Concurrency and Computation: Practice and Experience, 2012, to appear

Dynamic Branch Speculation in a Speculative Parallelization Architecture for Computer Clusters

Joan Puiggali[1], Boleslaw K. Szymanski[2,3], Teo Jové[1], Jose L Marzo[1]
[1]Broadband Communications and Distributed Systems
Institut d'Informàtica i Aplicacions (IIiA)
University of Girona, 17071 Girona, SPAIN
[2]Społeczna Akademia Nauk, ul. Sienkiewicza 9, 90-113 Łódź, POLAND
[3]NeST Center, Rensselaer Polytechnic Institute, Troy, NY, 12190, USA
{joan.puiggali, teodor.jove, joseluis.marzo}@udg.es, szymansk@cs.rpi.edu

Abstract:
This article describes a technique for path unfolding for conditional branches in parallel programs executed on clusters. Unfolding paths following control structures makes it possible to break the control dependencies existing in the code and consequently to obtain a high degree of parallelism through the use of idle CPUs. The main challenge of this technique is to deal with sequences of control statements. When a control statement appears in a path after a branch, a new conditional block needs to be opened, creating a new code split before the previous one is resolved. Such subsequent code splits increase the cost of speculation management, resulting in reduced profits. Several decision techniques have been developed for improving code splitting and speculation efficiency in single machine architecture. The main contribution of this paper is to apply such techniques to a cluster of single processor systems and evaluate them in such an environment. Our results demonstrate that code splitting in conjunction with branch speculation and the use of statistical information improves the performance measured by the number of processes executed in a time unit. This improvement is particularly significant when the parallelized programs contain iterative structures in which conditions are repeatedly executed.

Keywords: *Speculation, Multipath execution, Branch speculation, Computer clusters, Control dependencies.*

1. INTRODUCTION

The main obstacle in improving performance of speculative multithreaded architectures is the limited degree of parallelization imposed by the intrinsic dependencies that exist among parallel threads. These dependencies can be:

- *structural*, which arise when instructions in different threads might attempt to use the same hardware resources at the same time;
- *data,* which arise when instructions in different threads refer to the same data item; or
- *control*, which arise when the flow of execution of some threads is dependent on a control statement (a loop, a branch, etc.).

Speculation techniques in computer architectures are used to overcome execution constraints imposed by control and data dependencies. Speculation is not always successful because incorrect prediction of speculative values will start threads that should not be executed; if this happens, most of the implementations erase all incorrectly started threads. Recent studies, such as [25], have not attempted to eliminate the threads created by the wrong predictions of speculative values, but rather aimed at applying new speculation control techniques to predict the usefulness of wrong-path episodes. This article shows that execution of some wrong-path instructions, even if erased, still can improve the performance because they might pre-fetch into caches data and instructions that are later needed by correct-path instructions.

Branch speculation refers to predicting which branch of a control statement will be taken without knowing all the values used in the corresponding control statement. Incorrect branch speculations have a major impact on execution performance. A conventional way to reduce the performance loss incurred by incorrect branch speculations is to increase the branch speculation accuracy [5], [30], [43], which, however, requires predictors of increased complexity.

The idea of eager (multipath) execution [3], [14], [20], [56] [57], has been introduced to limit the performance loss incurred by incorrect speculations about branches that are difficult to predict. To this end, a branch predictor produces branch speculation together with its confidence level [27], [41] in this speculation. For low-confidence speculations, many paths are unfolded and the instructions from the different paths are executed. Otherwise, only the path predicted to be taken with high confidence is started. When the branch outcome is eventually calculated, the executed paths are selectively flushed to remove wrong paths. A major problem of these techniques is the complexity of the resulting processor design and the complexity of implementation of the adequate predictor.

Experiments, such as the ones presented in [2], show that multipath execution can offer sizeable instructions per cycle (IPC) improvements over the traditional single-path execution models.

A major idea of this paper is to take techniques designed for the improvement of single machine architecture performance and apply them to a cluster of single processor systems. Hence, in this paper we describe the application of multipath execution in a system developed to execute sequential programs using parallel and speculative techniques for the computer cluster architecture that we developed [34], [35], [36]. Our experiments show that the proposed approach yields high performance even in environments with low branch predictability.

The term “unfolding” refers here to the parallel execution of two paths of a branch; that is, when a branch is reached during the program execution, it creates two new threads, one for each path.

This paper is organized as follows. In section 2 we present previous work. Section 3 describes the multipath (or eager) execution and the different methods of its application. Section 4 justifies the method used in our implementation. Section 5 presents the Speculative Parallelization Architecture for Computer Clusters which is a platform used in our implementation. In section 6, we describe how we applied a dual path execution to a synthetic program**.** Section 7 explains how we adapted the unfolding paths to repetitive structures. Finally, Section 8 summarizes the conclusions of the paper and describes future lines of research.

2. RELATED WORK

Most of the multipath implementations of speculation have been done at the hardware level. The greatest performance enhancement of multipath-enabled processors is improvement of the processor’s throughput; it allows multiple path execution in the same pipeline through processor resource sharing. It enables to control execution of each path through path identifiers that track each path execution within the processor architecture.

The different paths are executed independently of one another until the corresponding branch is resolved. Then, all the instructions executed on the incorrect paths created at the branch are discarded and the instructions from the correct path continue normal execution. In a conventional speculative processor, only one path is executed and the incorrect prediction of the speculating value delays the execution of the correct path until the corresponding branch is resolved. Thus, the multipath execution avoids the misspeculation penalty at the price of resource sharing (including sharing of physical register files, functional units, and the reorder buffer and reservation station entries) and control. With multipath execution, demands for the processor resources are increased because instructions from all paths after unresolved branches share the hardware structures in the microarchitecture. An example of such multipath processor architecture is presented in [56].

The emergence of the speculative multithreading models (see [1], [10], [22], [28], [31]) and simultaneous multithreading (SMT) processors (e.g., [48], [49], [50] [56]), for speculative thread execution allows the architecture-aware compiler to parallelize sequential applications without being constrained by the data and control dependencies present in the program.

An SMT processor is able to issue instructions from multiple threads in the same cycle, thus, allowing multiple hardware contexts, or threads, to dynamically share the resources of a superscalar architecture. In [56], the SMT available resources are applied using the so-called threaded multipath execution (TME) technique to achieve a high instruction level parallelism when there are only one or a few processes running. By following both possible paths of a conditional branch, in the best case, the misspeculation penalty can be completely eliminated if there is a significant progress down the correct path when the branch is resolved. TME uses unused (spare) contexts to execute threads in alternative paths of conditional branches. As a result, the SMT's resources can be better utilized, and the probability of executing the correct path is increased. TME can also provide significantly higher processor utilization than conventional superscalar processors.

There are also combined compiler and architecture techniques to control the multithreaded execution of branches and loop iterations [54]. These techniques can be applied by a compiler to replace branches using speculative execution of both branch paths and speculative execution of loop iterations. The resulting code needs to be tuned to a specific architecture. In [55] a compiler technique called simultaneous speculation scheduling is proposed in combination with a 'minimal' multithreaded execution model to enable speculative execution of alternative program paths.

Some hardware implementations apply the different forms of eager execution. The 'nanothreaded' DanSoft processor [13] implements a multipath execution model using confidence information from a static branch speculation mechanism. The PolyPath architecture [20] enhances superscalar processor architecture through a limited multipath execution feature that employs eager execution. In [55], eager execution is used in a simultaneous multithreaded processor model. Finally, [51] proposes Disjoint Eager Execution, which assigns resources to branch paths whose results are most likely to be used, that is, branches with the highest cumulative execution probability.

3. SPECULATION VERSUS MULTIPATH EXECUTION OF BRANCH INSTRUCTIONS

How processors handle conditional statements is very important for their performance considering that one of every seven executed instructions is a branch. Low-level techniques are used to optimize the time needed to evaluate the condition and the address of the jump, but this is not quite enough. Waiting for the evaluation of a condition blocks the system and reduces its performance. Decreasing this cost can be accomplished with one of two methods: condition speculation and path unfolding.

The use of a speculative method to evaluate a condition often results in having to manage new conditions before the previous ones are resolved. The system must also be able to remove the path for which the branch has not been correctly predicted. However, it should also be taken into account that in most cases, it is highly probable that an instruction is executed more than once. Moreover, most of the methods used in predicting jump conditions have a very high rate of success. In studies like [16], [29], the application of speculation methods, regardless of whether implemented in hardware or software, have shown an accuracy of over 85%. These results motivated us to introduce speculative execution that relies on speculation about what will happen when a conditional statement is executed. This includes the case when the condition is guarding a loop body.

Speculations can be divided into two groups: static and dynamic. In the static case, the branch speculation is fixed, meaning it is always skipped or it does not depend on the information that the compiler puts into the code. The disadvantage is that static speculation is not adapting to the instruction behavior. In the dynamic speculation, the decision depends on the instruction’s behavior during execution and therefore requires the historical evolution information of the address of the instruction to which the jump is directed. The BTB (Branch Target Buffer) method is an example of dynamic speculation in which the jump instruction target address and historical information (the latter is typically limited to just 2 bits) are stored in a buffer [24].

Application	***Committed instructions (in millions)***	***Conditional branches (in millions)***	***Branches taken (%)***	***Misspeculation rate (%)*** ***SAg***	***gshare***	***combined***
compress	80.4	14.4	54.6	10.1	10.1	9.9
gcc	250.9	50.4	49.0	12.8	23.9	12.2
perl	228.2	43.8	52.6	9.2	25.9	11.4
go	548.1	80.3	54.5	25.6	34.4	24.1
m88ksim	416.5	89.8	71.7	4.7	8.6	4.7
xlisp	183.3	41.8	39.5	10.3	10.2	6.8
vortex	180.9	29.1	50.1	2.0	8.3	1.7
jpeg	252.0	20.0	70.0	10.3	2.5	10.4
mean	267.6	46.2	54.3	8.6	14.5	8.1

Table 1: *Comparison of different methods of speculation using the SPECint95 benchmark [11]; the labels in misspeculation rate column denote: SAg - a variant of the second-level branch history method; gshare - a two-level adaptive predictor with globally shared history buffer and pattern history table; combining - two-bit predictor with the gshare predictor*

Different variants of this method were introduced [19], each proposing different ways of making the speculation decision (one-bit predictor, two-bit predictor, etc.). The more bits a predictor uses, the higher the cost of necessary hardware is. Hennessy and Patterson [15] conducted a study using two bits of history. They showed that for programs in SPEC89 the speculation errors ranged from 1% (nasa7, tomcat), to 9%(spice), to 12% (gcc), and even to 18% (eqntott) when the BTB table had up to 4,096 entries. Other methods rely on correlation-based predictors [32] that speculate about a branch outcome taking into account the behavior of other branches. These methods are motivated by the observation that the outcome of a branch is often affected by the outcomes of recently executed branches. Other types of predictors, like two-level adaptive predictors [59] and hybrid predictors [29], also work with information collected from other jumps.

