Speculative Hedge: Regulating Compile-Time Speculation Against Profile Variations

Brian L. Deitrich Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
briand,hwu@crhc.uiuc.edu

Abstract

Path-oriented scheduling methods, such as trace scheduling and hyperblock scheduling, use speculation to extract instruction-level parallelism from control-intensive programs. These methods predict important execution paths in the current scheduling scope using execution profiling or frequency estimation. Aggressive speculation is then applied to the important execution paths, possibly at the cost of degraded performance along other paths. Therefore, the speed of the output code can be sensitive to the compiler's ability to accurately predict the important execution paths. Prior work in this area has utilized the speculative yield function by Fisher, coupled with dependence height, to distribute instruction priority among execution paths in the scheduling scope. While this technique provides more stability of performance by paying attention to the needs of all paths, it does not directly address the problem of mismatch between compile-time prediction and run-time behavior.

The work presented in this paper extends the speculative yield and dependence height heuristic to explicitly minimize the penalty suffered by other paths when instructions are speculated along a path. Since the execution time of a path is determined by the number of cycles spent between a path's entrance and exit in the scheduling scope, the heuristic attempts to eliminate unnecessary speculation that delays any path's exit. Such control of speculation makes the performance much less sensitive to the actual path taken at run time. The proposed method has a strong emphasis on achieving minimal delay to all exits. Thus the name, speculative hedge, is used. This paper presents the speculative hedge heuristic, and shows how it controls over-speculation in a superblock/hyperblock scheduler. The stability of output code performance in the presence of execution variation is demonstrated with six programs from the SPEC CINT92 benchmark suite.

1. Introduction

Path-oriented scheduling methods, such as trace scheduling [7] and superblock/hyperblock scheduling [10, 12], extract instruction-level parallelism from control-intensive programs by using speculation. Profile information or frequency estimation guides aggressive speculation, so that important execution paths can have their run time minimized. However, with limited execution resources, situations arise where one path will execute faster, only if another path gets delayed.

Fisher proposed the use of *speculative yield* to determine the profitability of speculating an instruction [8]. The speculative yield is an expected value function which is defined between basic blocks. It is the probability that an operation scheduled in basic block$_i$ produces useful work (meaning that its original basic block$_j$ executes when basic block$_i$ executes). Its use with dependence height has been shown to account for the needs of all paths during the scheduling process [2].

Speculative yield, coupled with dependence height, provides a good heuristic for path-oriented schedulers, but it does not address the problem of mismatches between compile-time prediction and run-time behavior. There is nothing inherent to speculative yield and dependence height that ensures that paths, which are shown by profile data to be unimportant, do not get delayed unnecessarily. This leads to execution-time slow-down when those paths are really executed at run time.

The *speculative hedge* heuristic, presented in this paper, attempts to ensure that no path gets delayed unnecessarily, even while performing aggressive speculation. Therefore, it *hedges* its reliance on the compile-time prediction data. It accomplishes this goal by accounting for different processor resources while scheduling, not just the common scheduling priority function of dependence height.

It should be noted that speculative hedge only solves part of the profile independence problem. Speculative hedge does not address the problem of determining what paths should be scheduled together. A poor selection of paths to schedule together will limit the amount of ILP that can be generated for a program, and cause a profile-dependence problem [5]. Speculative hedge limits the damage associated with a poor selection by ensuring that paths are not delayed unnecessarily. This property is especially important for any compiler which uses static analysis instead of profile information to determine which paths should be scheduled together [9].

The usefulness of the speculative hedge heuristic is demonstrated in a superblock/hyperblock scheduler. The remainder of the paper describes the heuristic, its implementation in the superblock/hyperblock scheduler, and the results obtained when the heuristic is applied to six SPEC CINT92 benchmarks. The paper is organized as follows: Section 2 provides background on terms used in the paper and related work. Section 3 provides the details of the speculative hedge heuristic. The performance results are reported in Section 4. A summary and a description of future work is given in Section 5.

2. Background

2.1. Late Times and Dependence Height

Dependence height is an important measure in determining an operation's scheduling priority. Many heuristics depend on it exclusively during scheduling. Speculative hedge uses dependence height as one component when determining scheduling priority. *Late times* are used by the speculative hedge heuristic to account for dependence height while scheduling.

