Control Speculation in Multithreaded Processors through Dynamic Loop Detection

Jordi Tubella and Antonio González
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya,
Campus Nord, Jordi Girona 1-3, Edifici D6, 08034 Barcelona, Spain
e-mail: {jordit,antonio}@ac.upc.es

Abstract

This paper presents a mechanism to dynamically detect the loops that are executed in a program. This technique detects the beginning and the termination of the iterations and executions of the loops without compiler/user intervention. We propose to apply this dynamic loop detection to the speculation of multiple threads of control dynamically obtained from a sequential program. Based on the highly predictable behavior of the loops, the history of the past executed loops is used to speculate the future instruction sequence. The overall objective is to dynamically obtain coarse grain parallelism (at the thread level) that can be exploited by a multithreaded architecture. We show that for a 4-context multithreaded processor, the speculation mechanism provides around 2.6 concurrent threads in average.

1. Introduction

Control speculation increases the potential parallelism that a processor can exploit (see [7] [12] among others). Branch prediction is the most studied control speculation technique, and it is incorporated in the large majority of past and current microprocessors (see [8] [13] among others). Such speculation approaches have been oriented to superscalar processors. However, little research has been done on control speculation for other emerging architectures. In particular, this work focuses on *multithreaded architectures*. In this paper, a multithreaded architecture refers to any architecture that can concurrently execute several threads of control[1] from a single sequential program, regardless of the approach used to obtain such threads. For instance, the multiscalar [9] architecture belongs to this class of architectures. In this particular case, the partition of a program into threads requires some compiler support.

The work that is presented in this paper is aimed at obtaining multiple threads from a sequential program without any user/compiler intervention. There are several reasons to argue for a hardware mechanism: (i) the static analysis that the compiler may perform is less accurate than the actual dynamic behavior than can be obtained during the execution; (ii) profiling mechanisms that characterize the execution of a program rely on a concrete set of input data; (iii) the instruction set architecture is not modified, giving backward compatibility with previous implementations. Inter-thread control speculation is based on a dynamic detection of the loops in a program. The threads are obtained based on the highly predictable behavior of the loops.

We present a general mechanism that allows the detection of loops with a reasonable cost and we also describe the application of this dynamic loop detection to implement the inter-thread control speculation in a multithreaded architecture. The results show that loops are an important source for accurate thread prediction. For instance, we show that for a 4-context processor, the proposed thread speculation approach can provide about 2.6 concurrent threads in average.

In addition to inter-thread control dependences, the proposed mechanism can be used to speculate on both inter-thread data dependences and the data that flow through them. The definition and implementation of a particular data/data dependence speculation mechanism is beyond the scope of this paper. However, to show its potential, we present some preliminary statistics about the predictability of values of live-in[2] registers and memory locations of speculative threads.

1. In this paper, a thread of control (or thread for short) refers to any contiguous region of the dynamic instruction sequence.

There are several proposals in the literature regarding multithreaded architectures oriented to the execution of a single sequential program. The most remarkable works are the Expandable Split Window paradigm [3], the Multiscalar [9], the SPSM [2], the Superthreaded [11] and the Multithreaded Decoupled [1] architectures. However, all of them require either some user/compiler intervention and/or some extensions in the instruction-set architecture. Control flow speculation for Multiscalar processors has been studied in a recent paper [5]. In such proposal the threads (called tasks in that paper) are delimited at compile time and the run-time mechanism is only responsible for predicting the sequence that such threads will follow. This paper proposes a novel approach that can be used by such type of architectures in order to obtain multiple threads of control by means of hardware mechanisms.

The proposed speculation mechanism is based on a loop detection scheme. Dynamic loop detection has been studied in [6] but the mechanism proposed in that paper is oriented to extract statistics from a trace generated by a program execution. It is not adequate for a hardware implementation and it does not care about control speculation.

This paper is organized as follows. Section 2 describes and analyzes the mechanism proposed for the dynamic detection of loops. Section 3 presents its application to thread control speculation. Section 4 presents some preliminary statistics about data speculation issues. Finally, the main conclusions are summarized in section 5.

2. Dynamic loop detection

Loops are a very common control structure in every program. A high percentage of all instructions executed in a program belong to loops. Since in addition the closing branches of loops are highly predictable, loops are potentially useful to perform control speculation. In this section we first define the types of loops that we consider in this paper. Then, an implementation to dynamically detect such loops and the performance exhibited for the SPEC95 benchmark suite is presented.

2.1. Loop definitions

For structured code, there is a unanimous definition of what a loop is. However, for non-structured code different interpretations can be applied. In this subsection, we present our particular definition of loops, which will be used in the rest of this paper. First, we define the static view of a loop in a program. Then, we define the dynamic view of a loop with the concepts of loop execution and loop iteration.

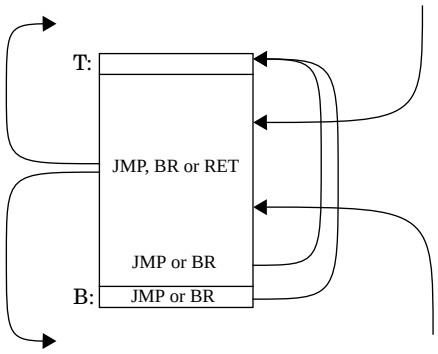

Figure 1: Static view of a loop.

