers to the level of thread of execution, and the corresponding parallelism is called thread-level parallelism (TLP).

One problem is that threads with erroneous speculations could compete for resources, displacing to correct threads, degrading their performances. The impact of this has been studied by Swanson et al., and they showed that the yield obtained by speculation is better in SMT, compared to other architectures [13]. This is due to the interleave instructions from different threads in the SMT processors, the speculation, and the balance of the percentage of instructions executed from wrong and correct threads.

A key aspect of HT is the strategy for creating collaborative threads from a given program. The most common strategy consists in extract for pre-execution, the set of instructions with minimal number of dependence of critical instructions [6,7].

At the level of compilation of TLS there are two tasks [3, 13]: the extraction from a sequence (which is made based on heuristics), and the optimization of the execution of the TLS model. Some computer architectures which support TLS are the Hydra project of Stanford University, STAMPede, and the proposition of Oplinger et al, and Cintra et al [3, 4, 13].

Another proposed approach is to try to overcome the limits of self-parallelization through explicit programming. Cintra et al provide an example of using TLS for explicit parallelization of an algorithm of computational geometry [3, 11]. Prabhu et al. have studied the possibilities to obtain TLS manually, for that, they have carried out minor modifications to benchmarks, obtaining 120% average acceleration for floating point applications and 70% for integer applications [10]. Similarly, Cintra et al. showed that manual changes to the code can lead to improvements in the yield of auto-parallelization with TLS, including accelerations of 454% achieved for various tests [3].

To carry out manual parallelization there are various libraries, tools and models. Ungerer et al. provide an overview of some of them [13]. However, these tools are limited to loops speculation and do not allow other speculative techniques and models. At present, there is not a standard general library that provides the developer the ability to make use of the speculative techniques.

4 A library for explicit parallelism based on thread-level speculation

4.1 Design

The library proposed in [14, 15] offers speculative functionalities to a programmer. The library consists of two layers: an outer layer and an inner layer, which correspond to the programming logic and data management. The outer layer offers a similar interface to the programmers of each type of speculation. The inner layer is responsible preserving the sequential semantics of data and control dependencies. The speculative functionalities were grouped into objects to each type of speculation. Each object (called speculators) maintains the states of the parallel sections along a speculative execution. It is responsible for tracking of the dependencies, through a log of reads and writes, and of the private copies (for SSDS and SMDS) or centralized maintenance of versioned copies (in the case of EL and SCS). Also, it performs dependency checking in each read or write to shared variables, restarting those speculative sections commit a violation.

- *Speculation of simple decision structures (SSDS):* The SSDS consists of four sections, which can be programmed with certain independence: two speculative branches and a pre-branch section responsible for the assessment of the condition that validates one or another branch. The additional section corresponds to one that invokes the speculative execution. For simplicity, we consider that the pre-branch section also is the invoke section. This simplification also applies to SMDS and EL. SSDS begins when the scheduler invokes the pre-section "Start speculative execution", and ends when the same programmer asks "Get speculative execution results." In the meantime, both the pre-branch and the branches can write and read shared data. In this type of speculation, the branches work on private copies of shared data, to avoid dependence violations WAW and WAR. The RAW violations are prevented when the pre-branch writes a shared data and detect the time which branches have read the data with an incorrect value, and restarts them. All the functionalities are offered through an object called especuladorEDS. This object processes the requests it receives. Execution starts at the pre-branch with the invocation of "especularConPreRamaExternaAlGestor ". This function starts the execution of both branches, which begin

to read and write shared data in parallel to the pre-branch. At then end, the pre-branch invokes the function "obtenerResultados".