One of the most important factors in deciding how to use a speculation is the speculation's confidence level [27], [41]. Considering that only a fixed percentage of accuracy can be achieved for all conditions, the rate of success in opening new paths at level *n* can be expressed as:

$$\text{Probability of success} = \prod_{\text{Level}=1}^{n} \text{Percentage of success}_{\text{Level}}$$

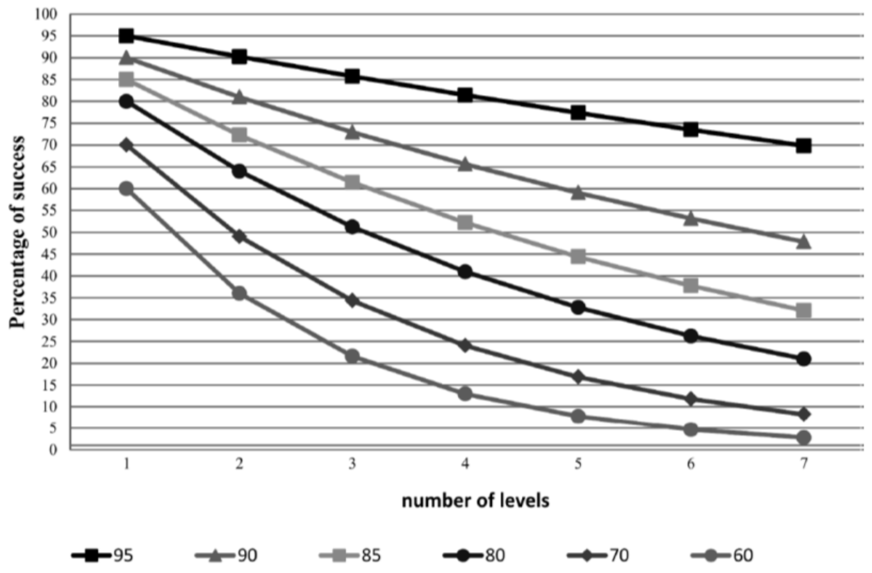


Figure 1: *Speculation success rate at different levels of condition nesting*

Number of levels	*Single condition success rate*					
	95%	*90%*	*85%*	*80%*	*70%*	*60%*
1	95.00	90.00	85.00	80.00	70.00	60.00
2	90.25	81.00	72.25	64.00	49.00	36.00
3	85.74	72.90	61.41	51.20	34.30	21.60
4	81.45	65.61	52.20	40.96	24.01	12.96
5	77.38	59.05	44.37	32.77	16.81	7.78
6	73.51	53.14	37.71	26.21	11.76	4.67
7	69.83	47.83	32.06	20.97	8.24	2.80
8	66.34	43.05	27.25	16.78	5.76	1.68

Table 2: *Speculation success rate at different levels of condition nesting*

The confidence levels obtained in different studies [27], [41] show that the confidence goes down considerably when speculations are made at increasing number of levels, as shown in Figure 1 and Table 2. It is also worth mentioning that different branches may have different rate of speculation success.

The *unfolding paths* or *eager* execution [3], [14], [20], [56] proceeds down both paths of a branch so no speculation is made. When a branch is resolved, all operations on the non-taken path are discarded. This method allows the system to take advantage of parallel hardware architecture. If there are idle processors, two paths of a branch can be executed without waiting for the result of the control condition. However, as we demonstrate later, parallelizing splits in the condition structures by unfolding paths is not always beneficial. Each additional split must duplicate the split condition data structure, so it increases the cost of management control but decreases the gain of splitting. For example, consider an extreme case with a scheme in which all nodes are branches. If all paths are open, as shown in Figure 2, the number of processors needed to run all levels would be:

$$\text{number of processors} = 2^{\text{actual level}+1} - 1$$

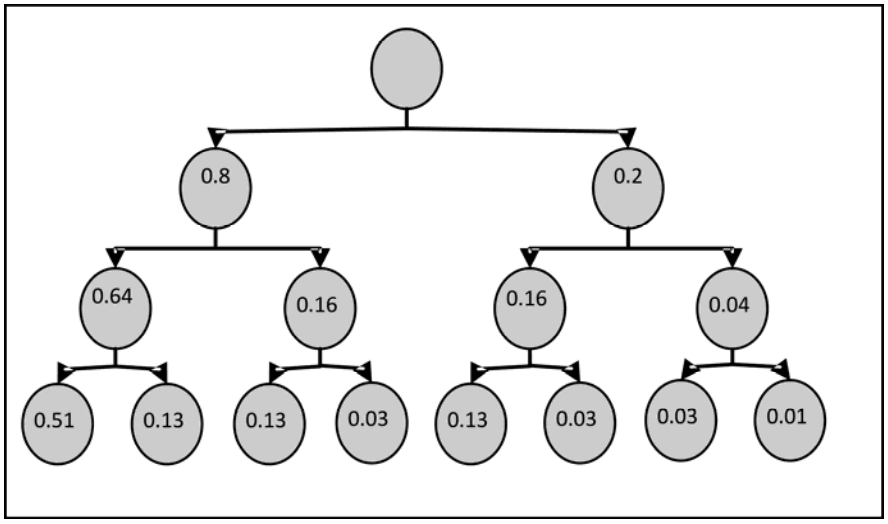


Figure 2: *Rate of speculation success at the different levels of accuracy for a fixed condition speculation success rate*

If only the first branch splits, as shown in Figure 3, and the other branches use speculation about which path to take to execute just one path, the number of needed processors can be expressed as:

$$\text{number of processors} = 2 * \text{actual level} - 1$$

This is because one-level unfolding executes the branch and splits the first level. Afterwards, a processor of each path executes it in parallel with other paths.

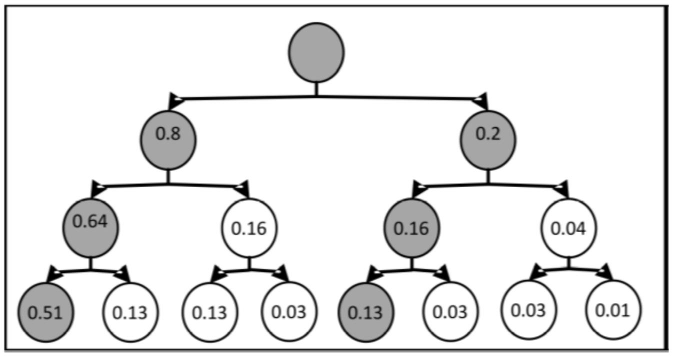


Figure 3*: One-level unfolding*

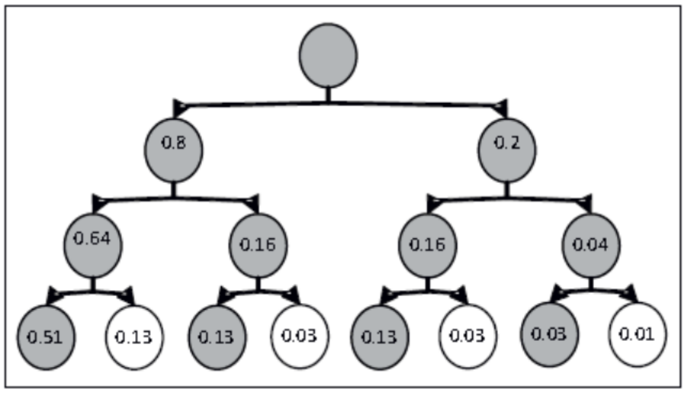


Figure 4*: Two-level unfolding*

If we generalize this formula, assuming that we split *n* levels (see Figure 4) and speculate about the paths taken at the remaining levels, the number of needed processors would be:

If $level_{unfolding} \geq level_{actual}$:

$$2^{\mathrm{level}_{\mathrm{actual}}+1}-1$$

Else:

$$\left(level_{actual}-level_{unfolding}+2\right)2^{level_{unfolding}}-1$$

The number of needed processors shown in Figure 5 and Table 3, demonstrates that from the third level on, the number of processors required to split all possible branches is large (15 or more). However, if only the first path is split and the other branches use speculation to execute one path, the number of needed processors is small (at most 7).

Consequently, the majority of techniques relay on mixed methods. Many of those use confidence estimation [42] to control speculation about the branch. For example, after a branch, if the confidence level in the speculation is low, both of its paths will be executed, otherwise, the speculation is used to execute the path predicted to be taken [52], [53].

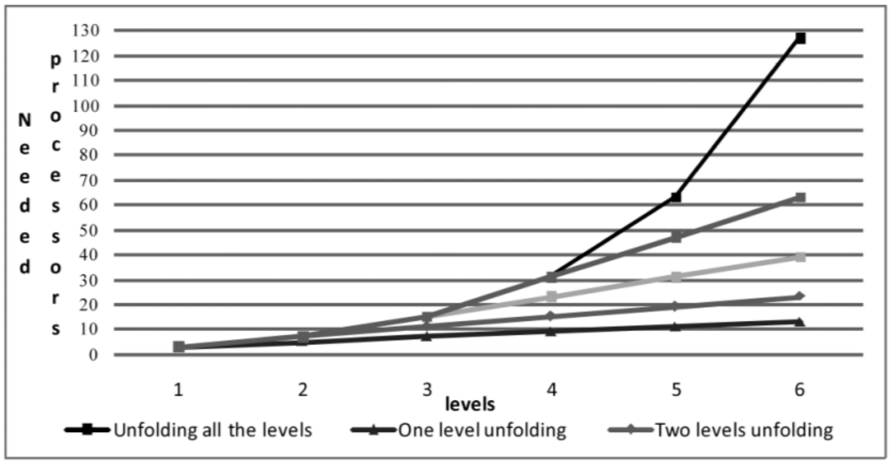


Figure 5: *Comparison of a number of processors required to achieve a certain level of unfolding*

Levels	1	2	3	4	5	6
Unfolding at all the levels	3	7	15	31	63	127
One-level unfolding	3	5	7	9	11	13
Two-level unfolding	3	7	11	15	19	23
Three-level unfolding	3	7	15	23	31	39
Four-level unfolding	3	7	15	31	47	63

Table 3: *Comparison of a number of processors required to achieve a certain level of unfolding*

4. A MIXED METHOD WITH CONFIDENCE ESTIMATION AND SINGLE UNFOLDING

We have developed an execution environment that allows two execution threads to be unfolded when a branch is found. A replication of the control structures is required to schedule the two branches as shown in Figure 6. This replication is done automatically when a branch is reached and later, when it is solved, these structures are merged back.

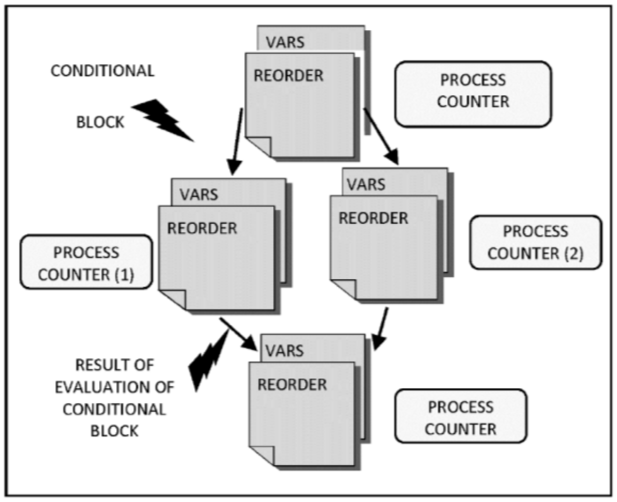


Figure 6: *Replication of the control structures*

The unfolding of threads of branch execution allows us to take advantage of the parallel architecture of the execution system. Processors that would otherwise be idle execute processes of the two paths of a branch (without waiting for the value of the condition). As discussed above, the generalization of this approach to nested conditional processes may not always be beneficial. Hence, we have chosen to study the optimal number of unfolding paths open simultaneously in the Speculative Parallelization Architecture for Computer Clusters.