There is a **loop** in a program, which it is identified by address **T**, when there is at least one backward branch or jump to address T. There may be more than one branch with the same target address. In this case, we consider that all such branches are closing branches of the same loop. A main attribute of a loop is the highest address that contains a backward branch or jump to address T. This address is denoted by address B. All instructions in the range of addresses [T,B] constitute the **body of loop T**. Once entered in the loop body, it is possible to leave it with an instruction that forces the execution control flow to proceed to an address outside the range of the loop. This control flow instruction may be a branch, a jump or a return instruction. There may be any number of subroutine activations inside a loop body. Note that this definition of the static loop body does not include the bodies of the subroutines that are activated.

Figure 1 shows a static view of a loop. Exit branches can be at the beginning or at the end of the loop body in the case of *while* or *do_while* high-level loop structures. It is also possible to leave a loop from any other part of it, as it is the case of a *break*, *goto* or *return* high-level language instructions.

On the other hand, we also define a dynamic view of a loop. Considering address B as the highest address of all executed branch or jump instructions to target address T, an **execution of loop T** consists of a certain number of sequentially executed instructions which are delimited by the following conditions.

An execution of loop T is initiated when the first instruction whose address belongs to the loop body (range of addresses [T,B]), is executed.

An execution of loop T is terminated by one of the following instructions: (i) a not taken branch at address B, or (ii) a taken branch or a jump at an address belonging to the

2. A live-in register (live-in memory location) is a register (memory location) that is live-on-entry to a thread.

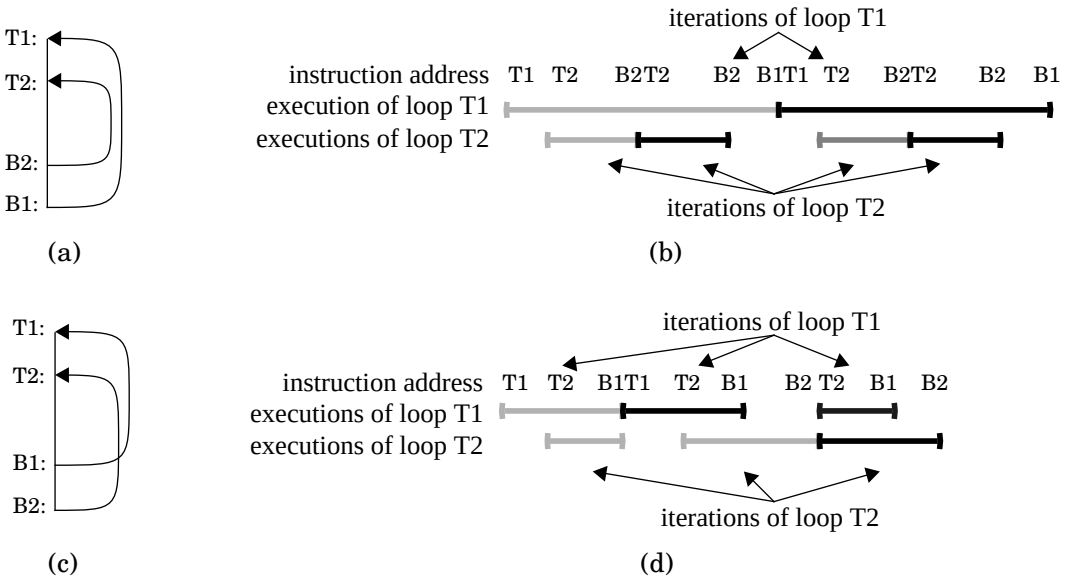

Figure 2: Nested and overlapped loops: (a) Static view of two nested loops; (b) Dynamic samples of loop executions and loop iterations of two nested loops; (c) Static view of two overlapped loops; (d) Dynamic samples of loop executions and loop iterations of two overlapped loops.

loop body to a target address outside the loop body, or (iii) a return instruction at an address belonging to the loop body.

Since usually any subroutine activation returns to the address below the call instruction, we consider that inside a loop execution there may be any number of nested subroutine activations. Note that instructions belonging to the subroutine body also belong to the loop execution.

Moreover, when a loop is inside a recursive subroutine, note that the different instantiations of the same loop that are obtained through recursive activations without any return in between are considered to belong to the same loop execution.

All instructions in a loop execution are divided into a certain number of loop iterations. An **iteration of loop T** consists of a certain number of sequentially executed instructions belonging to an execution of loop T with the following characteristics. The first iteration of a loop T is started when its loop execution is also initiated. The rest of the iterations always begin at address T. All iterations of loop T, except the last one, always finish with a taken backward branch or jump to address T. The last iteration finishes when its loop execution also finishes.

A given dynamic instruction may belong to several loop executions. This occurs when loop structures are nested or overlapped.

Loops T1 and T2, with corresponding B1 and B2 branch addresses, are nested when the range of addresses [T2,B2] is included into [T1,B1]. In this case, loop T2 is the innermost, and all instructions in the execution of loop T2 also belong to the execution of loop T1. Loops T1 and T2 are overlapped when T2 > T1 and B2 > B1. In this case, instructions in the first iteration of one of these loops also belong to the execution of the other loop. Figure 2 shows some samples of loop executions and loop iterations when two loops are either nested or overlapped.

2.2. Hardware mechanism for loop detection

In order to detect loop executions and loop iterations, we introduce the Current Loop Stack (CLS). This stack is devoted to contain all loops which are being currently executed. The top of the stack corresponds to the innermost loop, and the remaining loops are stored according to the nesting order.