- *Speculation of multiple decision structures (SMDS):* is identical to SSDS, except that the model uses a greater number of branches. Taking this into account, the user can provide to the pre-branch the ability to add dynamically speculative branches.
- *Loops Speculation (EL):* The LE starts with the invocation to "Start speculative execution", part of the pre-loop section. With that starts parallel execution of a given number of iterations, which can write and read shared data, as well as pre-loop section. For this case, violations that should be checked are: at the moment to read, there is no smaller iterations where the variable has written on the variable to read. At the time to write, there is no greater iterations where the variable has been read or written. Consequently, both for reading and writing, maybe it must restart iterations to ensure correct results. The pre-branch can also add iterations the model and obtain the results of the execution, function must wait for all iterations are completed before returning successfully. The execution starts with the invocation of "especularConPreLazoExternoAlGestor", it creates a thread for each iteration, which goes to the "iteration speculative" state, while the pre-loop switches to the "pre-loop speculative" state. Once they start running, both iterations as the pre-loop can pass to states "Writing in shared variable", "Reading shared variable" and "Aborting execution at given iteration." Additionally, the pre-loop can move to state "Add iteration". When the pre-loop execution ends (invoking "obtenerResultados()"), it must wait for the completion of the iterations, before returning successfully. If any iteration fails to complete, the pre-loop returns "error".
- *Speculation of critical sections (SCS):* consists of two types of parallel sections for each critical section. The first is the invoking section, the second type of section corresponds to the speculative critical sections. The number of instances of the critical sections is given by the number of processes that share the critical section. This model does not consider the code prior to the critical section. The model works as follow: from any thread, run speculatively certain critical section. This invocation should expect the speculations conclude the execution of theirs critical sections. Moreover, a section corresponds to the speculative critical section runs identically to the iterations section in the model EL, limited to perform reads and writes, maintaining sequential consistency relative to the different instances of the critical section.
- *Speculation by HT (SHT):* it has the simplest design, since it does not involve dependency tracking, reboot or complex communication between threads. In this model there are two actors: who invoked, and the program of the thread. The user can invoke two threads from the main program; the first performs a forward load that is useful to accelerate the main program. The second is a pre-execution thread.

4.2 Structural Aspects of the library

The library has a class (see figures 1 and 2), called especuladorHT, to implement speculation by HT, which has a single function: speculative. It receives the instructions and parameters of a partner thread for execution in detached condition. This is the only class implemented decoupled. The other classes can be grouped into three sets. The first corresponds to the speculators, responsible for providing the functionalities for each type of speculation. The second corresponds to the classes that define and contain the different sections to be executed (branches, iterations, pre-loop, pre-branch or critical sections). The third set consisting of the classes that allow tracking of units, ie follow the dependences of reading and writing of data.

For the TLS model there is a general class, called speculator, which encapsulates the information about whether the speculator is making a run or not (see figures 1 and 2). Also, allows access from different threads to a shared data. The EspeculadorED class inherits from the speculator class and it has only a mutex attribute to synchronize access to a variable indicating the shared branches. This class is inherited by especuladorEDS and especuladorEDM, responsible for providing the user with all the instructions that enable the functionalities described previously for these techniques.

Speculator class also is inherited by especuladorEL and especuladorSC classes. The threads that the library uses for executing the models are defined in the class hebraEnElModelo. This class has two kids: hebraReiniciable and seccionPrevia. The first has as attributes the instructions and parameters of the thread and encapsulates the functionalities to initiate, terminate and restart the thread. seccionPrevia execute the previous section. From hebraReiniciable descends two classes: ramaEspeculativa and iteracionEspeculativa. From seccionPrevia descends seccionPreviaEL. ramaEspeculativa, seccionPreviaEL and iteracionEspeculativa indicate when they have successfully completed theirs execution.

The relationship between sections and speculators is of composition. Thus, an especuladorEDS is composed by a previous section and two speculative branches, and an especuladorEDM is composed by a previous section and one or more speculative branches. The EL speculator consists of a previous section and one or more EL speculative iterations. The SCS speculator consists only of speculative iterations, representing the critical sections in the order of invocation.

The last group of classes, corresponds to the data classes that enable dependency tracking (see figure 2). There are two kinds: reading log, with a mutex to synchronize access to the register from any thread. The second class is

copia_de_dato, which relates the original data address with a thread. With this class, SSDS and SMDS speculators store the copies of data for each speculative branch. Also, with this kind speculators EL and SCS store previous values of shared data.