Let's consider a parallel architecture with 11 processors executing 11 processes with the following control diagram (without data dependencies) in which the loops at the bottom are executed three times:

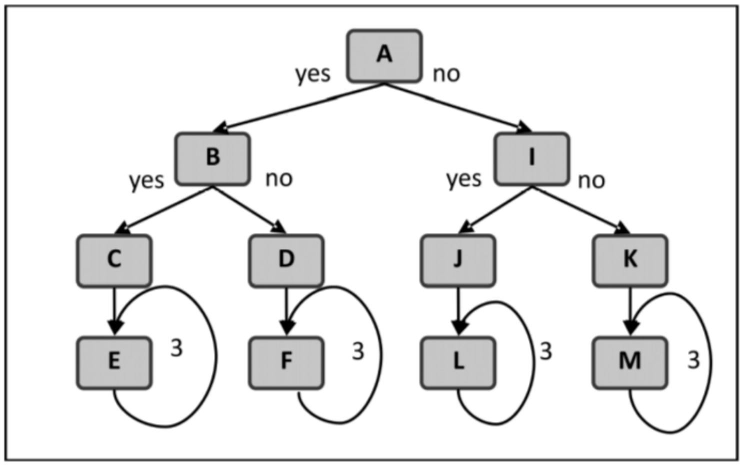


Figure 7: *Control diagram without data dependencies*

Assuming that the correct paths are the ones that follow "yes" branches of conditions A and B, when all three branches are unfolded, the resulting execution would be:

(1) *{A,B,C,D,I,J,K,E,F,L,M}*
{E,E}

On the other hand, the sequential execution of branches would require three steps:

(2) *{A}*
{B}
{C,E,E,E}

If only the first condition is unfolded, and outcomes of the second and third conditions are speculated about, the resulting execution under the same assumptions about conditionals would be:

(3) *{A,B,I,C,J,E,L,E,L,E,L}*

Since the sequential execution of branches would again be (2), the gain is higher in (3) than in (1) if limited number of processors are available.

In the above case, the high gain in (3) is achieved because the speculation predicted the conditions correctly. If the speculation predictions of the conditions are incorrect, for example, when the condition B is predicted to follow "no" path, the result of unfolding of the first condition and speculating about outcomes of the second and third conditions is:

(4) *{A,B,I,D,J,F,L,F,L,F,L}* *{C,E,E,E}*

Comparing (3) to (4), a reduction of single unfolding performance is observed. Taking into account this loss and the gain obtained by successful speculation, and adding the extra cost imposed by opening all branches, it is clear that in this case, it is more efficient not to open more than one branch simultaneously. Clearly, when several processors are available, all processes belonging to the same depth will be executed in parallel, without increasing the depth (with 19 processors available unfolding of all levels will result in single step execution). Yet, while many branches are executed, only one of them is correct. Hence, the more branches are opened, the smaller the gain and the higher the management cost.

If there are as many processors as potential processes, for nested branches without loops a binary execution tree will emerge. An example of such a tree with branches nested to level 3 is shown in Figure 8.

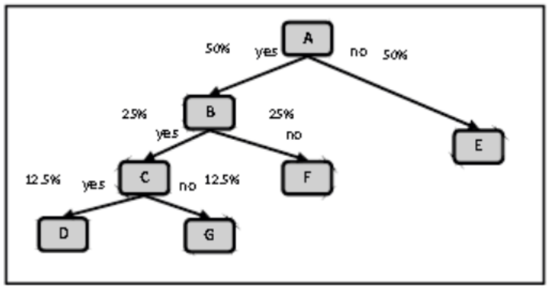


Figure 8: *The probabilities of taking each path in the execution tree*

The percentages in the execution tree define the probabilities of following each path. It is assumed that unlimited number of processors can be executed in parallel. The unfolding of one versus three branches is compared using the following notation:

Cpi_{sc} = the execution time obtained with no path unfolding but with speculation on the longest path. The best case is if we speculate about executing paths *A,B,C,D* that can be executed in parallel on four processors in one step, and this is the end of execution for 1/8 of all cases. In the remaining 7/8 of all cases, the correct "right" path is executed (one additional step) since all the conditions are known and all the partial results have been obtained.

$$Cpi_{sc} = 0.125 * 1\,cycle + 0.875 * 2\,cycle$$
$$Cpi_{sc} = 1.875 * cycle$$

Cpi_{e1} = the execution time obtained with single unfolding and speculation on the longest path. In one step, paths *A,B* are executed by unfolding while paths *C,D*, are executed by speculation. This gives the final results in half of the cases when branch 1 resolves to “no”. In additional 1/8 of all cases, the result is also obtained in one step when all three branches resolve to “yes”. However, in the remaining 3/8 of all cases, it is only necessary to execute either path *F* or path *G*, and we know which one because the results of conditions *B* and *C* are known at this point, so the result is computed in two steps.

$$Cpi_{e1} = 0.625 * 1\, cycle + 0.375 * 2\, cycle + Cges_1$$
$$Cpi_{e1} = 1.375 * cycle + Cges_1$$

Cpi_{e3} = the execution time obtained with triple unfolding. Here, all seven processes are executed in 1 step and the right combination is selected to get the result because values of all branches are known at that point of execution.

$$Cpi_{e3} = 1 * 1\, cycle + Cges_3$$
$$Cpi_{e3} = 1 * cycle + Cges_3$$

$Cges_3$ = the time of management with triple unfolding

$Cges_1$ = the time of management with single unfolding (which is smaller than $Cges_3$)

The unit of execution time is the time to execute a process. Hence, the larger the processes are, the more beneficial the unfolding is. Clearly, when $Cges_1 < 0.5$, single unfolding is more beneficial than pure speculation. Additionally, it could be argued that $Cges_3 \geq 3*Cges_1$ because triple unfolding needs to duplicate at least three times more data than single unfolding. Under this assumption, triple unfolding is better than single unfolding (and, of course, in this case it is also better than speculative execution) when $Cges_1 < 0.1875$.

This result confirms that in a scheme without loop structures, single unfolding is not always better than speculation nor is triple or multiple unfolding always worse than single unfolding. The ratio between the time of management of unfolding and the execution time of the processes determines if single unfolding is better than the other methods.

This analysis is even more precise when more is known about the conditions of branches. For example, if “no” paths have probabilities of only 10% and “yes” path 90%, then we get:

$Cpi_{sc} = 0.729*1+0.271*2=1.271$
$Cpi_{e1} = 0.829*1+0.171*2+Cges_1=1.171+Cges_1$
$Cpi_{e3} = 1+Cges_3$

In this case, the single unfolding is best only if $0.0855< Cges_1<0.1$, so in a narrow interval, whereas triple unfolding is better for a much wider interval of $Cges_1<0.0855$.

These results confirm that when more levels unfold, the time needed for managing unfolding must be low to achieve a gain over the other methods.

Finally, Figure 9 compares the unfolding performance of all levels versus unfolding performance of one level as a function of the number of processors used. As can be seen in this figure, when all levels are being unfolded, increasingly more processors are needed to descend to the subsequent level because it is necessary to run all processes of that level.

$$\text{number of processors} = 2^{\text{level}+1} - 1$$

For example, seven processors are needed to unfold the second level.

If only the first level splits and the others are speculated about, the number of processors needed to execute is smaller.

$$\text{number of processors} = 2 * \text{level} - 1$$

In this case, obtaining the second level results requires five processors, as shown in Table 4.

Level	1	2	3	4	5	6	7	8	9	10
Unfolding at all levels	3	7	15	61	33	127	255	511	1023	2047
Unfolding at one level	3	5	7	9	11	13	15	17	19	21

Table 4: *Levels of unfolding versus the number of processors used in execution*

Comparison of the two models run with 15 processors shows that unfolding all levels stops at third level when all available processes are used. In contrast, with single level unfolding, there are enough processors to execute up to 7 levels in parallel. However, the use of speculation in single unfolding limits the guarantee of execution correctness to only the first level path. So, in the worst case (where all speculations happen to be wrong) in single level splitting, two levels are lost compared to unfolding all levels. Conversely, if all the speculations are correct, the single unfolding gains four levels over unfolding of all levels. Considering that speculation success rates up to the third level are quite high, it is very likely that single unfolding will not lose any levels to misspeculation.

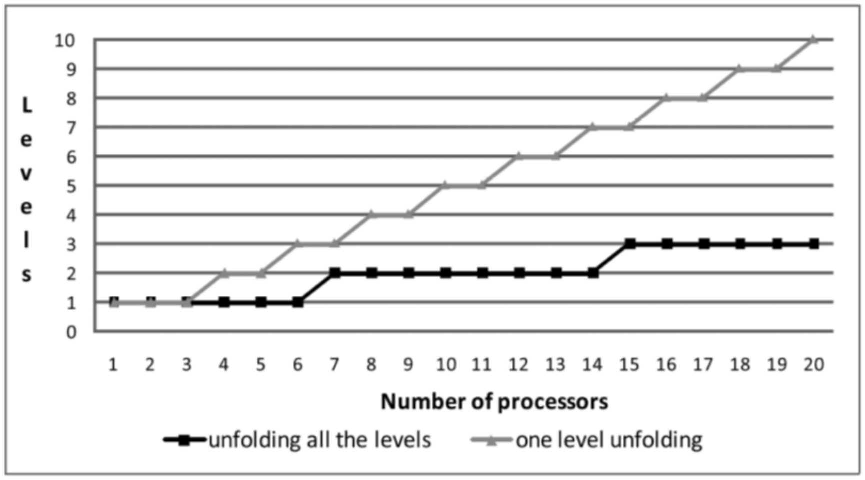


Figure 9: *Levels of unfolding versus the number of processors used in execution*

For these reasons we choose to use a mixed method approach that does not allow more than one level of splitting paths.

In this paper, we compare four methods (two of them unfolding branches and the remaining two speculating on branches without unfolding them) to see which gives the best results. We also evaluate the method without splitting to measure the benefits obtained by the investigated four methods.

5. SPECULATIVE PARALLELIZATION ARCHITECTURE FOR COMPUTER CLUSTERS

Speculative Parallelization Architecture for Computer Clusters [34], [35], [36], [37], [46], [47] achieves parallelism by using speculation in distributed environments, allowing the parallel execution of a sequential program in a computer cluster. It simulates the behavior of a superscalar system by implementing instruction level parallelism that attempts to break true data and control dependencies by speculating on future data values and future branch results, respectively. Speculation is based on the fact that the program behavior is usually repetitive and consequently predictable, as demonstrated in studies of branches [43], memory dependencies, and data values [4]. Software speculation has recently shown promising results in parallelizing such programs [8], [18], [45]. The relevant techniques can be classified into two types:

- Software speculation: Compilers carry out the necessary coding. The resulting speculation cannot be applied dynamically [44], [23], [61].
- Hardware speculation: It requires duplicated hardware elements, e.g., adding extra registers to store provisional values until they are resolved [4], [11], [17].

The above techniques allow the processor to divide program execution into several parallel threads, and therefore increase the program's degree of parallelism. Moore's Law (processing power doubles each 18 months) and Gilder's Law (bandwidth triples each 12 months) show that the speed of information transmission and synchronization between workstations decrease faster than processing speed increases. These premises make the idea of transporting speculation techniques to a distributed environment composed of cheap workstations attractive. The complete design of the Speculative Parallelization Architecture for Computer Clusters system [34], [35], [36], [37] consists of three subsystems:

- The parallelizing subsystem (see Figure 10) transforms the original sequential program into the parallel format needed by the execution environment. The program is divided into blocks that can be executed in parallel. Either two or three programs (depending on the type of the original program) are generated as a result of the translation process: a farmer, a worker, and optionally a farmer/worker. A prototype implementation of this subsystem automatically transforms C code into MSSPACC format C code by splitting loops and conditions in the corresponding blocks with their input and output variables. The description of its implementation is omitted here for the sake of brevity[1].When dividing a sequential program into blocks, it is very important to choose the correct block size, since it can affect system performance significantly. We are currently working on enhancing this aspect of the parallel subsystem following three options for optimizing block size: (i) user annotations of the block boundaries (the easiest but the least automatic choice), (ii) statistical information collected

[1] For details see University of Girona Technical Report IIiA 12-02-RR titled "The parallelizing subsystem implementation," by J. Puiggalí, T. Jové, J. Marzo, and J. Suy.

prior to parallelization, and (iii) a dynamic subsystem that can join blocks to improve system performance.

The farmer manages the parallelism and the speculation of the system. The worker runs at each of the processors; it contains the code of one of the blocks into which the sequential program has been divided. The farmer/worker program can reduce the farmer bottleneck by distributing the tasks to some of the other processors, each of which works then as a sub-farmer.