The elements in the CLS contain two fields (T,B). Field T stores the target address of a loop (its identifier) and field B stores the highest address of all branch or jump instructions executed so far to address T. The CLS is updated when executing three kinds of instructions (branch, jump and return) in the following manner (consider PC as their instruction address).

Whenever a backward branch or jump instruction to target T is executed, the CLS is searched. If there is not any entry with target address T and the branch is taken, it means that a new loop execution is started. In this case,

loop (T,PC) is pushed onto the CLS. If the branch is not taken, no action is performed. It means that a loop with only one iteration has been executed. If loop T is found in the entry *i* of the CLS and the branch is taken, an iteration of loop T has finished and consequently, a new iteration of the same loop execution is started. The CLS entries in the range [top,i+1] are popped out (we assume that the top of the stack corresponds to the highest address). If PC is higher than the value of field B, this field is updated. If the branch is not taken and the value of field B is lower than or equal to PC, it means that both the iteration and the execution of loop T have finished. The CLS entries in the range [top,i] are popped out.

Whenever the address of a jump or a taken branch belongs to a loop in the CLS, it is checked whether the target address is outside the loop body. All loops that meet this condition are removed from the CLS (i.e., it is considered that their executions have finished). Finally, for any executed return instruction, all loops in the CLS whose body comprise such instruction are also popped out.

In the most frequent case, when an iteration of a loop finishes, the entry related to this loop is at the top of the CLS. Note that the top of the CLS corresponds to the innermost loop that is being executed. Nevertheless, there are two situations that may cause an iteration of a loop that is not located at the top of the CLS to finish. In these situations, all innermost loops are popped out (as described two paragraphs above), and thus, their execution is finished.

The first situation occurs when a subroutine call in a loop body never returns to this loop body (e.g., when the *setjmp()* library call is used). If this loop is nested inside another one, when an iteration of the outer finishes it implies the termination of the inner one. It could also happen that no outer loop exists, and thus, the loop would remain in the CLS at the end of the execution. Nonetheless, we have observed that the CLS is always empty at the end of the entirely execution of the SPEC95, which means that this event never happens for this benchmark suite. In any case, such situation could be handled by periodically flushing the contents of the CLS.

The second situation is caused by not differentiating the instantiations of the same loop T produced in recursive subroutine activations. For instance, given the following structure of a recursive subroutine:

```
s() {
    if () {
          for () s();      /* loop T1 */
    } else {
          for () s();      /* loop T2 */
    }
}
```

Suppose that initially loop T1 is being executed. The recursive call to `s()` causes a new activation of the subroutine and this time the else part is executed. Since call instructions do not terminate a loop execution, T2 is considered to be nested into the previous T1 execution. If `s()` is again activated from T2 and then T1 is executed, this loop will be found in the CLS and it will be considered that a new iteration of this loop begins. In this case, loop T2 is popped out and is considered to be terminated. The next iteration of T2 will be considered as a different execution. Notice that this is just one possible way to classify loop iterations into loop executions in the presence of recursive subroutines. Anyway, this event rarely happens and thus, it has a very low influence on the final performance.

Dynamic loop detection is based on identifying backward control transfer instructions. This means that the first iteration of a loop execution is not detected until it has finished. Thus, a loop is not considered until the second iteration begins. In this way, figure 2 depicts the first iteration of each loop execution in grey because it is not detected with the proposed mechanism.

When the CLS is full and a new loop must be pushed onto it, the deepest entry is lost. This policy tends to penalize the outermost loops, which are the least common ones. However, as it is shown in section 2.2.1, a few entries are enough to guarantee no overflow for most programs.

2.2.1 SPEC95 loop statistics. The previous mechanism to detect loops has been applied to obtain statistics of the SPEC95 benchmark suite. The methodology to collect these data (and the rest of data presented in the paper) is the following. The benchmarks have been compiled using the DEC Alpha compiler with the following options: *-O5 -tune ev5 -migrate -ifo* (for C programs) and *-O5 -tune ev5* (for Fortran programs). They have been instrumented with the *atom* tool [10] and run using the reference input data, except for the *gcc*, *ijpeg* and *perl* programs. For these programs, a single file of the reference input data has been used.

Table 1 shows the number of instructions (#instr/10^9), the static number of loops (#loops), the average number of loop iterations per loop execution (#iter/exec), the average number of instructions per loop iteration (#instr/iter), the average nesting level (avg. nl) and the maximum nesting level (max. nl). These figures correspond to the whole execution of the programs.