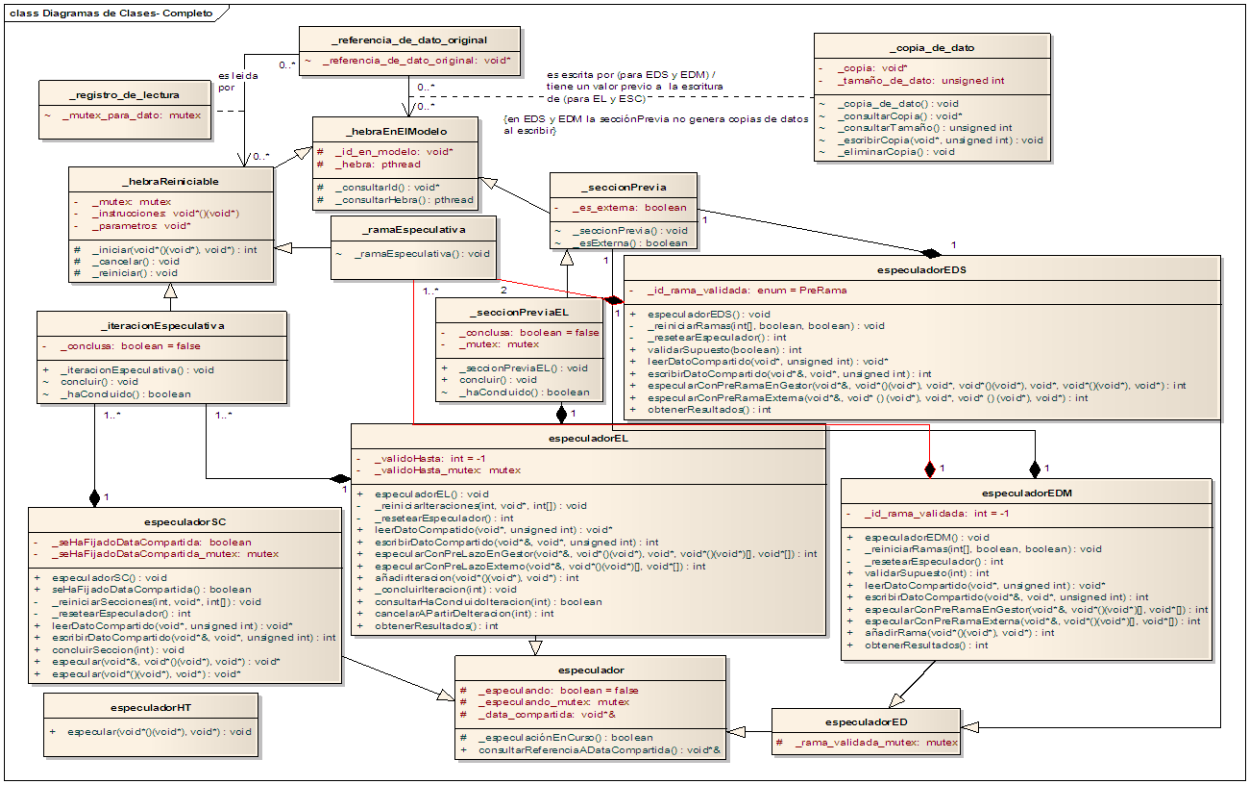

Figure 1: Class diagram of the library

4.3 Implementation

The current proposal is implemented using the pthreads library. The class design presented previously was articulated in a family of seven classes for objects speculators, five classes for parallel sections, and three additional classes to contain tracking data dependencies, all this included into two header files (one for the case of TLS and the other one for HT), to be added in the program that is going to use the library (see figure 2).

The support for dependency tracking mechanisms is managed by classes which map the original data and the sections that modify them. An extense explication about the way to use these classes is carried out in [14] using simply three classes: _dataCopy, _data_iterative_access_log, and _single_data_readers_log.

The design of the synchronization is a complex aspect of the library. The library defines a policy that divides its uses among private areas of each section and the space of coordination. We have defined a conservative scheme, which chose all mutexes following a particular order from the space of coordination to private. The resulting synchronization is perhaps the most susceptible to revision in future implementations.