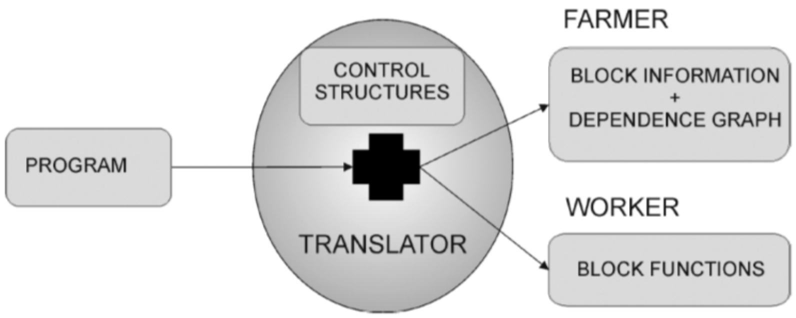


Figure 10: *The parallelizing subsystem*

- The execution subsystem (see Figure 11) applies speculation to run the parallelized applications in a computer cluster composed of single processor machines running PVM (Parallel Virtual Machine). The execution environment behaves like a superscalar processor, where the blocks are like the instructions into which the sequential program has been divided, and the processors on which the worker program runs are like the functional units. The following data speculation mechanisms are used: data value speculation [26]; last value predictor [26]; stride predictor [39]; and context-based value predictor [40]. Control dependencies are managed with branch speculation techniques based on a BTB (Branch Target Buffer) with 2-bit history [60]. Blocks executed because of incorrectly predicted values or wrong branch speculations are discarded and their execution is restarted from the last stable point.

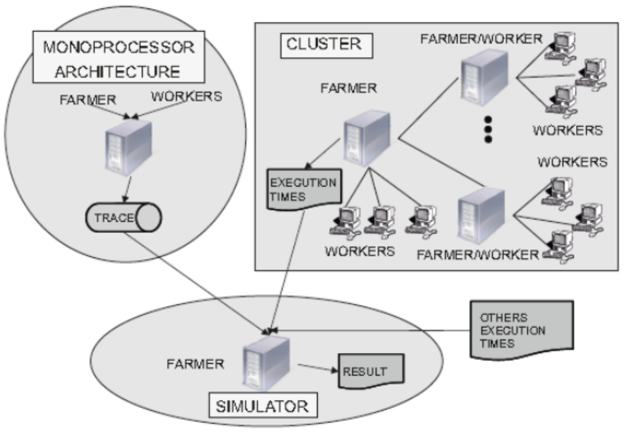


Figure 11: *Execution subsystem and simulation subsystem*

- The simulation subsystem (see Figure 11) evaluates the impact of technological evolution or the effects of using computer clusters larger than currently available. The simulation can run on a single workstation, using the information obtained from the single processor execution (the trace of the program) and the cluster execution model (the execution cost of different blocks).

The study and development of both subsystems has been initiated simultaneously. The parallelization subsystem is currently being designed. The execution subsystem has been already developed in C on PVM. It runs in computer clusters of up to 20 (PC) units. The design of the execution subsystem is based on both theoretical analysis and a new simulation subsystem that has been extensively used [34][35][36]. This allows the extrapolation of the results to the PVM subsystem configurations of ideal clusters, i.e. those that are not actually available. The simulation uses the runtime, transmission and control values obtained from the actual executions in the cluster [34],[35]. The sequential execution times have been obtained from the execution subsystem and from the simulator output. To analyze and validate the performance, synthetic programs have been used. However, pending access to the actual parallelization subsystem, two real applications have been manually adapted (the travelling salesman problem [34] and a program to generate virtual scenes illuminated by radiosity [47]).

In our recent work [37], the execution subsystem has been enhanced allowing out-of-order executions (OoOE) [33], [58]. The introduction of OoOE in the processor design implies that the execution of instructions can start any time and the final result will not be affected even when there is a blockage caused by data dependencies. This takes advantage of instruction cycles that would otherwise be wasted, and so yields an improvement in system performance. In current computer architectures, OoOE is a paradigm already used in many microprocessors.

6. DUAL PATH EXECUTION OF A SYNTHETIC PROGRAM

In this section, we describe how a synthetic program is used to measure the efficiency of unfolding two paths. The program has a loop before the branch and two loops inside each branch (see Figure 12). We use a simulation tool [35] that takes into account the overhead of each technique to obtain the results.

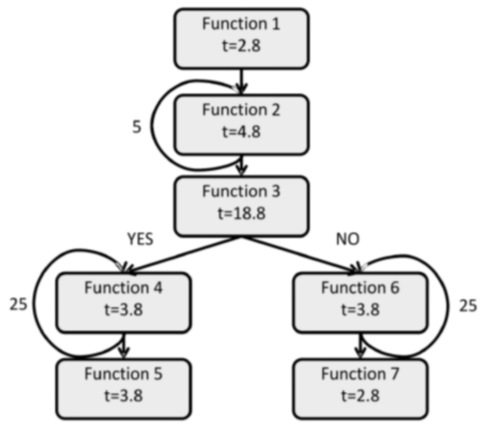


Figure 12: Algorithm 1

During the simulation, the result of the first branch was delayed to allow the other branches to be executed before knowing their results. The first version of the synthetic program has no data dependencies, while the second version has exactly two such dependencies (function 2 and function 4). In both cases, the dependencies can be addressed through speculation in two iterations. The control dependency created by the loop is solved by speculation, so there is no delay of the execution due to such dependencies. The resulting algorithm is shown in Figure 12.

The first part of the experiment was carried out on both synthetic program versions assuming that the condition is true and using two approaches—one with unfolding paths and the other without. In the second case, the condition is solved through speculation (the speculation predicts that the condition is true). The execution times obtained are shown in Table 5.

The number of cpu's	**Without dependencies**				**With dependencies**			
	With unfolding paths	**Without unfolding paths**	**Δ**	**%**	**With unfolding paths**	**Without unfolding paths**	**Δ**	**%**
1	170.11	170.11	0.00	0.00	170.11	170.11	0.00	0.00
2	90.42	86.85	-3.57	-4.11	98.19	98.19	0.00	0.00
3	64.50	61.23	-3.27	-5.34	80.46	80.46	0.00	0.00
4	54.84	50.05	-4.79	-9.57	68.20	68.20	0.00	0.00
5	47.92	44.68	-3.24	-7.25	65.28	65.28	0.00	0.00
6	43.20	40.61	-2.59	-6.38	62.22	62.21	-0.01	-0.02
7	40.13	38.34	-1.79	-4.67	59.47	59.41	-0.06	-0.10
8	39.56	38.19	-1.37	-3.59	58.30	58.19	-0.11	-0.19
9	38.16	34.37	-3.79	-11.03	58.33	58.17	-0.16	-0.28
10	36.77	34.22	-2.55	-7.45	57.58	57.37	-0.21	-0.37
11	36.64	34.07	-2.57	-7.54	57.03	57.02	-0.01	-0.02
12	35.89	34.07	-1.82	-5.34	56.55	56.50	-0.05	-0.09
13	35.37	34.07	-1.30	-3.82	56.37	56.37	0.00	0.00
14	35.19	34.07	-1.12	-3.29	56.37	56.37	0.00	0.00
15	34.42	34.07	-0.35	-1.03	56.37	56.37	0.00	0.00
16	34.32	34.07	-0.25	-0.73	56.37	56.37	0.00	0.00
17	34.27	34.07	-0.20	-0.59	56.46	56.37	-0.09	-0.16
18	34.22	34.07	-0.15	-0.44	56.64	56.37	-0.27	-0.48
19	34.17	34.07	-0.10	-0.29	55.15	55.10	-0.05	-0.09
20	34.17	34.07	-0.10	-0.29	55.05	54.90	-0.15	-0.27
21	34.14	34.07	-0.07	-0.21	55.83	54.88	-0.95	-1.73
22	34.14	34.07	-0.07	-0.21	55.33	54.88	-0.45	-0.82
23	34.12	34.07	-0.05	-0.15	56.68	54.88	-1.80	-3.28
24	34.07	34.07	0.00	0.00	56.48	54.88	-1.60	-2.92
…	…	…	…	…	…	…	…	…
35	34.12	34.07	-0.05	-0.15	58.96	54.88	-4.08	-7.43

Table 5: *Execution times of a synthetic program when the speculation correctly predicts the condition; Δ columns contain the absolute difference while % columns contain the percentage of the difference between execution times with and without unfolding paths*

Table 5 demonstrates that the synthetic program without data dependencies executed with a small number of processors performs better with the speculation without unfolding the branch. Yet, as the number of processors increases, the difference decreases until the execution times are equal. This is because in speculation without unfolding, if the path chosen is the correct one, all executing processes contribute towards the progress of the computation. In contrast, with unfolding, both paths are opened after the branch, so some processes will be assigned to the path that have been started but do not need to be executed. This is reflected in the results shown in Table 5. The difference in the execution times of these two methods decreases when the number of available processors increases. This is because the unused processors can execute the processes corresponding to the erroneous branch without delaying the execution of the correct path. With 24 processors available, the execution times are identical in both methods but the number of processes started without unfolding is 33, while with unfolding it is 58.

In the synthetic program with data dependencies, the method that speculates on branches, even in the case when the speculation prediction is correct, the executed path of the branch still has a dependency that requires a second iteration to speculate on the data value. On the other hand, the method that unfolds new execution paths for branches assigns the paths for which the speculation incorrectly predicted data to idle processors. Thus, as shown in Table 5, there is almost no difference in performance of the method speculating on branches and the method unfolding new execution paths on branches.

The same experiment was carried out assuming the contrary outcome of the condition—that is that the condition is false—and the results obtained are shown in Table 6.

In this case, unfolding the paths gives good results regardless if there are data dependencies in the synthetic program or not. The results for the synthetic program without data dependencies are better than the results for the synthetic program with data dependencies. This is because in the latter, the incorrect path has data dependencies and the process executing this path is blocked until the dependency is resolved.

The improvement achieved by unfolding paths versus speculating on branches reaches 17.45% using 14 processors for the synthetic program without data dependencies and 15.52% for the synthetic program with data dependencies. This is because in speculation without unfolding, the system starts the execution of processes in the incorrect path and proceeds until the value of the condition is obtained. Once the misspeculation is detected, the processes on the incorrect path are erased. However, in speculation with unfolding, the system starts process execution of both paths and later keeps the one that was started with the correct value of the condition.