2.3. Gathering loop information

The above presented CLS allows to detect the start and the end of loop iterations and loop executions. Such information can be used by a multithreaded processor to create multiple threads of control, each one corresponding to a different iteration of a loop. However, in general such a control speculation approach will require additional infor-

	#instr/ 10^9	#loops	#iter/ exec	#instr/ iter	avg. nl	max. nl
applu	53.02	189	3.50	261.08	5.16	7
apsi	33.06	207	10.75	229.34	3.14	5
compress	61.05	45	6.27	84.65	2.52	4
fpppp	144.49	83	3.05	3217.80	6.66	9
gcc	1.93	1229	5.28	80.21	3.43	7
go	38.87	709	3.76	156.60	4.86	11
hydro2d	50.57	291	29.37	127.66	3.50	4
ijpeg	40.98	198	20.75	336.26	6.37	9
li	70.77	94	3.48	107.80	5.15	10
m88ksim	79.19	127	9.38	39.82	1.98	5
mgrid	102.81	142	28.93	512.68	4.93	6
perl	30.66	147	3.11	47.02	1.35	5
su2cor	40.23	213	51.23	257.17	3.50	5
swim	40.75	79	188.54	278.89	2.99	3
tomcatv	32.05	91	57.18	224.82	3.01	4
turb3d	96.27	152	4.11	239.44	3.97	6
vortex	94.98	220	12.08	215.56	3.06	6
wave5	35.69	195	56.15	164.25	3.12	5

Table 1: Loop statistics.

mation about the loop iterations and the loop executions. For instance, it can be useful to know the number of iterations per execution, or the live-in register values of each iteration. The former can be used to determine the number of threads that are to be created whereas the latter can be used to enforce data dependences among threads. In general, two types of information are required: one at the level of the loop iteration and the other at the level of the loop execution. The particular information to be stored depends on the concrete implementation. In this subsection, we describe a general framework that is common for any implementation.

The information previously mentioned is stored by means of two tables (LET and LIT). The LET, which stands for Loop Execution Table, stores information about previous loop executions. The LIT, or Loop Iteration Table, is used to characterize the iterations of a loop.

These tables are associatively searched, and every entry is identified by the same identifier of loops, that is, the loop target address T. Entries in both tables are inserted when the execution of a loop starts. We consider a LRU replacement policy. More concretely, the entry discarded in LIT corresponds to the loop that has initiated a new iteration least recently, while the entry discarded in LET corresponds to the loop that has initiated a new execution least recently.

During the execution of the program, these tables are

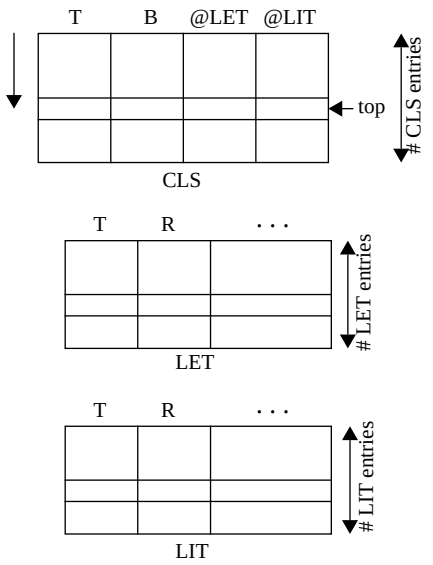

Figure 3: CLS, LET and LIT structures.

accessed in order to know the behavior of a given loop. In this case, entries in the tables are directly accessed through pointers that are stored in the CLS stack. A NULL pointer is used when the loop is not stored in the related table.

Figure 3 depicts the data structures used to dynamically detect loops (CLS) and to gather information about them (LET and LIT). Entries in the CLS contain the target and branch addresses of a loop (fields T and B, respectively), and the corresponding pointers to LIT and LET (fields @LET and @LIT, respectively). Entries in the LIT and the LET contain field T (target loop address) and field R (used to implement the LRU replacement policy). The rest of the fields in each table entry depend on the kind of information that it is decided to gather from the loops for each particular multithreaded implementation.

Our ongoing work focuses on using the LET to predict the number of iterations of each loop and to generate speculative threads accordingly. In order to implement a stride predictor, each LET entry contains, in addition to the T and R fields, the last iteration count and the difference between the previous two counts. The LIT is used to store information related to the live-in registers and memory locations of the last iteration of the loop. For each live-in register or memory location it stores the value at the beginning of the last iteration and the last stride, so that live-in values of future iterations can be predicted with a stride predictor. Besides, for live-in memory locations it stores the last effective address and the last stride so that inter-thread memory operations can be speculated through address prediction using a similar scheme as that proposed in [4] for superscalar processors. In this way, threads cor-

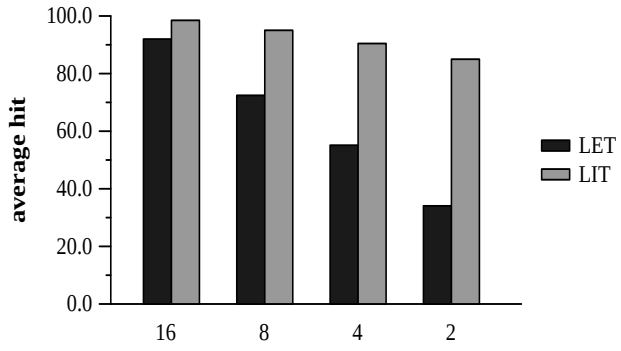

Figure 4: LET and LIT hit ratios.

responding to different (maybe dependent) iterations can proceed in parallel, without any synchronization if the predicted values are correct.

2.3.1 Performance. Depending on the particular implementation, the contents of the LIT/LET are useful after several iterations/executions. For instance, to predict the number of iterations as the last number of iterations of the same loop, only a previous execution is required. On the other hand, if the prediction is based on a stride predictor, two executions are required to compute a stride. To evaluate the performance of the proposed scheme, we consider that the contents of the LIT/LET are useful after two iterations/executions.