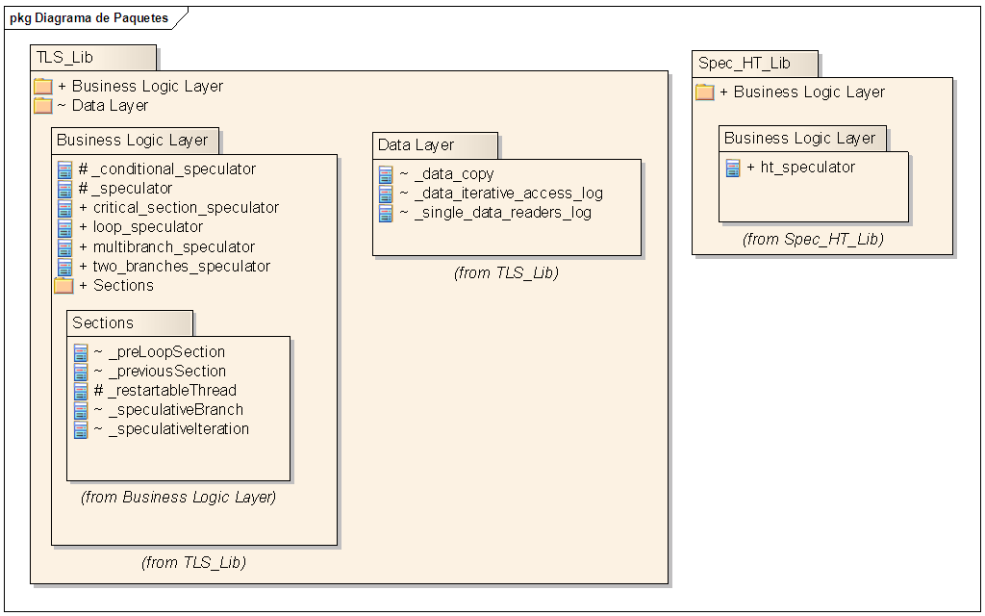

Figure 2: Package diagram of the library

4.4 Estimation of complexity

Table 1 presents an estimation of the complexity for each function implemented on the library.

The complexity in Table 1 of SSDS depends mainly on the number of data copied by the branches, being logarithmic with respect to reading and writing operations. As expected, the complexity of SMDS is almost identical to that of SSDS, but with a great dependence on the number of branches. The EL execution time is initially determined by the number of iterations. However, when the readings and writings increase, the execution time of EL depends on the number of readings and writings by iterations. The SCS has the same complexity as EL.

Table 1: Estimation of the complexity of the library

Function of the library	Complexity
n : # of threads • m:# of shared data • p:#of copied data by thread • q:#of written data by thread • r: # of previous values of a data for the iterations • s: # of critical sections of a data	
two_branches_speculator::_reset_branches	O(p)+O(q)+Cost to create and cancel a thread.
two_branches_speculator::get_shared_data	O(1)
two_branches_speculator::validate_supposition	Cost to reset a branch
two_branches_speculator::read_data	for pre-branch: O(1) for the branches: Olog(p)+Olog(m)
two_branches_speculator::write_data	for pre-branch: Olog(m) or Cost of reset_branches for the branches: Olog(p)
two_branches_speculator::speculate (Regardless of the time of the speculative execution itself)	O(p) or Cost of reset a branch
two_branches_speculator::speculate (case pre-branch)	O(1) or Cost of reset a branch
two_branches_speculator::get_results	O(p) or Cost of reset a branch
multibranch_speculator::_reset_branches	O(n)*(O(p)+O(q)+Costo de crear y cancelar hebras.)
multibranch_speculator::get_shared_data	Olog(n)
multibranch_speculator::validate_supposition	Olog(n)+Cost of reset a branch.
multibranch_speculator::read_data	for pre-branch: Olog(n) for the branches: Olog(n)+Olog(p)+Olog(m)

multibranch_speculator::write_data	for pre-branch: Olog(n)+Olog(m) or Cost of reset a branch for the-branches: Olog(n)+Olog(p)
multibranch_speculator::speculate (Regardless of the time of the speculative execution itself)	O(n)+O(p) or Cost of reset a branch
multibranch_speculator::speculate	O(n) or Cost of reset a branch
multibranch_speculator::get_results	O(p) or Cost of reset a branch
multibranch_speculator::append	Olog(n)
loop_speculator:_reset_sections	O(n)*{[Olog(q)*Olog(s)]+[Olog(p)*Olog(r)]+Cost to cancel and restart threads}
loop_speculator::get_shared_data	Olog(n)
loop_speculator::commit	Olog(n)
loop_speculator::read_data	[Olog(n)+Olog(m)+Olog(r)] or Cost to reset sections
loop_speculator::write_data	Olog(n)+Olog(m)+Olog(r)+Olog(s) or Cost to reset sections.
loop_speculator::append	Olog(n)
loop_speculator::valid_til	Olog(n)+Cost to reset sections
loop_speculator::speculate (Regardless of the time of the speculative execution itself)	O(n) or Cost to reset sections
loop_speculator::speculate	O(n) or Cost to reset sections
loop_speculator::get_results	O(n) or Cost to reset sections
critical_section_speculator::_reset_sections	O(n)*{[Olog(q)*Olog(s)]+[Olog(p)*Olog(r)]+Cost to cancel and restart threads}
critical_section_speculator::read_data	[Olog(n)+Olog(m)+Olog(r)] or Cost to reset sections.
critical_section_speculator::write_data	Olog(n)+Olog(m)+Olog(r)+Olog(s) or Cost to reset sections.
critical_section_speculator::speculate (Regardless of the time of the speculative execution itself)	O(n) or Cost to reset sections

Table 2 presents an estimate of utilization of memory of the speculative classes. These values represent an upper bound because in practice, as the branches are canceled or culminating iterations, the memory usage is dynamically reduced.