A late time for an operation is the latest time that an operation can be scheduled without delaying an exit. An operation has a set of late times associated with it, one for each exit. Late times are calculated based on the latencies associated with the dependence graph used for scheduling. Figure 1 shows the late time values for a simple dependence graph. The late times are a 2-tuple in the figure. The first entry in the tuple is the operation's late time for exit 0, and the second entry is the operation's late time for exit 1.

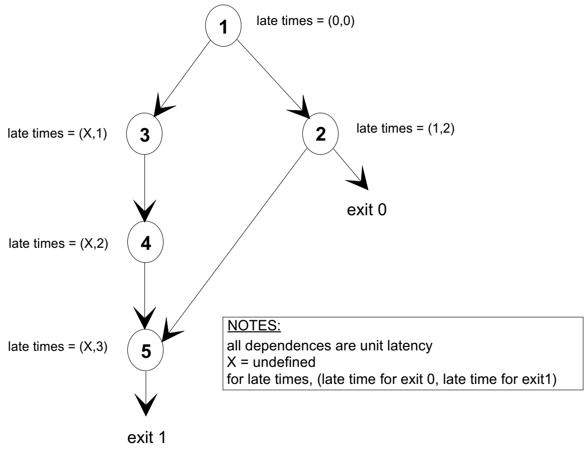

Figure 1. Dependence graph used for scheduling

Late times are computed via a bottom-up traversal of the dependence graph. The calculation is done once for every exit. An exit operation has its late time defined as the dependence height of the exit (which would have been determined with a top-down traversal of the dependence graph) minus one. In the example, operation 2's late time for exit 0 is 1, and operation 5's late time for exit 1 is 3. All other operations have their late times computed based on the exit late times of operations located later in the dependence graph. Operation$_x$'s late time for exit$_i$ (LT_x) is defined as:

For all dependence arcs exiting operation$_x$ and entering any operation$_y$, $LT_x = \min_{for\_all\_ops_y}(LT_y - lat_{xy})$[1]

LT_x = op$_x$_late_time_for_exit$_i$,

LT_y = op$_y$_late_time_for_exit$_i$,

lat_{xy} = latency_of_dep_arc_between_ops_x_and_y

If an operation does not reach an exit through a dependence chain, the operation does not contain a *valid late time* for that exit. Therefore, its late time for the exit is undefined.

Speculative hedge utilizes late times in the conventional manner, but allows an exit operation's late time for that exit to be defined by the most constraining resource height, instead of always using dependence height. In Figure 1, if the processor being scheduled is single-issue, the exit associated with operation 5 cannot be issued as quickly as the dependence height dictates. Since issue width is the limiting resource, operation 5's late time for exit 1 would actually be four (the issue width height minus one). Therefore, all the late times associated with exit 1 would be one greater than the values shown in the figure.

2.2. Previous Work and Other Heuristics

DEPENDENCE HEIGHT AND SPECULATIVE YIELD

Fisher suggests that the multiplication of dependence height by speculative yield is a good candidate for the scheduling priority function [8]. The appeal is that dependence height is commonly used as the priority function, and speculative yield allows it to take into account the probability of taken branches.

Bringmann utilized this concept in a list scheduler for superblocks [2]. In his method, a static heuristic utilizes the exit probabilities[2] and an operation's late times to generate the priority for that operation. The operations are then scheduled greedily, so that the highest priority operation that is available gets scheduled. The priority function used in this scheduler is:

$Priority_x = \sum_{ValLT_x}(Prob_i * (MaxLT + 1 - LT_x))$

$ValLT_x$ = for_all_valid_late_times$_i$_of_op$_x$

$Prob_i$ = probability_of_taking_exit$_i$

$MaxLT$ = max_late_time_in_dep_graph

LT_x = op$_x$_late_time_for_exit$_i$

[1] In a practical implementation, if all dependence arcs go in the forward direction relative to the original program order, the equation can be applied once for each operation by visiting the operations in the reverse program order.

[2] Since a superblock has only one entrance point, exit probabilities are equivalent to Fisher's speculative yield values

To illustrate how the priority values are obtained, reference Figure 1. Assume exit 0 is taken 25% of the time and exit 1 is taken the other 75% of the time. The priority values obtained are:
$\text{operation}_1 = 0.25 * (3 + 1 - 0) + 0.75 * (3 + 1 - 0) = 4$, $\text{operation}_2 = 2.25$, $\text{operation}_3 = 2.25$, $\text{operation}_4 = 1.5$, and $\text{operation}_5 = 0.75$.