The performance of this general mechanism to gather information about the loops is measured through LET and LIT hit ratios. The LET hit ratio measures, when a new execution of a loop is started, whether two complete executions of the same loop have been detected since it was stored in the table. The LIT hit ratio measures, when a loop iteration starts, whether two complete iterations have been detected since it was stored in the table. This condition is not tested for the first iteration of all the executions because the first iteration is not detected until it finishes.

Figure 4 shows the average LET and LIT hit ratios for the whole SPEC95 benchmarks. The number of CLS entries is 16, which is enough to store the maximum number of current loops (we have shown in table 1 that the maximum nesting level is lower than 16). The number of entries of the LIT and LET is 2, 4, 8 and 16. A trade-off between the space needed for the tables and the hit ratio could be to choose 4 entries for the LIT (90.50% hit ratio) and 16 entries for the LET (91.98% hit ratio). If a large quantity of information is stored in each table, a 2-entry LIT and a 8-entry LET have also an acceptable performance (85.00% and 72.44% hit ratios, respectively).

2.3.2 Additional issues. When a new loop is executed, the proposed approach always inserts a new entry in both tables for that loop. It may be convenient to disable the recognition of some loops by introducing a new table containing those potential loops that are not suitable for speculation. This table should be associatively accessed before inserting a new entry in the LET and the LIT. For example, those loops with a poor prediction rate may be good candidates to store in this table. In this way, a loop with more reliable information is not eliminated.

Taking into account that it is preferable to store the innermost loops in front of the outermost, we have considered an alternative replacement algorithm that inhibits the insertion of a loop in the LIT and the LET when it implies to eliminate a loop that is nested into it. This policy needs to store for each loop, which other loops are nested into it. It has been evaluated that the improvement on the hit ratio is negligible with respect to the LRU algorithm. This is so because when the nesting level of loops is not higher than the number of entries of the LIT and LET, the behavior of this policy is identical to LRU. We have shown that the average nesting level of loops in the SPEC95 is not very high. Thus, we have not considered any more this policy.

3. Control speculation in multithreaded processors

In this section we show how to apply the previous scheme to dynamically generate threads, obtained from a single sequential program, for a general multithreaded architecture.

We consider a multithreaded architectural model consisting of several thread units (TUs) which are able, at least, to fetch and decode instructions from different parts (or threads) of the same sequential program. The rest of the instruction stages may be performed by either a replicated or a shared set of functional units. The TUs of a multithreaded architecture can be in a non-speculative, speculative or idle state. Initially, there is one non-speculative TU and the rest of them are idle. When thread control speculation is performed, some idle TUs change to the speculative state. All speculative TUs maintain the order in which the associated threads must be executed inside the sequential program. When a speculative TU reaches its termination point, it waits until the non-speculative TU confirms the control flow of the execution, that is, the speculative thread becomes non-speculative.

A particular implementation of the multithreaded model must define three additional issues regarding control speculation:

- When inter-thread control speculation can be performed?
- Which are the threads to be speculated?
- When the verification of a speculation must be done?

For instance, the mechanism defined in [5] for the multiscalar processor define these 3 issues in the following manner. Speculation can be done when a task is initiated. Notice that the responsibility to arrange the code into tasks relies on the compiler, unlike the approach proposed in this paper. When a task begins, it is predicted the following task based on the recent history. The verification of the speculation is performed when the task that provoked the speculation finishes. In the scheme proposed in this paper, all these issues involving thread control speculation are done entirely by hardware, based on loop detection.

We define the concept of *thread-level parallelism (TLP)* to refer to the parallel execution of threads. Thread-level parallelism is measured through the average number of active and correctly speculated threads per cycle (TPC). The TPC is the main source of additional parallelism that can be provided by the novel control speculation approach. In addition to the TPC, the final amount of parallelism exploited by a particular implementation will depend on the approach to deal with inter-thread dependences and intra-thread parallelism.

The potential TLP that can be exploited if loops are automatically detected is very high. Figure 5 presents the TPC that a thread speculation mechanism based on loop detection for an ideal machine with infinite TUs can provide. This mechanism speculates only when the non-speculative thread detects a loop execution. The management of data dependences is an orthogonal issue with respect to control speculation that will be considered in future work, as outlined in section 2.3. For each program, the left bar corresponds to the TPC when executing all instructions, whereas the right bar reflects the execution of the first 10^9 instructions. It can be seen that most programs behave approximately in the same way when executing only a reduced part of it. In the rest of the paper, figures will only refer to the reduced part of the program. It is important to emphasize the potential large amount of parallelism that can be exploited with the loop detection mechanism we have presented in the previous section.

In the following subsections we propose a realistic thread control speculation technique and analyze its behavior.

3.1. Thread control speculation using dynamic loop detection

The thread control speculation that we propose in this paper answers the 3 issues described in the previous section in the following manner.

3.1.1 When speculation is performed?. Whenever a loop iteration starts in the non-speculative thread. Only the non-speculative thread can create speculative threads.

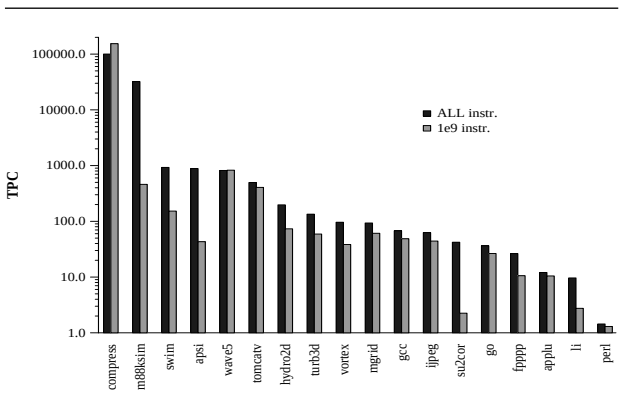

Figure 5: TPC for infinite TUs.