Table 2: Upper bound for memory usage by the speculative classes

Class	Upper bound memory usage
SSDS, SMDS	$\sum_{k=0}^{n}$ data used by the branch
EL, SCS	$\sum_{k=0}^{n}$ data used in each parallel section

Where n is the number of parallel sections.

5. Experiments

5 Definition of the experimental platform

Instead of using a simulation platform, we use a personal computer under realistic processing conditions. We use the ideal platform (SMT multicore). We have used a Dell computer with an Intel ® Pentium ® Dual Core 4, which offers one thread per core. This platform imposes a restriction of ~ 70 concurrent threads.

For each test program we study the coverage of its sections in an execution. The coverage analysis was performed using gcov, massif and Cachegrind tools [10, 14]. The first determines the number of times each line of code is

executed, but does not provide information on runtime. The profiles generated by Cachegrind determines the percentage of time of the program on certain functions, including the percentage of cache faults by function. The execution time was measured using the tools distributed with SPEC2000 benchmarks. The memory usage was measured with the massif tool.

Having identified these factors, we proceeded to design the speculative strategy in order to define the more large possible coverage. This strategy is implemented through the use of the library here presented, leading to a software version optimized by speculative parallelism. This version was evaluated with the tools already described, except for the average time, which was obtained by the command time-p, for a execution repeated the same number of times that for the base case.

We compare our approach with Prabhu work [10], which is distinguished by adopting an explicit view on the same level of this study. This view means that the programmer is responsible for both jobs: to choose exactly where in the code and how it is made the speculation. 4 tests were selected belonging to floating point benchmarks SPEC CPU 2000:

- *188.ammp* is a simulation of molecular physics. It models the behavior of a chemical complex composed by a protein and a protein inhibitor. The program simulates the interaction of the HIV virus with the inhibitor indinavir. This program has been used in several studies on the treatment of HIV. About 85% of the execution time of this program focuses on mm_fv_update_nonbon function. The Mm_fv_update_nonbon function is responsible for updating the values of the energy state of an array of atom, according to the action of non-linked forces (electrostatic and van der Waals forces). To accomplish this, consists of 9 non-nested sequential loops, each of which goes through all the array positions, but each loop accesses different variables, except the last one that depends on all previous. Prabhu studied the possibility of a speculative pipelining between loops [10]. In the present work another approach is used: the mm_fv_update_nonbon function was divided into nine functions to be invoked sequentially (loop1, loop2, ... loop9), each corresponding to one of the sequential loops. These loops are perfectly parallel, ie, that the iterations have not violations of dependencies. Consequently we use the LE speculator.
- *183.equake* corresponds to a simulation of seismic wave propagation using finite element methods. About 75% of the execution time of the program corresponds to the smvp routine, which is invoked 3855 times. This routine consists of an inner loop of 30169 iterations and an internal structure of type "repeat for" that runs about 8 times for each iteration of the loop. "repeat for" updates a single field of the matrix by iteration, but it is different in each case. Additionally, there is another internal routine, called smart, which has four critical sections to access zones of a matrix, and it is invoked 325 times. Prabhu propose a parallel assigning of the iterations, loop by loop [10]. In our platform this approach is not feasible, due to the concurrent number of threads required (30169). Our alternative was divide the inner loop into four speculative sections, and run each one a quarter of the 30169 iterations. The parallelization of this mutex was implemented through a mix among the SCS and EL speculators, called 3855 times, with 4 parallel critical sections in charge of executing the function smart of each smvp_for_spec branch.
- *179.art:* implements an adaptive resonance neural network (ART) for image recognition. ART is trained to recognize the image of a helicopter and an airplane in a thermal image. Approximately 96.6% of the runtime are divided among the match and train routines, which are invoked 500 times. The match routine has five simple decision structures into a loop of 10 iterations, and the other routine has a multiple decision structure. The Prabhu parallelization consists in of internal loops [10]. Our alternative mixes EL speculation with SSDS speculation for the first routine, and a SMDS speculation for the second one. A cancellation in the SSDS speculator implies all iterations are canceled for this decision.
- *177.mesa:* implements a 3D graphics library. For the test generates a 3D plot from 2D one. 90.32% of its running time is carried out by general_textured_triangle function, which invokes several small functions. The alternative proposed by Prahbu to optimize this code is to convert each call to this function (a total of 44.402 million) in a thread speculative [10]. We exploit another parallel alternative, we group these called in group without violations (in this way, we reduce the number of threads) and we use the EL speculator, using library functions that handle the tracking of the dependencies (to guarantee the consistence of each group).