The number of cpu's	**Without dependencies**				**With dependencies**			
	With unfolding paths	**Without unfolding paths**	**Δ**	**%**	**With unfolding paths**	**Without unfolding paths**	**Δ**	**%**
1	170.11	170.11	0.00	0.00	170.11	170.11	0.00	0.00
2	91.17	95.25	4.08	4.28	93.37	98.2	4.83	4.92
3	67.07	71.95	4.88	6.88	69.00	79.72	10.72	13.45
4	53.79	61.29	7.50	12.24	57.24	66.16	8.92	13.48
5	48.72	56.05	7.33	13.08	52.30	62.22	9.92	15.94
6	44.85	50.38	5.53	10.98	48.64	58.75	10.11	17.21
7	40.13	45.96	5.83	12.68	46.49	56.70	10.21	18.01
8	39.06	44.04	4.98	11.31	45.06	54.78	9.72	17.74
9	38.16	43.97	5.81	13.21	44.91	54.71	9.80	17.91
10	37.09	44.17	7.08	16.03	44.81	54.91	10.10	18.39
11	36.64	42.90	6.26	14.59	44.81	53.64	8.83	16.46
12	35.87	42.70	6.83	16.00	44.81	53.44	8.63	16.15
13	35.37	42.50	7.13	16.78	44.81	53.24	8.43	15.83
14	34.92	42.30	7.38	17.45	44.81	53.04	8.23	15.52
15	35.14	42.17	7.03	16.67	44.81	52.91	8.10	15.31
16	35.64	42.17	6.53	15.48	44.81	52.91	8.10	15.31
17	36.14	42.17	6.03	14.30	44.81	52.91	8.10	15.31
18	35.64	42.17	6.53	15.48	44.81	52.91	8.10	15.31
19	37.14	42.17	5.03	11.93	44.81	52.91	8.10	15.31
20	37.64	42.17	4.53	10.74	44.81	52.91	8.10	15.31
21	37.66	41.08	3.42	8.33	44.81	51.82	7.01	13.53
22	38.16	41.08	2.92	7.10	44.81	51.82	7.01	13.53
23	38.19	41.08	2.89	7.04	44.81	51.82	7.01	13.53
24	38.14	41.08	2.94	7.16	44.81	51.82	7.01	13.53
…	…	…	…	…	…	…	…	…
35	38.14	41.08	2.94	7.16	44.81	51.82	7.01	13.53

Table 6: *Execution times of a synthetic program when speculation incorrectly predicts the condition; Δ columns contain the absolute difference while % columns contain the percentage of the difference between execution times with and without unfolding paths*

According to Table 7, with speculation predicting the incorrect value and the synthetic program with data dependencies, the number of processes executed in a system with 15 processors is smaller with path unfolding than without. When the number of processors is larger than 15, the number of executed processes stays the same, regardless of the number of processors. This is because in the system using path unfolding and with unused processors, all processes executing both paths of a branch start execution before having the value of the condition. The difference in time between the two methods of dealing with branches arises because without path unfolding, the system must wait for the condition evaluation to start the processes of the correct path, thereby wasting CPU time. Presence of data dependencies in the synthetic program improves the performance of path unfolding because the execution of an incorrect path with such data dependencies is blocked until the dependency is resolved.

	WITHOUT DEPENDENCIES				WITH DEPENDENCIES			
	Incorrect speculation		Correct speculation		Incorrect speculation		Correct speculation	
Cpu count	Without unfolding	With unfolding	Without unfolding	With unfolding	Without unfolding	With unfolding	Without unfolding	With unfolding
1	34	33	33	34	35	34	33	34
2	38	36	33	35	36	34	33	36
3	43	38	33	37	37	34	33	42
4	47	40	33	40	38	34	33	44
5	52	43	33	42	39	34	33	49
6	54	44	33	43	40	34	33	53
7	53	43	33	43	41	34	33	57
8	57	45	33	45	41	34	33	59
9	59	47	33	47	41	34	33	59
10	59	49	33	49	44	34	33	59
11	59	51	33	51	42	34	33	59
12	59	53	33	53	45	34	33	59
13	59	55	33	55	40	34	33	59
14	59	57	33	57	42	34	33	59
15	59	59	33	58	44	34	33	59
...	...	...	...	...	...	...	...	...
35	59	59	33	58	64	34	33	59

Table 7: *Number of processes started in the synthetic program in all investigated cases*

Comparing the performance of the two possibilities, unfolding paths of branches or not (see Figures 13 and 14), we realized that the loss that can take place using unfolding in the worst cases is smaller

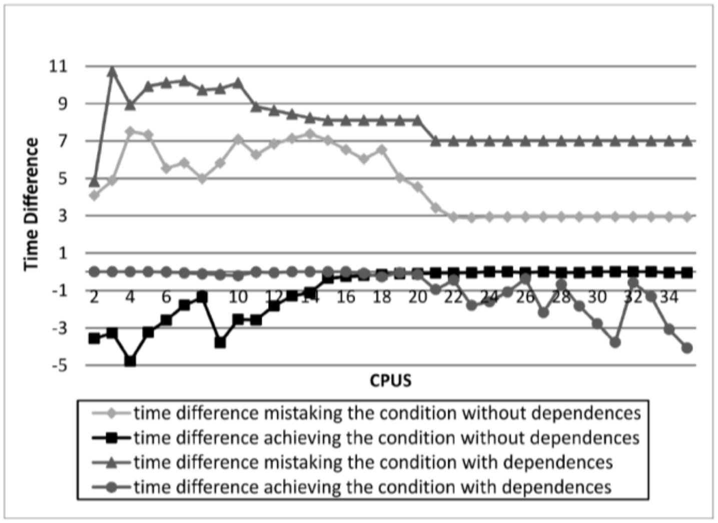


Figure 13: *Comparison of the absolute differences between execution times when speculating correctly and incorrectly on the outcome of a condition*

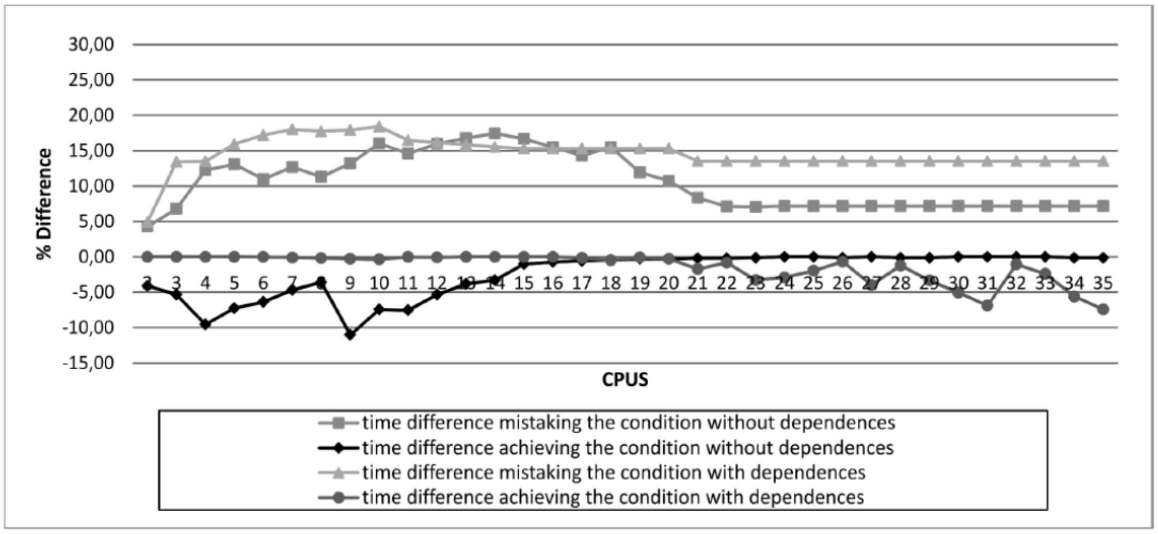


Figure 14: *Comparison of the relative differences between execution times with correct and incorrect speculation*

than the benefit that unfolding can obtain in the best cases. Moreover, this loss decreases with the number of available processes, becoming zero for the large number of such processors, while the benefit is sustained. This is due to the use of processors which otherwise would stay inactive (see Figures 15 and 16). This demonstrates that the use of path unfolding is beneficial in our approach.

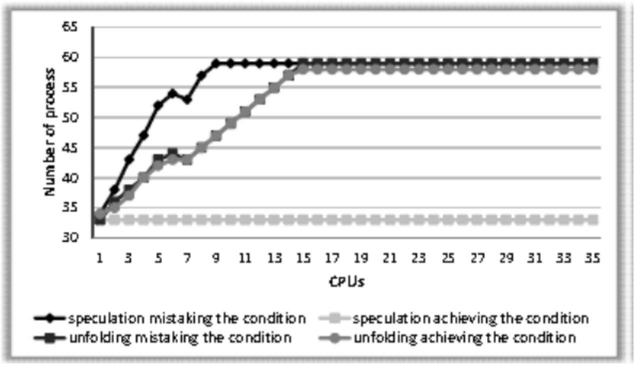


Figure 15: *Number of processes created without data dependencies*

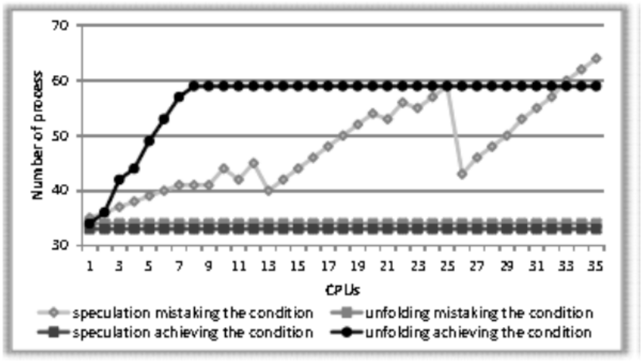


Figure 16: *Number of processes created with data dependencies*

7. ADAPTING THE UNFOLDING PATHS TO REPETITIVE STRUCTURES

The behavior of a condition in a repetitive structure is very different from the previously considered case because in speculation without path unfolding, the branch target buffer (BTB) method can be used to decrease the probability of selecting the incorrect path. As an example, consider a synthetic program from Figure 17 that contains a repetitive structure. There is a condition present in function 3 which is called four times during execution. We assume that the correct path corresponds to the condition yielding false. We also use a 2 bit BTB that defaults initially to the condition yielding true. We have the following cases when comparing the behavior of each speculative method chosen:

1) *Speculation using BTB in the election (see Figure 18).* In this method the first time the iteration (process 3) is reached, it assumes that the condition will yield true and process 4 is executed. Other processes are also started and execute until process 3 finishes its execution. At this point it can be seen that a wrong execution was done. Subsequently, the scheduler eliminates all the processes executed erroneously from the point where the error has taken place. In the second iteration the same thing happens again because the BTB continues indicating that the condition will yield true. In the third iteration, the BTB predicts the condition correctly and from this point forward, it continues predicting all speculations are correctly.

2) Speculation without the use of the BTB in the election (see Figure 19). If the BTB is not used, speculation is always done by choosing the branch set by default. In our case, the election in all four iterations will be incorrect. Therefore, the processes will execute incorrectly after the evaluation of the condition (block 3).

3) Speculation with unfolding paths (see Figure 20). As we have explained previously, in this method every time a branch is found and no unfolding is currently active, the branch will be unfolded and its two paths will be executed in parallel. In this example, we would be executing in parallel four iterations of the two paths. When the branch is evaluated, the incorrect path would be automatically eliminated.

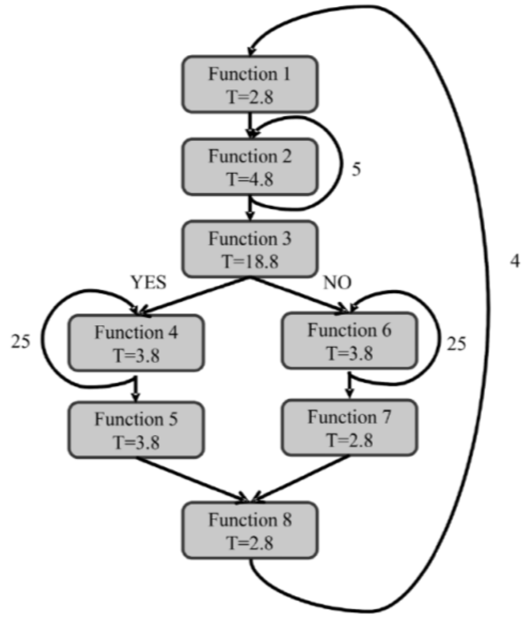


Figure 17: *Synthetic program with repetitive structure*

A comparison of the three methods reveals that speculation without BTB obtains the worst results. This is because it must eliminate the blocks and redo the states of all iterations. In the other two methods, the speculation unfolding paths are good if the number of iterations is small. However, when the number of iterations increases, the advantage is reduced because the speculation with BTB is increasingly successful. This implies that to maintain a good performance, speculation with path unfolding requires a larger number of processors than the speculation with BTB does.