3.1.2 Which threads are speculated?. The answer to this question is the number of threads that are speculated and their identification.

The speculated threads, if any, are always consecutive iterations of the same loop that has initiated an iteration in the non-speculative TU. Note that when an iteration of a loop begins, that loop is the innermost, but it may become non-innermost if other loops are detected before the iteration of that loop finishes. To preserve the order among threads, the identifier of the TU assigned to each speculated thread is placed in the entry of the CLS associated to the loop.

With respect to the number of speculated threads we have considered 3 policies:

- IDLE. The number of speculated threads is equal to the number of idle TUs existing in that moment.
- STR. The number of speculated threads is based on the iteration count of the last execution plus the stride between the last two executions. If the stride is reliable (a two-bit saturating counter is used), the number of speculated threads is the minimum between the number of idle TUs and the number of predicted remaining iterations of the current execution. If the stride is not reliable but the number of iterations of the last execution is known, it is used the same policy but predicting that the iterations of the current execution will be the same as the last one. In case that neither the number of iterations nor the stride are known, any idle TUs is allocated to a further iteration of the same loop.
- STR(i). It is based on the last strategy but it adds a parameter *i*, which corresponds to the maximum number of non-speculated loops that can be nested into a loop that is being speculated. If this limit is exceeded, all speculative threads corresponding to the outermost loop are squashed. In this way, idle

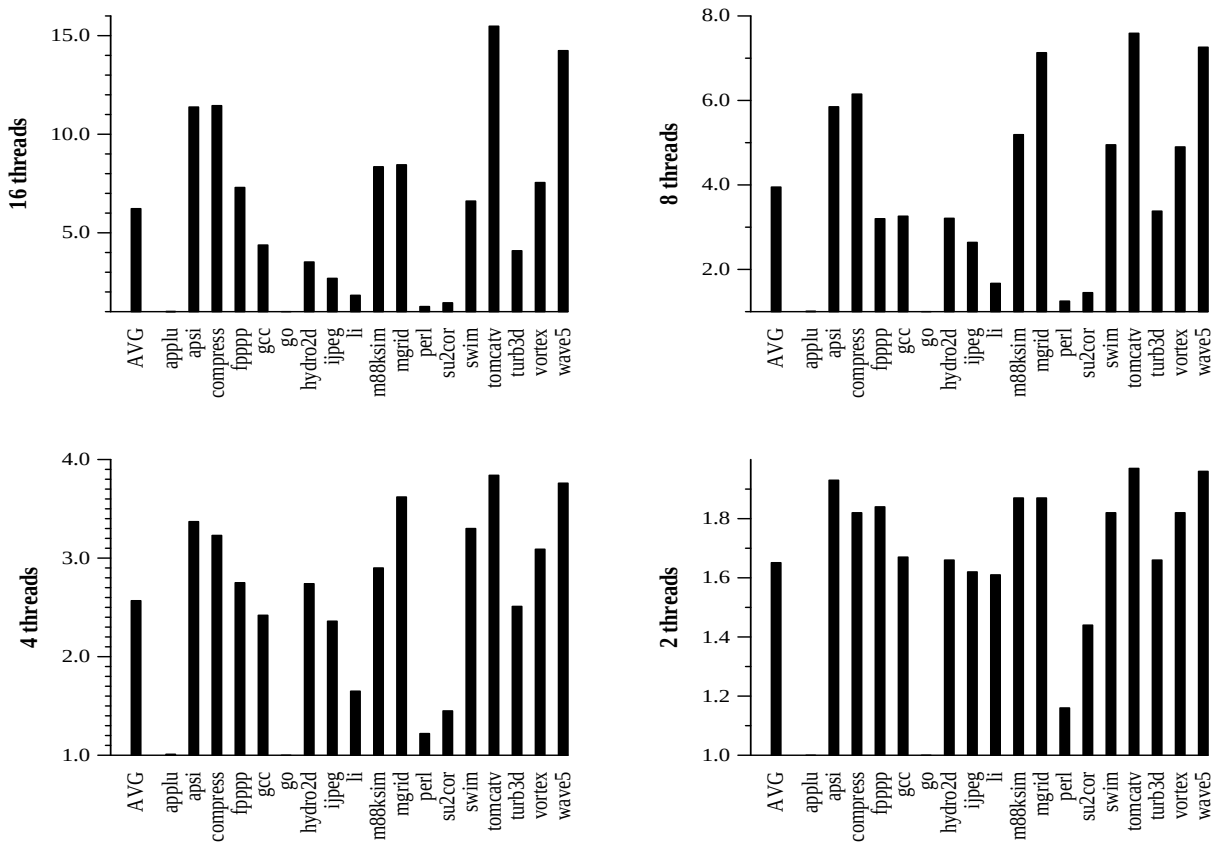

Figure 6: TPC in the SPEC95 suite for 2, 4, 8 and 16 TUs using the STR policy.

TUs can be used to speculate in inner loops.