Table 3 shows the execution time of the base case, for each test in our platform.

Table 3: Execution time for the tests: base case

Benchmarck	Ejecution time
188.ammp	347
183.equake	94.9
179.art	382
177.mesa	43.4

5.2 Results Analyses

Table 4 shows the results. Despite the limitations of the experimental platform, the parallelization results with the library were consistently positive in all cases to improve execution time, which proves the working hypothesis, ie by using speculative techniques at the level of developer is possible to obtain improvements in performance.

In our study, for 188.ammp we obtain an acceleration of 1.5x with respect to the original. A result similar to that obtained by Prabhu for a real memory system with 8 processing cores. For 183.equake, in our study obtains an execution on average 1.9 times faster than the original program. This value is lower than that obtained by Prabhu for a real memory system with 8 cores (~ 2.25x), but better with respect to results obtained in [15]. But, the percentage of memory used by smvp routine is reduced to 61.55% in our study. The running time for 179.art was 174 seconds, which represents an acceleration of ~ 2.2x with respect to the base case. In this new version, we also have improved the memory usage with respect to [15], but we obtain the same accelerations. The acceleration obtained for Prabhu 179.art is second ~ 2.25x values, similar to our results because we have improved our parallel version of 179.art algorithm. Finally, 177.mesa reaches an average of 25 seconds, corresponding to 1.8 times faster than the original program. The acceleration obtained is lower than that achieved by Prabhu, of ~ 2x, but the memory usage of our parallel version is very low.

Table 4: Results

benchmark	Speculation Type	Results	Prabhu results [10]
188.ammp	LE speculation	acceleration of 1.5 with respect to the original	acceleration of 1.5 with respect to the original
183.equake	mix among the SCS and EL speculation	1.9 times faster than the original program.	2.25 times faster than the original program
179.art	Two parts: the first one mixes EL with SSDS speculation, the second one is a SMDS speculation	An acceleration of ~ 2.2x with respect to the base case	An acceleration of ~ 2.25x with respect to the base case
177.mesa	EL speculation	1.8 times faster than the original program	2 times faster than the original program

In this occasion we are obtained significant improvement in the use of memory with our speculative models. Particularly, we observe that has a favorable effect on the parallelization in the case of 183.equake, but not in the case of 179.art, because we have improved the previous results obtained in [15]. Figures 3 and 4 show the memory usage for the parallelization carried out in [15] and the new version of parallelization carried out during this study.

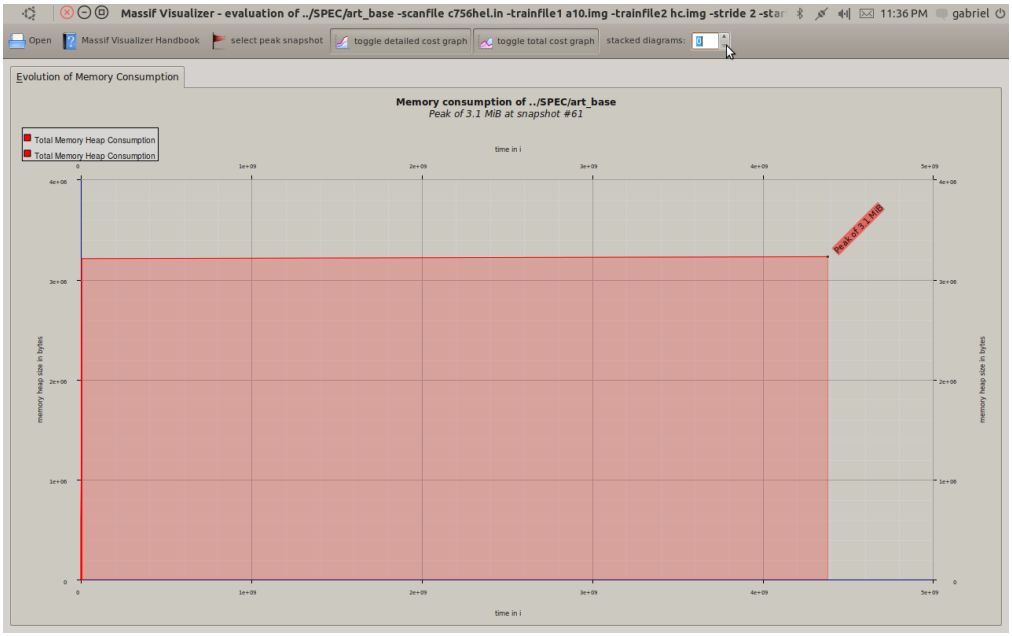

Figure 3: Memory usage for the new parallelization of 183.equake with our library