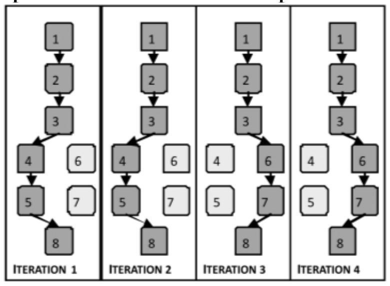


Figure 18: *Execution times using speculation with BTB*

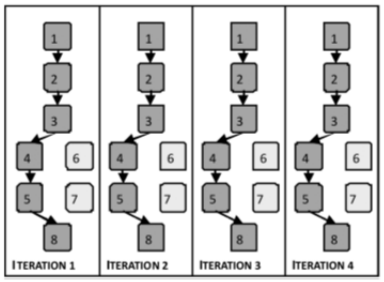


Figure 19: *Execution times using speculation without BTB (set value)*

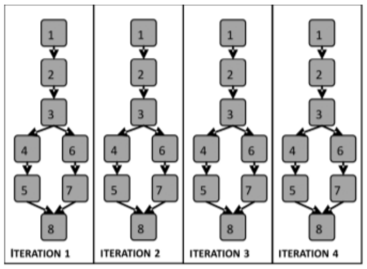


Figure 20: *Execution times using speculation with unfolding paths*

To overcome this disadvantage of the iterative structures for speculation with path unfolding, we introduced a modification of this method: the introduction of the historical statistic of process behavior.

A number of passes through the condition and threshold of the percentage of the same values will dictate what types of method the predictor will use. A branch whose chosen answer is statistically significant is likely to take the same path repeatedly. For such branches, we will not unfold their paths because we expect unfolding to be unnecessary.

It is important to decide what value to use as a threshold, or in other words what percentage of the same values we will consider to be statistically significant. The decision tree for making this decision is shown in Figure 21.

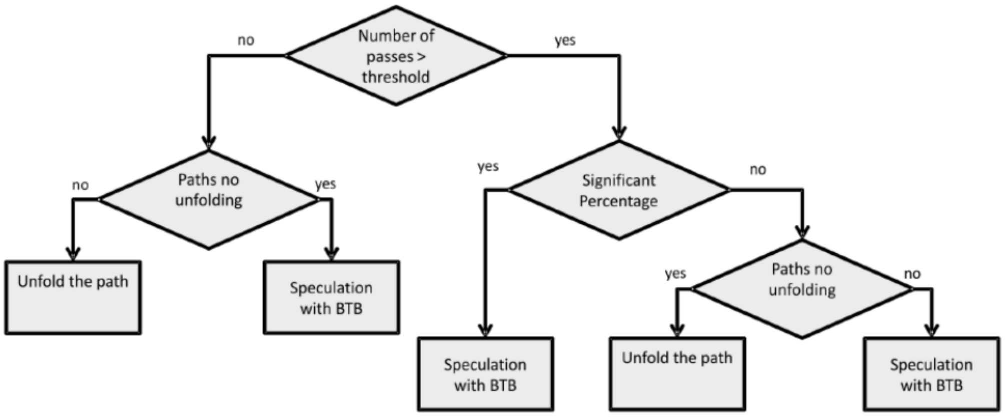


Figure 21: *Scheme of the mix system for speculation with unfolding paths*

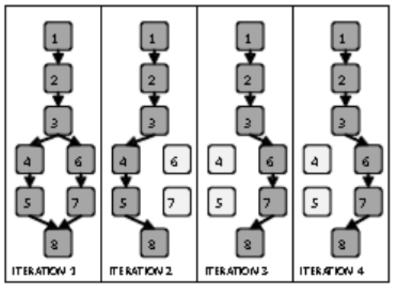


Figure 22: *Execution using speculation with unfolding paths according to statistical values*

If we apply this modification to the path unfolding method in the previous example, assuming that the threshold was three passes and over 75% of the same values, the result would correspond to Figure 22. As can be observed, it preserves the benefits that we obtained with the path unfolding method versus the speculation with BTB method. There is also an added benefit of using this modification in the speculation with the BTB method to improve the speculation success rate. In conclusion, the modification improves both of the discussed above methods.

We executed the synthetic program in the simulator obtaining the results for the four methods shown in Figure 23.

- *Execution Time*

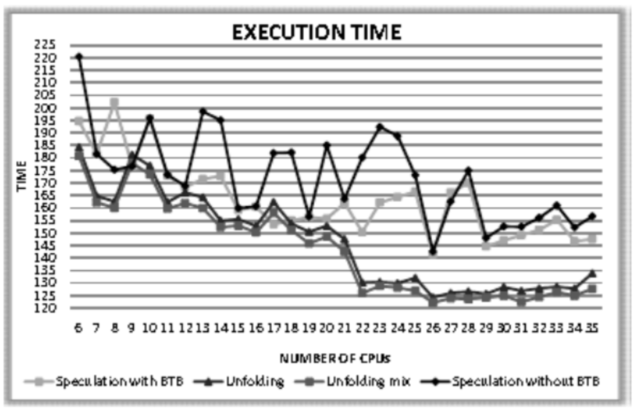


Figure 23: *Comparison of execution times of the different methods*

Cpu count	Speculation without BTB	Speculation with BTB	Unfolding paths	Unfolding mix
1	689.82	689.82	689.82	689.82
2	379.42	379.42	365.40	361.08
3	281.23	281.23	257.92	254.10
4	235.06	235.06	212.16	208.17
5	217.56	217.56	186.62	183.23
6	220.36	194.64	184.22	180.83
7	181.54	181.52	164.76	162.19
8	175.29	202.16	162.56	159.99
9	176.62	176.51	181.23	177.84
10	195.84	195.57	176.79	173.40
11	173.27	172.90	162.30	159.73
12	168.78	168.36	166.22	161.93
13	198.46	171.56	164.22	159.93
14	195.04	172.71	154.92	152.35
15	159.89	159.11	155.47	152.90
16	160.64	160.46	152.82	150.25
17	181.90	153.46	162.29	158.00
18	182.14	154.96	153.69	151.12
19	156.53	156.31	150.46	145.82
20	184.98	155.70	152.82	148.54
21	163.66	161.98	147.53	142.55
22	180.06	150.32	130.18	126.01
23	192.32	162.18	130.39	128.84
24	188.71	164.33	129.95	128.21
25	173.05	166.48	132.08	126.92
26	142.55	141.37	124.27	122.22
27	162.50	166.15	126.02	123.92
28	174.92	169.93	126.62	123.62
29	148.00	144.67	125.67	124.17
30	152.60	147.03	128.42	124.97
				…
35	156.65	147.52	133.92	127.72

Table 8: *Comparison of execution times of the different methods*

The plot in this figure starts with the execution times for six processors because the very high execution times with a smaller number of processors would distort the graph. The graph demonstrates that speculation without BTB obtains the worst result and its execution times are very high because of the large number of paths executed that ultimately are erased. The method of path unfolding performs very well. Although this method executes faster than the speculation method with BTB, when there are many processors available, the difference is reduced. This is because starting from the second iteration, the speculation with BTB finds the correct path. Finally, the mixed method of path unfolding always yields the fastest execution.

- *The number of started processes*

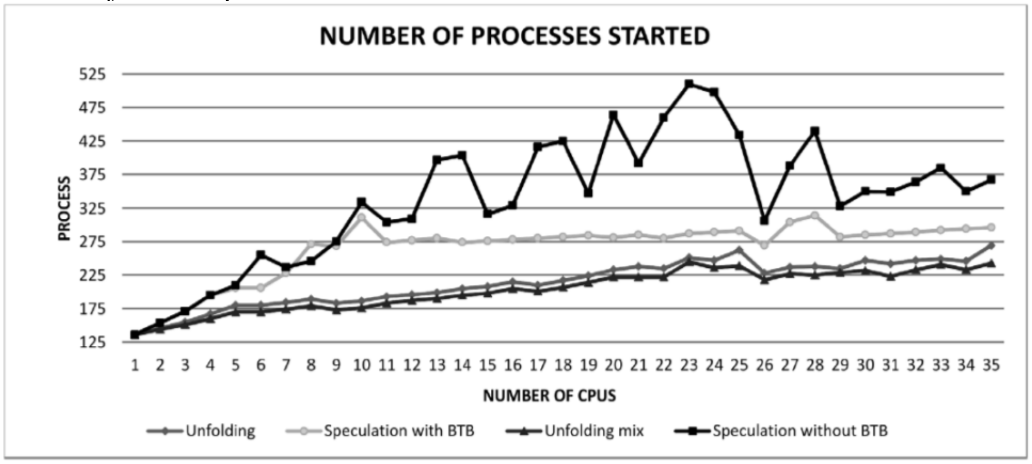


Figure 24: *Comparison of the number of processes started by the different methods*

Cpu count	Speculation without BTB	Speculation with BTB	Unfolding paths	Unfolding mix
1	136	136	136	136
2	154	154	146	144
3	171	171	155	151
4	195	195	167	160
5	210	206	180	170
6	255	206	180	170
7	237	228	184	174
8	246	271	189	179
9	275	268	183	173
10	334	311	186	176
11	304	274	193	183
12	309	277	196	187
13	397	280	199	190
14	404	274	205	195
15	316	276	208	198
…				
20	464	281	233	222
25	434	291	262	239
30	350	285	247	232
35	367	296	269	243

Table 9: *Comparison of the number of processes started by the different methods*

Figure 24 shows the number of processes started by each method. Clearly, the speculation without BTB starts the largest number of processes but it also makes the most mistakes. Both unfolding paths and speculation with BTB produce values quite similar to each other. The speculation for mixed unfolding paths is the best due to combining advantages of the previous two methods.

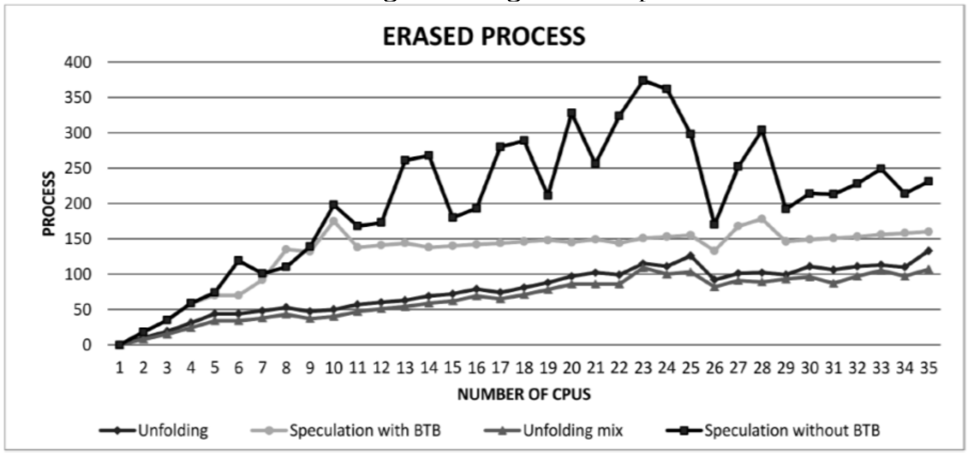


Figure 25: *Comparison of the number of processes erased by the different methods*

Cpu count	Speculation without BTB	Speculation with BTB	Unfolding paths	Unfolding mix
1	0	0	0	0
2	18	18	10	8
3	35	35	19	15
4	59	59	31	24
5	74	70	44	34
6	119	70	44	34
7	101	92	48	38
8	110	135	53	43
9	139	132	47	37
10	198	175	50	40
11	168	138	57	47
12	173	141	60	51
13	261	144	63	54
14	268	138	69	59
15	180	140	72	62
				...
20	328	145	97	86
25	298	155	126	103
30	214	149	111	96
35	231	160	133	107

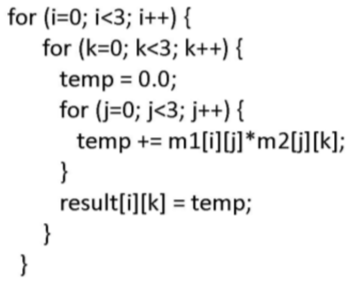
Table 10: *Comparison of the number of processes erased by the different methods*

As shown in Figure 25, the behavior of the methods in terms of the number of erased processes is very similar to behavior observed in terms of the number of the started processes, so the same conclusions apply.