An important issue of any parallel execution is load balancing. The proposed speculation approach allows for simultaneous speculation on several loops. In this case, the speculative threads are ordered from innermost to outermost (i.e., the non-speculative thread corresponds to an iteration of the innermost loop; the following one corresponds to the same loop or to an outer loop and so on). Since usually innermost loops are smaller than outermost ones, it will rarely happen that a thread is stalled at the end waiting for the termination of a previous thread that corresponds to an inner loop. Besides, we have measured that in average about 85% of the iterations of a loop follow the same control flow (see section 4). Thus, most of the iterations will have the same amount of instructions and it will rarely happen that a thread is stalled at the end waiting for the termination of a previous one corresponding to the same loop.

3.1.3 When verification is performed?. Verification is performed by the non-speculative thread when it starts a loop iteration or it finishes a loop execution.

When a loop iteration starts, it is checked if an iteration of the same loop had been speculated. In this case, the speculative thread associated to the first speculated iteration becomes the new non-speculative thread and the TU associated to the current non-speculative thread becomes idle. The new non-speculative thread updates all global data structures used for thread speculation (CLS, LIT and LET).

When a loop execution finishes, all speculative threads executing further non-existent iterations of the same loop are squashed. This corresponds to a control misspeculation.

3.2. Performance

The TPC (active and correctly speculated threads per cycle) obtained with these speculation policies has been computed for the execution of the first 10^9 instructions of all the programs in the SPEC95.

Figure 6 shows the TPC for the STR speculation policy. The number of TUs is 2, 4, 8 and 16. Data is shown for every program of the SPEC95. Note that the achieved TPC is considerable. For a small number of TUs, the proposed control speculation approach keeps them busy most of the time (the average TPC for 2 and 4 TUs is 1.65 and 2.6

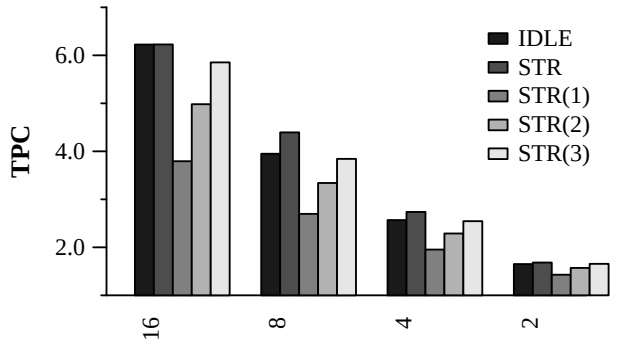

Figure 7: TPC in the SPEC95 suite for 2, 4, 8 and 16 TUs using IDLE, STR, STR(1), STR(2) and STR(3) policies.

respectively). As the number of TUs increases, their utilization decreases but it is still acceptable even for 16 TU. In this case, the average TPC is 6.2 but several programs achieve a TPC higher than 10. It is remarkable the high efficiency of the mechanism for *tomcatv* and *wave5*. For both programs, the maximum TPC is nearly achieved.

Moreover, figure 7 compares the different policies that have been described: IDLE, STR and STR(i), for *i* ranging from 1 to 3. The bars show the TPC averaged for all the programs of the SPEC95. The STR policy behaves slightly better than the IDLE policy. The STR(i) policy behaves worse than STR, which is due to the higher number of correct speculations that are squashed. Notice that in addition, both policies differ in the selection of loops that are speculated. STR(i) favors the speculation of inner loops; the lower *i* is, the more favored the inner loops are. In general, inner loops have smaller granularity, which may be beneficial when inter-thread data dependences are considered. Although this issue is beyond the scope of this paper, we consider the STR(i) policy, and concretely STR(3), to be very attractive when data dependences are taken into account.

Finally, table 2 shows some figures about the STR(3) speculation algorithm when the number of TUs is 4. The columns are the number of control speculations performed (#spec.); the average number of speculated threads per control speculation (#threads/spec.); the thread control speculation hit ratio (hit ratio); the average number of instructions since a thread is speculated until it is performed its verification (#instr. to verif.); and the TPC. Notice that the hit ratio is quite high for most programs, which confirms the accuracy of the control speculation approach.

	#spec.	#threads/ spec.	hit ratio (%)	#instr. to verif	TPC
applu	218661	2.62	54.51	2316	2.21
apsi	118637	2.91	90.48	2301	3.51
compress	2804450	2.69	100.00	91.94	3.23
fpppp	3417	1.67	86.92	191727	2.71
gcc	1206937	2.06	76.05	370	2.37
go	18427	2.09	71.17	69749	1.06
hydro2d	706635	2.99	99.43	433	2.52
ijpeg	150450	2.72	96.54	1608	2.36
li	1567433	1.71	69.16	353	1.75
m88ksim	1097194	2.77	97.32	292	2.78
mgrid	7900	2.80	97.50	36523	3.71
perl	3114338	2.33	60.34	35	1.17
su2cor	4906331	2.22	99.92	45	1.94
swim	61005	3.00	99.91	4455	3.48
tomcatv	111394	2.86	77.24	2363	3.85
turb3d	106237	2.99	99.18	2417	3.84
vortex	131024	2.12	90.25	2502	3.03
wave5	165950	2.60	99.95	1778	3.75

Table 2: Control speculation statistics.

4. Preliminary data speculation statistics

The focus of this paper is the proposal of a control speculation mechanism to generate threads from a sequential program. The management of inter-thread dependences is an ongoing work that is outlined in section 2.3. In this section, we present some preliminary statistics to give a flavor of the potential of such approach.