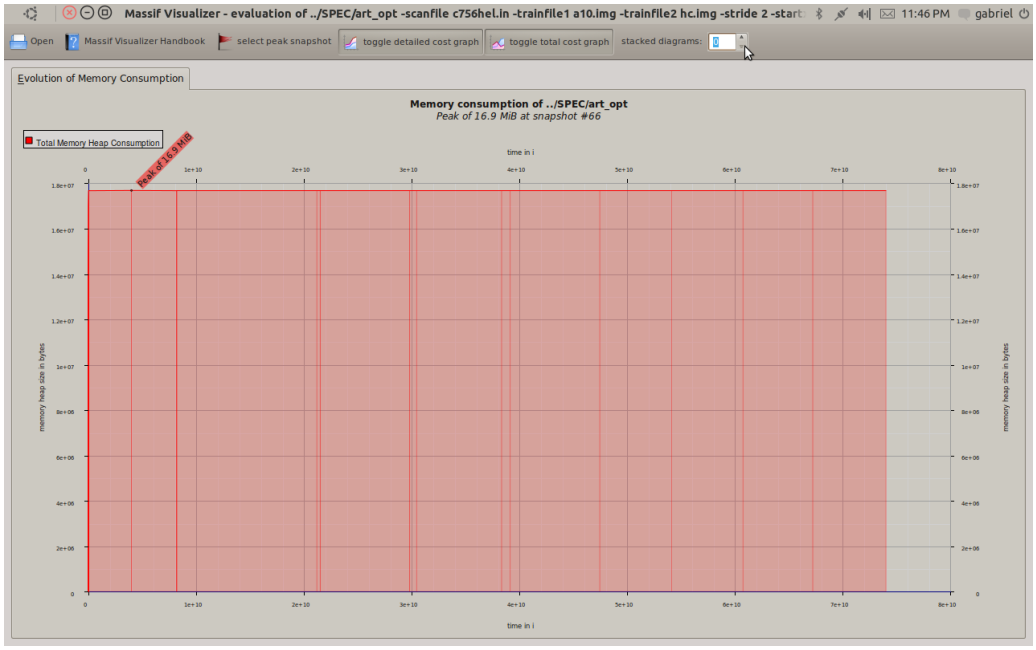

Figure 4: Memory usage for the parallelization of 183.equake carried out in [15]

We have improved the memory usage in our new version of parallel program for the case of 183.equake, due to number of copies generated (we have reduced the number of threads to manage the critical sections and the "repeat for" instructions of the internal loops of the smvp routine). In this way, we have reduced the size of the heap of 16.9MB to 3.1MB (this is shown in Figures 3 and 4 using massif tool). This smaller number of threads allows to free the memory to bring to it more data (it reduces the page faults), in order to improve significantly the accelerations obtained (we pass from an acceleration of 1.53x to an acceleration of 1.9x with respect to the original program).

The comparison of the results with Prabhu work's [10] is hard, because our work has used different speculative strategies to obtain a good performance with our library. Additionally, there is no a standard to define the best speculative strategy for each benchmark. This reveals that not only is necessary a programmable library as presented here, but also a common speculative strategy to compare results. The library described here could be used to study the different speculative implementations for a same program, and compare them.

6. Conclusions

This paper studies the problem of parallelism from the point of view of the speculation. This paper present the design and implementation of a library for explicit parallelism based on thread-level speculation. Additionally, the paper shows its utilization in several benchmarks, and presents the performance results. Particularly, we test different speculative model using our library.

The key aspects of this approach can be summarized as: a) we can offer speculative techniques like common programming functions b) This techniques must be used by the programmer c) the speculative functionality must be provided in a way that hides the complexity of support mechanisms.

The more complex aspects of the design were those related to dependency tracking mechanisms and synchronization. The design can be improved, for example using a pool of threads, using checkpointing, with a better planning of threads, among others.