8. EXAMPLES OF REALISTIC PROGRAMS USING MSSPACC

- *Matrix vector multiplication*

In this example, a real algorithm is used (instead of a synthetic one shown in previous section). We selected the following algorithm for the dense matrix (of size *N*x*N*) multiplication:



```
for (i=0; i<3; i++) {
    for (k=0; k<3; k++) {
      temp = 0.0;
      for (j=0; j<3; j++) {
        temp += m1[i][j]*m2[j][k];
      }
      result[i][k] = temp;
    }
 }
```

Figure 26: Selected dense matrix multiplication algorithm

It contains three nested loops. To parallelize this algorithm, the parallelizing subsystem uses the two internal loops as code for the worker and the external loop to define the number of execution times (controlled by the farmer process)[2]. There is a dependence caused by variable "i". This data dependency would cause blocking until the current iteration is ended. Some parallel implementations would resolve this by applying the "loop unrolling" technique (i.e. unfolding all the iterations). Instead, MSSPACC uses speculation in a dynamic way. Therefore the results obtained by MSSPACC would be equivalent to those obtained by a parallel execution when implementing "loop unrolling". The original version of the problem is a sequential one, written in “C” and compiled without loop unrolling or heavy optimization. In comparisons, we report only execution times.

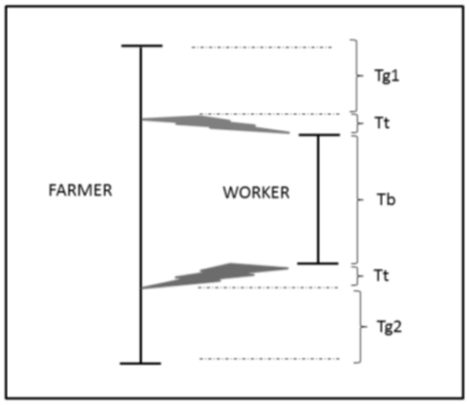
Different matrix sizes have been used to observe the system performance: N = 500, 1000, 2000. A cluster of 22 Intel Core2 Duo E4700 2.60GHz with 1 GB RAM was used. The comparison does not include supercomputers or multiprocessor system because the proposed system is not intended to compete with explicitly parallel programs executed on multiprocessors or scalar processors. Instead, MSCPACC aims at extracting and exploiting parallelism from sequential program executed on computer clusters.

The management time incurred by the farmer process includes the following components (Figure 27):

- *Initialization time* (*Ti*) needed to initialize all processes, start up all workers and initialize all data structures: 7.07 10^{-3} sec.

[2] the codes of workers and the farmer and messages sent between them are described in in detail in [36].

- *Start up time* (*Tg_1*) needed to check if a process can start and to obtain its input values, speculating them if necessary: $1.46\cdot10^{-5}$ sec.
- *Message transmission time* (*Tt*) consumed sending and receiving a message between 2 processes: $2\cdot10^{-6}$ sec.
- *Block execution time (Tb).*
- *Data update time* (*Tg_2*) representing the average time that the farmer process needs to read the worker's messages and to store the resulting values, plus the time needed to sort data writes and to correct speculation errors: $1.9\cdot10^{-5}$ sec.



-

Figure 27: *Management time components*

It should be noted that the management times used by the farmer process are very small with respect to the execution times. The measured absolute values were normalized to the sequential execution time. Figure 28 shows that with larger matrixes, the performance improves thanks to more efficient use of worker processors. Similarly, increasing the number of processors improves performance and lowers management overhead.

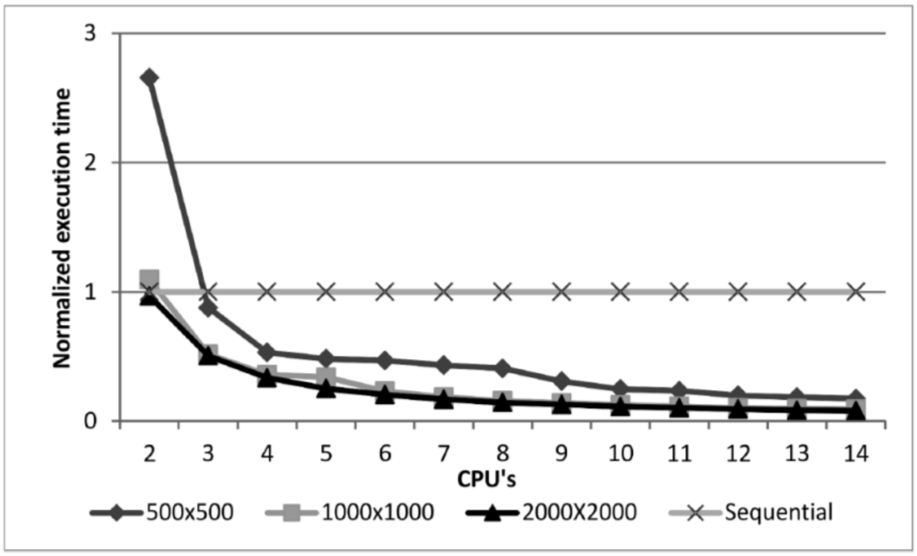


Figure 28: Normalized execution times for MSSPACC

In Table 10 the actual values of the executions are shown. The results (execution time) of the three different experiments are normalized with respect the sequential execution time which is shown in the second column.

# of CPU	Seq.	2	3	4	5	6	7	8	9	10	11	12	13	14
500x500	**0,81**	2,15	0,71	0,43	0,39	0,38	0,35	0,33	0,25	0,2	0,19	0,16	0,15	0,14
1000x1000	**11,12**	12,16	5,74	3,99	3,78	2,56	2,06	1,73	1,56	1,38	1,26	1,13	1,07	1,02
2000X2000	**78,08**	75,57	39,45	26,21	19,63	15,97	13,13	11,22	10,01	8,8	8,03	7,22	6,6	6,27

Table 11: Execution time (sec) for MSSPACC

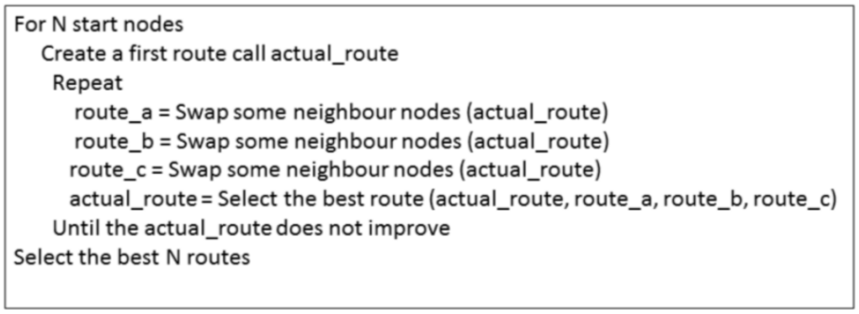
The results show that time management is small relative to run time of blocks. Thus, after applying loop unrolling to the parallelization of matrix multiplication on a cluster, the execution times of the resulting code under PVM or MPI would be similar to MSSCPACC execution times.

In conclusion, the MSSPACC performs well being at least as good as a system using "loop unrolling" in the parallelization of matrix multiplication on a cluster. The main difference is that MSSPACC executes dynamically using speculation.

- *Travelling salesman problem*

The first example showed that using the MSSPACC we can automatically parallelize the sequential algorithm for execution on a cluster getting the same results as the explicit parallelization via loop unfolding would achieve. The second example shows the case in which parallel algorithm restricted by dependencies benefits from speculation introduced by MSSPACC system using the farmer/worker model. We use the well-known "Travelling Salesman Problem" (also studied in [7]) that calculates the shortest Hamiltonian circuit in a graph [6][9][38]. The problem is NP-hard.

We selected the following optimized algorithm [34], designed for parallel execution with or without speculation on a cluster of 20 Pentium III, 1.7 Ghz computers with 512MB RAM:



```
For N start nodes
    Create a first route call actual_route
     Repeat
        route_a = Swap some neighbour nodes (actual_route)
        route_b = Swap some neighbour nodes (actual_route)
       route_c = Swap some neighbour nodes (actual_route)
       actual_route = Select the best route (actual_route, route_a, route_b, route_c)
     Until the actual_route does not improve
Select the best N routes
```

Figure 29: Selected "Travelling Salesman Problem" algorithm

Figure 30 shows the execution times as a function of the number of start cities (varying from 3 to 10) and execution methods: parallel execution without speculation and parallel execution with speculation. In the latter, two different size implementations are used: with a farmer with three workers and three sub-farmers, each of which supervises also three workers, dnoted as (1/3)*4 system with a total of 16 processors, and the similar system with four sub-farmers, denoted as (1/4)*3 system with 20 processors. As it can be observed in Figure 30, the speculative execution method is able to reduce the execution time drastically. Data and control dependences limit the maximum parallelism that the algorithm can efficiently use. In this example, the speculation is able to predict the values of the induction variables easily which significantly increases the parallelism degree of the program. On the other hand, the use of different numbers of farmer-worker groups does not offer a significant enhancement of the execution time.

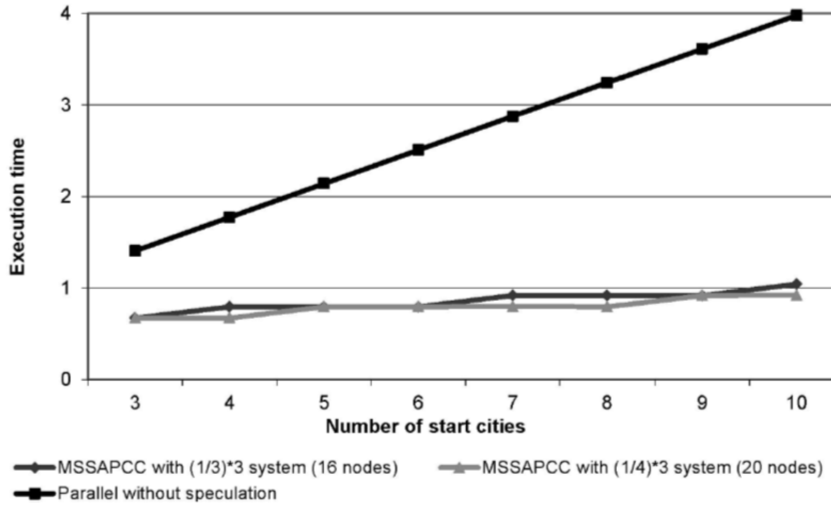


Figure 30: *Execution times for non-speculative and speculative techniques (MASSAPCC)*

9. CONCLUSIONS

The use of unfolding paths in the branch structures allows us to break the control dependencies existing in the code and obtain a high degree of parallelism through the use of currently inactive CPUs. The main challenge of implementing this technique is to efficiently deal with multiple branches. Within longer branches, opening two new paths by unfolding a conditional branch before the previous unfolding is resolved would increase the cost of management, thereby reducing the benefits of such unfolding. To avoid this drawback, we propose to suppress unfolding additional branches until the current branch is resolved and apply speculation to the subsequent branches instead.

In this paper we have compared four possible implementations of dealing with a branch. Two of them use speculation without splitting paths (one with historical information about the behavior of the condition and one without). Two others split paths when a new branch is encountered (one with historical information about the behavior of the condition and one without).