We have first identified for each loop the different control flows that its iterations can take. The sequence of instructions that make up an iteration with a particular control flow is called a path. We have then evaluated that the most frequent path of each loop accounts for 85% of all the iterations in the SPEC95.

For these iterations, we have measured if the contents of live-in registers and memory locations can be predicted based on the value computed in the last iteration of the loop plus its stride (that is, the difference between the last two consecutive iterations). Figure 8 depicts the data we have obtained. It contains the percentage of iterations covered by the most frequent path (*same path*); the percentage of correctly predicted live-in registers (*lr pred*); the percentage of correctly predicted live-in memory locations (*lm pred*); the percentage of iterations with all their live-in registers correctly predicted (*all lr*); the percentage of iterations with all their live-in memory locations correctly predicted (*all lm*); and the percentage of iterations with all

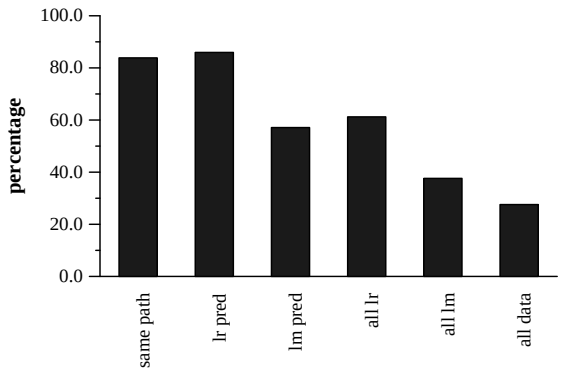

Figure 8: Data speculation statistics.

their live-in values correctly predicted (*all data*).

These figures have been obtained assuming that LIT and LET tables have enough capacity to store all the loops in the program. Notice the high potential of the mechanism to predict the live-in values. Current work is focused on developing a complete mechanism to execute sequential programs in the framework of a multithreaded architecture using the dynamic detection and speculation of loops that has been proposed in the paper.

5. Conclusions

We have presented a technique oriented to the automatic detection of loops and the dynamic computation of information that characterizes them. We have applied this mechanism to obtain speculative threads from a sequential program. This source of parallelism can be exploited by an architecture supporting multiple threads of control. The proposed mechanism is hardware-based and does not require any special feature in the ISA.

We have shown that the amount of thread-level parallelism that can be obtained from loops is very high for an unlimited resource machine. For a feasible configuration with 2, 4, 8 and 16 contexts, the proposed mechanism achieves a TPC (average number of active and correctly speculated threads per cycle) of 1.65, 2.6, 4 and 6.2, respectively.

Dealing with inter-thread data dependences is the follow-up of this work. Preliminary results show that a high percentage of the values that flow through such dependences can be predicted and thus, their corresponding synchronization can be avoided.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under grant CICYT TIC 429/95. The results presented in this paper have been obtained with the computing resources of the European Center for Parallelism of Barcelona (CEPBA).

References

[1] M.N. Dorojevets and V.G. Oklobdzija, “Multithreaded Decoupled Architecture”, *Int. J. of High Speed Computing*, 7(3), pp. 465-480, 1995.

[2] P.K. Dubey, K. O’Brien, K.M. O’Brien and C. Barton, “Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-Assisted Fine-Grained Multithreading”, in *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 109-121, 1995.

[3] M. Franklin and G.S. Sohi, “The Expandable Split Window Paradigm for Exploiting Fine Grain Parallelism”, in *Proc. of Int. Symp. on Computer Architecture*, pp. 58-67, 1992.

[4] J. González and A. González, “Speculative Execution via Address Prediction and Data Prefetching” in *Proc. of the ACM Int. Conf. on Supercomputing*, pp. 196-203, July 1997

[5] Q. Jacobson, S. Bennet, N. Sharma and J.E. Smith, “Control Flow Speculation in Multiscalar Processors”, in *Proc. of Int. Symp. on High-Performance Computer Architecture*, pp. 218-229, 1997.

[6] M. Kobayashi. “Dynamic Characteristics of Loops” in *IEEE Transactions on Computers*, C-33(2), pp. 125-132, 1984.

[7] M.S. Lam and R.P. Wilson, “Limits on Control Flow Parallelism”, in *Proc. of Int. Symp. on Computer Architecture*, pp. 46-57, 1992.

[8] J.E. Smith, “A Study of Branch Prediction Strategies”, in *Proc. of Int. Symp. on Computer Architecture*, pp. 135-148, 1981.

[9] G.S. Sohi, S.E. Breach and T.N. Vijaykumar, ”Multiscalar Processors”, in *Proc. of Int. Symp. on Computer Architecture*, pp. 414-425, 1995.

[10] A. Srivastava and A. Eustace, “ATOM: A System for Building Customized Program Analysis Tools”, in *Proc. of Conf. on Programming Languages Design and Implementation*, 1994.

[11] J-Y. Tsai and P-C. Yew, “The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation”, in *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 35-46, 1996.

[12] D.W. Wall, “Limits of Instruction-Level Parallelism”, in *Technical Report WRL 93/6*, Digital Western Research Laboratory, 1993.

[13] T. Yeh and Y.N. Patt, “A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History”, in *Proc. of Int. Symp. on Computer Architecture*, pp. 257-266, 1993.