The library was evaluated through a case study consisting of a sub-set of the SPEC CPU 2000. In the absence of a specific methodology for the explicit use of speculative parallelism, we chose to begin the study with an analysis of coverage, which allowed choosing the sections with more weight for optimization. As a second step, we carried out an analysis of the dependencies in these sections, to define the alternatives of decomposition/speculation. The criterion for choosing between these alternatives was the number of threads, and the criteria to reduce the number of violations of dependencies. In general, the library accelerates the execution of the programs. For the case studies in this paper, we were obtained accelerations in the order of 1.8x by using the library. Additionally, we have determined that the memory usage of a parallel version of an algorithm can influence in some case in the acceleration obtained. The empirical methodology used to address the problem of speculative parallelism exploitation, can be improved.

Basically, two aspects limit our results. First, we have not a large knowing of the program; normally a programmer knows where it should parallelize its code. An expert on the program can have a better exploitation of speculation parallelism. Secondly, the experimental platform limited the performance of tests, due to that we are not exploited fully all the benefits of speculative dependency tracking. We need extend the evaluation of the library to cover all forms of speculation offered, with variations in speculative dependencies.

Acknowledgement

To the CDCHT project I-1237-10-02-AA of the Universidad de Los Andes, and the FONACIT project PEI-2011001325, for their financial support.

References

[1] J Aguilar, E. Leiss, E. *Introducción a la Computación Paralela*. Consejo de Publicaciones, Universidad de Los Andes, 2004

[2] J. Aguilar, E. Leiss. “Data dependent loop scheduling based on genetic algorithms for distributed and shared memory systems”. *Journal of Parallel and Distributed Computing*, Elsevier, vol. 64, pp. 578-590, 2004.

[3] M. Cintra, N. Ioannou, J. Singer, S. Khan, P. Xekalakis, P. Yiapanis, A. Pocock, G. Brown, M. Lujan, I. Watson, “Toward a more accurate understanding of the limits of the TLS execution paradigm”, in *Proc. of the IEEE International Symposium on Workload Characterization*. Washington, DC, USA: 2004, pp. 1-12.

[4] M. Cintra, J. Dou, “A compiler cost model for speculative parallelization”. *ACM Transactions on Architecture and Code Optimization*. vol. 4, 2007.

[5] M. Hill, M. Marty, “Amdahl’s law in the multicore era”. *IEEE Computer*, vol. 41, pp. 33-38. 2008.

[6] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. Veidenbaum, C. Polychronopoulos, C. “Tight analysis of the performance potential of thread speculation using SPEC CPU 2006”, in Proc. *Symp. on Principles and Practice of Parallel Programming (PPoPP)*. 2007, pp. 215–225.

[7] J. Lu, A. Das,A. Wei-Chung, “Dynamic helper threaded prefetching on the Sun UltraSPARC® CMP processor”, in *Proc. of the 38th Annual ACM/IEEE International Symposium on Microarchitecture*: November New York, USA, 2005, pp. 12-17.

[8] C. Oancea, A. Mycroft, T. Harris, “A lightweight in-place implementation for software thread-level speculation”, in *Proc. of the 21st annual symposium on Parallelism in algorithms and architecture*, Calgary, Canada, 2009, pp. 223-232.

[9] V. Packirisamy, A. Zhai, W. Hsu, P. Yew, T. Ngai, “Exploring speculative parallelism in SPEC2006”, *in Proc. Intl. Symp. On Performance Analysis of Systems and Software (ISPASS)*. 2009: pp. 77–88.

[10] M. Prabhu, “Parallel programming using thread-level speculation”. Doctor of Philosophy (Computer Sciences), Stanford University, Palo Alto, USA, 2005.

[11] Standard Performance Evaluation Corporation. "CFP2000 (Floating Point Component of SPEC CPU2000)": http://www.spec.org/cpu2000/CFP2000/.

[12] C. Tian, M. Feng, V. Nagarajan, R. Gupta, "Copy or discard execution model for speculative parallelization on multicores", in *Proc. of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 330-341.

[13] T. Ungerer, B. Robic, J. Silc, "A survey of processors with explicit multithreading". *ACM Computing Surveys,* vol 35, pp. 29-63, 2003.

[14] J. Aguilar, K. Campero, "Librería para la especulación paralela en plataformas TLS", Centro de Estudios en Microelectrónica y Sistemas Distribuidos (CEMISID), Universidad de Los Andes, Tech. Rep. 35-2012, 2012

[15] J. Aguilar, K. Campero "A library for parallel thread-level speculation", in *Proc. of the XXXIX Conferencia Latinoamericana en Informática (CLEI 2013),* IEEE Xplore, vol. 2, Venezuela, 2013, pp. 66-76.