Using only dependence height to guide the scheduler can delay exits unnecessarily though. This is shown in Figure 2. Assuming the latencies marked in the figure, the second branch, operation 7, can be retired in the third cycle when one only considers dependence height. If the profile information shows that exit 1 is always taken and dependence height guides the scheduling priority function, operations 1 and 2 would be scheduled in the first cycle. The earliest time exit 1 can actually be retired is in the fourth cycle because there are eight operations that must be issued on this two-issue machine. Since the issue width of the processor is the limiting factor, exit 0 should be retired right away (regardless of the profile information). This allows exit 0 to be retired immediately, and still allows exit 1 to be retired as soon as possible. This case illustrates the main concept behind speculative hedge—account for processor resources so exits are not delayed unnecessarily.

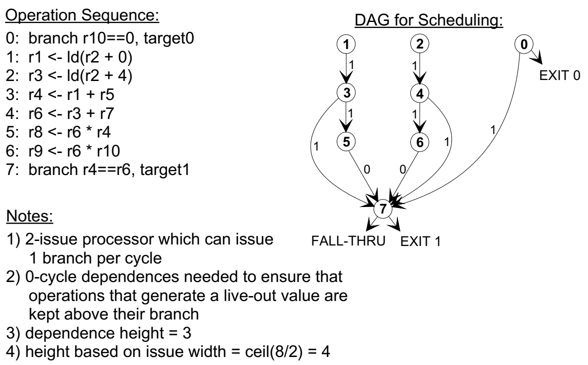

Figure 2. Example of dependence height not being a problem.

CRITICAL PATH

Critical path is a profile-independent scheduling heuristic that has been used in many types of schedulers [13, 6, 11]. Its application to superblock scheduling consists of using the dependence height associated with the last exit in a superblock.

The *dependence height and speculative yield* priority function creates identical schedules to *critical path* if the profile information shows that the last exit has a probability of one, and all other exits have a probability of zero. This scheduling heuristic suffers from over-speculation along the longest path of control, and the potential for delaying earlier side exits unnecessarily. Speculative hedge can use the same profile-independent assumption that the last exit is always taken, and create better schedules through its use of resource information, along with the dependence information.

SUCCESSIVE RETIREMENT

Successive retirement is another profile-independent scheduling heuristic which attempts to retire each exit in-order, as early as possible [4, 16]. This heuristic minimizes speculation, so that it only speculates when there are no non-speculative instructions available.

This scheduling method works well for narrow-issue processors, where speculation is not very important. However, it can potentially lose performance by not speculating enough for wide-issue processors. When a narrow-issue processor is scheduled for, successive retirement and speculative hedge create similar schedules. This results from speculative hedge's ability to understand when the processor's issue width is the limiting factor to the retirement of exits.

3. Speculative Hedge Heuristic

3.1. High-Level Overview

Speculative hedge takes both dependence height and resource constraints into account while scheduling. Many previous methods have considered only dependence height as the limiting factor. As shown previously, taking into account only dependence height can delay seemingly unimportant exits.

To account for the resources accurately, speculative hedge uses a dynamic-priority scheme. The dynamic-priority calculation is needed because exits can be constrained by different resources at various times during the scheduling process. The type of resource that constrains an exit's retirement has a direct effect on scheduling decisions. The compilation-time implications of this approach are addressed in Section 4.

In order to determine the best operation to schedule next, speculative hedge uses several priority values. The priority values used are:

- Helped weight - The sum of the taken probabilities for all exits having a *critical need* met by an operation.
- Helped count - The total number of exits having a *critical need* met by an operation.
- Minimum late time difference - The minimum difference between the current cycle time being scheduled and any of an operation's late times.
- Original program order - A unique number which is based on the original program location of an operation.