The results demonstrate that the use of unfolding combined with speculation using statistical information (the BTB technique) achieves the best time performance and the highest number of processes executed correctly. These gains are especially high for iterative structures in which the conditions are repeatedly executed. This is due to not splitting the very high percentage of branches that are predicted correctly. In contrast, splitting branches without BTB executes more paths that must be later discarded and therefore gives worse results. When the number of CPUs is large, the results of splitting without BTB improve because the discarded processes use processors that would be otherwise idle and this helps to match the results of the technique without splitting.

In future research we will study how performance evolves when process sizes vary at runtime. The environment will be also modified to enable higher degrees of parallelism and evaluation of the impact of system enhancements on performance.

REFERENCES

[1] Akkary H, Driscoll MA. A Dynamic Multithreading Processor. *Proc. Annual ACM/IEEE International Symp. Microarchitecture (MICRO-31)*, 1998: 226–236.

[2] Ahuja P, Skadron K, Martonosi M, Clark D. Multipath Execution: Opportunities and Limits. *Proc. 12th International Conference on Supercomputing*, 1998: 101–108.

[3] Aragón JL, González J, González A, Smith JE. Dual Path Instruction Processing. *Proc. 16th International Conference on Supercomputing,* 2002: 220–229.

[4] Calder B, Reinman, G, Tullsen, D. Selective Value Prediction. *Proc. 26th Annual International Symp. Computer Architecture*, 1999: 64–74.

[5] Chang PY, Evers M, Patt YN. Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference. *Proc. International Conference on Parallel Architectures and Compilation Techniques*, 1996: 48–57.

[6] Christofides, N. Worst-Case Analisys of a New Heuristic for the Travelling Salesman Problem. *Management Sciences Research Report* 388, 1976.

[7] Dantzig G.B., Fulkerson R., Johnson S.M. Solution of a large-scale traveling salesman problem. *Operations Research* 2, 1954: 393-410.

[8] Ding C, Shen X, Kelsey K, Tice C, Huang R, Zhang C. Software Behavior-oriented Parallelization. *Proc. Programming Language Design and Implementation*, San Diego, USA, 2007: 223–234.

[9] Eilon, S., Watson-Gandy, C., & Christofides, N. *Distribution Management.* Londres: Griffin, 1971.

[10] Franklin M. *Multiscalar Processors.* Kluwer Academic: Cambridge, Massachusetts, 2002.

[11] Gonzalez J, Gonzalez, A. The Potential of Data Value Speculation to Boost ILP. *Proc. 12th International Conference of Supercomputing*, 1998: 21–28.

[12] Grunwald D, Klauser A, Manne S, Pleszkun A. Confidence Estimation for Speculation Control. *Proc. 25th Annual International Symp. Computer Architecture*. Barcelona, 1998: 122–131.

[13] Gwennap L. DanSoft Develops VLIW Design. *Microprocessor Report* 1997; **11**(2):18–22.

[14] Heil TH, Smith JE. Selective Dual Path Execution. *Technical Report, University of Wisconsin-Madison, ECE*, 1997.

[15] Hennessy JL, Patterson DA. *Computer Architecture: A Quantitative Approach, 2nd edition*, Morgan Kaufmann Publishers Inc: San Francisco, California, 1996.

[16] Hwu WW, Conte TM, Chang, PP. Comparing Software and Hardware Schemes For Reducing the Cost of Branches. *Proc. 16th Annual International Symp. Computer Architecture*, 1989: 224–233.

[17] Jacobson Q, Bennett S, Sharma N, Smith, J. Control Flow Speculation in Multiscalar Processors. *Proc. 3rd International Symp. High-Performance Computer Architecture*, 1997: 218–229.

[18] Jiang Y, Mao F, Shen X. Speculation with Little Wasting: Saving Cost in Software Speculation through Transparent Learning. *Proc. 15th International Conference Parallel and Distributed Systems (ICPADS),* 2009: 543–550.

[19] Khanna R, Verma S, Biswas R, Singh JB. Implementation of Branch Delay in Superscalar Processors by Reducing Branch Penalties. *Proc. International Advance Computing Conference (IACC),* 2010: 14–20.

[20] Klauser A, Paithankar A, Grunwald D. Selective Eager Execution on the PolyPath Architecture. *Proc. International Symp. Computer Architecture*, 1998: 250–259.

[21] Klauser A, Grunwald D. Instruction Fetch Mechanisms for Multipath Execution Processors. *Proc. Annual ACM/IEEE International Symp. Microarchitecture (MICRO-32)*, 1999: 38–47.

[22] Krishnan V, Torrellas J. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. Computers,* 1999, 48(9): 866–880.

[23] Larson E, Austin T. Compiler Controlled Value Prediction Using Branch Predictor Based Confidence. *Proc. 33rd Annual ACM/IEEE International Symp. Microarchitecture (MIRCO-33)*, 2000: 327–336.

[24] Lee J, Smith A. Branch Prediction Strategies and Branch Target Buffer Design. *Computer*, 1984; **17**:6–22.

[25] Lee CJ, Kim H, Mutlu O, Patt Y. A Performance-Aware Speculation Control Technique Using Wrong Path Usefulness Prediction, *HPS Technical Report, TR-HPS-2006-010,* The University of Texas at Austin, 2006.

[26] Lipasti M, Wilkerson C, Shen J. Value Locality and Data Speculation. *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996: 138–147.

[27] Malik K, Agarwal M, Dhar V, Frank MI. Paco: Probability-based Path Confidence Prediction. *Proc. International Symp. High Performance Computer Architecture (HPCA)*, 2008: 50–61.

[28] Marcuello P, Gonzalez A. Clustered Speculative Multithreaded Processors. *Proc. International Conference Supercomputing,* 2001: 365–372.

[29] McFarling S, Hennessy J. Reducing the Cost of Branches. *Proc. 13th Annual International Symp on Computer Architecture*, 1986: 396–404.

[30] McFarling S. Combining Branch Predictors. WRL *Technical Notes TN-36*, DigitalWestern Research Laboratory, 1993.

[31] Olukotun K, Hammond L, Willey M. Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. *Proc. International Conference Supercomputing*, 1999: 21–30.

[32] Pan S, So K, Rahmeh J. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992; Boston, MA: 76–84.

[33] Petit S, Sahuquillo López JP, Ubal R, Duato J. A Complexity-Effective Out-of-Order Retirement Microarchitecture. *IEEE Transactions on computers*, 2009; **58**(12):1626–1639.

[34] Puiggali J, Jové T, Salanova S, Marzo J. Execution Speed Up Using Speculation Techniques in Computer Clusters. *Proc. International Mediterranean Modelling Multiconference (IMM),* 2006: 561–568.

[35] Puiggali J, Jové T, Salanova S, Marzo J. Limit of TLS Execution of Sequential Programs on Clusters. *Proc. International Symp. Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, 2006: 12-19.

[36] Puiggali J, Jové T, Segovia J, Marzo J. Master/Slave Speculative Parallelization Architecture for Computer Clusters. *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*, 2007: 1-16

[37] Puiggalí J, Jové T, Marzo JL. Out-of-Order Execution in Master/Slave Speculative Parallelization Architecture for Computer Clusters. *Proc. International Simulation Multi-Conference (ISMC)*, 2011: 179–184.

[38] Ravikumar, C. Parallel techniques for solving large scale travelling salesperson problems. *Microprocessors and Microsystems, 16*(3), 1992: 149-158.

[39] Sazeides Y, Vassiliadis S, Smith J. The Performance Potential of Data Dependence Speculation and Collapsing. *Proc. Annual IEEE/ACM International Symp. Microarchitecture (MICRO-29),* 1996: 238–247.

[40] Sazeides Y, Smith J. The Predictability of Data Values. *Proc. Annual IEEE/ACM International Symp. Microarchitecture (MICRO-30),* 1997: 248–258.

[41] Seznec A. Storage Free Confidence Estimation for the TAGE Branch Predictor. *Proc. 17th International Symp. High Performance Computer Architecture (HPCA),* Feb. 2011: 443–454.

[42] Šilc J, Ungerer T, Robic B. Dynamic Branch Prediction and Control Speculation. *International Journal High Performance Systems Architecture*, 2007; **1**(1): 12–13.

[43] Smith J. A Study of Branch Prediction Strategies. *Proc. International Symp. Computer Architecture*, 1991: 135–148.

[44] Steffan J, Colohan C, Zhain A, Mowry T. Improving value communication for thread-level speculation. *Proc. International Symp. High-Performance Computer Architecture (HPCA)*, 2002: 65–75.

[45] Tian C, Feng M, Nagarajan V, Gupta R. Copy or Discard Execution Model for Speculative Parallelization on Multicores. *Proc. Annual ACM/IEEE International Symp. Microarchitecture (MICRO-41)*, 2008: 330–341.

[46] Trias A, Aciar S, de la Rosa JL, Puiggalí J, Jové, T. An Agents Approach for Master/slave Hierarchical Clusters. *6th European Workshop on Multi-Agent Systems (EUMAS)*, 2008: 59-70.

[47] Trias A, Puiggalí J, Castro F, Jové T, Sbert M, Marzo JL. Speculative Parallelization of Multipath Radiosity Algorithm. *Proc. 12th International Symp. Performance Evaluation of Computer & Telecommunication Systems (SPECTS)*, 2009: 89–95.

[48] Tullsen D. Simulation and Modeling of a Simultaneous Multithreading Processor. *Proc. 22nd Annual Computer Measurement Group Conference*, Dec. 1996: 819–828.

[49] Tullsen D, Eggers S, Emer J, Levy H, Lo J, Stamm R. Exploiting choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. *Proc. 23rd Annual International Symp. Computer Architecture*, 1996: 191–202.

[50] Tullsen D, Eggers S, Levy H. Simultaneous Multithreading: Maximizing On-chip Parallelism. *Proc. 22nd Annual International Symp. Computer Architecture*, 1995: 392–403.

[51] Uht A, Sindagi V. (1995) Disjoint Eager Execution: an Optimal Form of Speculative Execution. *Proc. Annual ACM/IEEE International Symp. Microarchitecture (MICRO-28),* Ann Arbor, MI, 1995: 313–325.

[52] Ungerer T, Robic B, Šilc J. Multithreaded Processors. *The Computer Journal*, 2002; **45**(3):320–348.

[53] Ungerer T, Robic B, Šilc J. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, 2003; **35**(1):29–63.

[54] Unger A, Zehendner E, Ungerer T. A Combined Compiler and Architecture Technique to Control Multithreaded Execution of Branches and Loop Iterations. *ACM SIGARCH Computer Architecture News*, 2000; **28**(1):53–61.

[55] Unger A, Ungerer T, Zehendner E. Static Speculation, Dynamic Resolution. *Proc. 7th Workshop on Compilers for Parallel Computers,* Linkoping, Sweden, 1998: 243–253.

[56] Wallace S, Calder B, Tullsen D. Threaded Multiple Path Execution. *Proc. International Symp. Computer Architecture*, 1998: 238–249.

[57] Xekalakis P, Cintra M. Handling Branches in TLS Systems with Multi-Path Execution. *Proc. International Symp. High Performance Computer Architecture (HPCA),* 2010: 1–12.

[58] Yanyan G, Xi L. Formal Verification of Out-of-order Processor. *Proc. International Conference on Computer Modeling and Simulation (ICCMS),* 2009: 129–135.

[59] Yeh T, Patt Y. A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History. *Proc. 20th Annual International Symp. Computer Architecture, San Diego, CA*, 1993: 257–266.

[60] Yeh T, Patt Y. Two-level Adaptive Branch Prediction. *Proc. Annual ACM/IEEE International Symp. Microarchitecture (MICRO-24)*, 1991: 51–61.

[61] Zilles, C, Sohi G. Master/slave Speculative Parallelization. *Proc. Annual ACM/IEEE International Symp. Microarchitecture (MICRO-35)*, 2002: 85–96.