The first two priority values are utilized whenever the scheduling of an operation satisfies a *critical need* for an exit. A critical need can be dependence height, issue width, or any other restricted processor resource, which must be dealt with in order for an exit to be retired by its *retirement goal*. An exit's retirement goal is the earliest time that an exit can be retired based on dependence height or any resource constraint. An operation satisfies a critical need by having the properties necessary to retire an exit quickly. One example of satisfying a critical need occurs when an operation has a late time equal to the current cycle being scheduled and an exit is dependence height limited. Another example happens when an operation is an integer add operation and integer ALU issue width is the limiting factor for an exit. If an operation can satisfy a critical need, it *helps* an exit by allowing it to retire quickly.

Helped weight is the first criterion used to determine which operation should get scheduled. Helped weight's value is defined as the sum of all taken-exit profile weights that can be retired as soon as possible, if the current operation gets scheduled immediately. For example, if a superblock contains three exits (with each exit taken 25%, 40%, and 35% of the time respectively), an operation which helps the first two exits retire would have a helped weight of 0.65. An operation with the highest helped weight value gets scheduled next since it helps the most probable exits.

Of the four priority values, helped weight is the only one that depends on profile information. The other three priority values allow the speculative hedge heuristic to be less sensitive to profile variations. They allow fair decisions to be made for all exits, regardless of the compile-time predicted behavior.

The *helped count* priority value is used when operations share the same helped weight. Helped count is the number of exits having a critical need met by an operation. This criterion gives zero weight exits priority. If the weight of all the exits helped between two operations is equal, the operation helping the most exits should get scheduled first.

The helped weight and helped count priority values work together to prevent exits from being delayed unnecessarily. They accomplish this with the accurate accounting of resources. For example, if an important exit's retirement is issue-width limited, all operations located prior to the exit will meet the critical criteria for the exit. Therefore, operations helping an earlier, less important exit are critical for the important exit too. These operations would be chosen for scheduling because they will include both exits in their helped weight and helped count values. Even if the earlier exit was shown to never be taken based on the profile information, helped count would ensure the exit does not get delayed.

On the other hand, if dependence height is the limiting factor in an important exit's retirement, speculative hedge will work similar to the *dependence height and speculative yield* heuristic. Therefore, it will delay certain exits in favor of an exit that the profile information shows to be important. Speculative hedge does not keep exits from being delayed because of speculation. It only tries to minimize the number of exits and the penalty for exits that are delayed unnecessarily.

Another important aspect of speculative hedge is that it only looks at the exits that are helped by scheduling a particular operation. Speculative hedge's priority values do not directly deal with the condition where the scheduling of an operation might delay another exit. This is indirectly handled through the priority values for other operations. In order for the scheduling of operation_x to delay an exit_i, there must exist another operation_y, whose scheduling would allow exit_i to not be delayed. Since operation_y helps exit_i, it gets exit_i's contribution for helped weight and helped count. Therefore, the trade-off that needs to be made between the delaying of an exit_i and another exit is done via the comparison of operations x and y's helped weight and helped count values.

If the helped weight and helped count values are equal, the next criterion used is *minimum late time difference*. As defined earlier, each operation has a set of late times, one for each exit. An operation's late time difference for an exit is defined as the difference between the current cycle being scheduled and the operation's late time for that exit. An operation's minimum late time difference is the minimum of all the exits' late time differences.

This criterion is important because it helps anticipate which operations are about to become critical. Its biggest benefit comes when the helped weight and helped count values are both zero. This condition occurs after several operations have already been scheduled for the current cycle, all the scheduled operations have met all the exits' critical needs, and there is still at least one open slot in the current cycle.

The final criterion, original program order, is used when all the previous criteria have tied. It is used so that an operation located earlier in the program order gets scheduled first. This keeps unnecessary speculation from occurring and allows a deterministic schedule.

The priority values are used by speculative hedge to determine which operation should be scheduled next. The speculative hedge heuristic computes the priority values in the following way. First, the retirement goals and critical needs for every exit that has not been retired are determined. Then, all the late times for unscheduled operations are computed using the exit retirement goals as the defined late times for their corresponding exit operations. Finally, the priority values for all operations available for scheduling are computed, and the operation with the highest priority gets scheduled.

The remainder of this section describes the details needed to utilize the speculative hedge heuristic effectively. First, the determination of exit retirement goals and critical needs is presented. Second, the application of the priority value calculation is shown. Finally, some potential problems with speculative hedge are discussed.

3.2. Exit Retirement Goals and Critical Needs

The key components of speculative hedge are the estimation on the number of cycles that remain before an exit can be retired, and the determination of which resources must be utilized in order to not delay the exit. This information is used directly by the dynamic priority function to determine if an exit has a critical need, and whether the scheduling of an operation will enable that need to be met. The scheduling of an operation meeting a critical need either shortens the unscheduled dependence height or consumes a resource so that resource need in the future is alleviated. As with the priority calculation, this information is recomputed dynamically during the scheduling process.

In speculative hedge, the estimation of when an exit can be retired is optimistic. It is a lower bound on how quickly an exit can be retired [14]. An estimate may change during the course of scheduling because conflicting needs between different exits force some exits to delay. It is the dynamic priority function's responsibility to effectively deal with the trade-offs that must be made during the scheduling process.

There are three main needs that are always considered, independent of the processor that is targeted. They are dependence height, the branch unit, and issue width. Other processor resources that are restricted, such as memory operation issue width or special decoder requirements, are considered when they apply.

For each need, a cycle estimate is made which determines when an exit can be retired, based exclusively on that need. The maximum of all the cycle estimates for each need yields the retirement goal for an exit. Any need whose cycle estimate equals the exit retirement goal is considered critical.

The cycle estimate based on dependence height is a straight-forward computation which determines the dependence height that must be honored before an exit can be retired. It must account for unscheduled operations and latencies for already scheduled operations that have not been fully satisfied.

The cycle estimate based on the branch unit is a function of the number of branches that can be issued in the same cycle, and the retirement goals for previous exits sharing a common control path. Branches guarding a common control path cannot be reordered. Therefore, an exit with a branch must have a cycle estimate which is at least the same as the retirement goal for a preceding branch along a common control path. If an exit is a fall-thru, it must have a retirement goal that is at least the same as any preceding exit's retirement goal.

If the number of preceding branches having the same retirement goal is equal to the number of branches that can be issued in any cycle, the current exit's cycle estimate is the preceding retirement exit goal plus one. This happens because there are not enough branch units to retire all of those exits simultaneously. In the one branch-per-cycle case, the cycle estimate based on the branch unit is always one more than any preceding exit retirement goal.

Speculative hedge's cycle estimate based on issue width (ce_{iw}) could be:

$ce_{iw} = \lceil num\_ops\_to\_sched\_before\_exit / issue\_width \rceil$

This simple calculation does not detect enough issue-width problems though. Many times, an exit may have an issue-width problem early in its dependence chain, but contain few operations later in its dependence chain. This is illustrated in Figure 3. In this example, a two-issue machine needs to execute three loads which, based on dependence height, must be executed in cycle zero in order to not delay the exit. Since it is impossible to issue them all simultaneously, the exit is really issue width limited at the beginning of its dependence chain.

To deal with this effectively, speculative hedge detects these cases, and keeps track of the late times that are included in any issue-width problem. This allows the speculative hedge heuristic to ensure that the issue-width problem is taken care of properly. For an operation to help solve an issue-width problem, it must have a late time for the exit that is less than or equal to the late time where the issue-width problem ends.

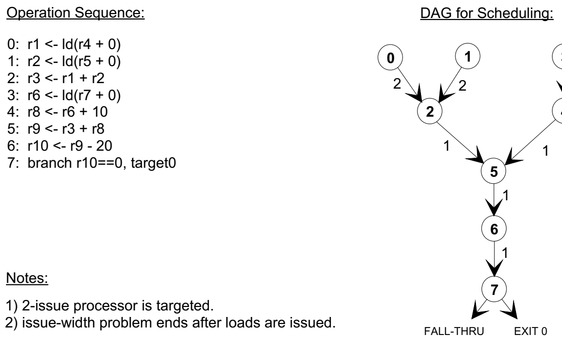

Figure 3. Example of an issue-width problem located at the beginning of a dependence chain.

The cycle estimates for other restricted resources, like a limited number of integer ALU operations that can be issued in any cycle, can be handled similarly to the general issue-width problem just discussed.

3.3. Priority Calculation

Now that the exit retirement goals and critical needs have been determined, the dynamic priority calculation is performed. This dynamic scheme does require more computations than a static priority scheme, but is needed in order to account for all the changing resource requirements. The compilation-time implications of this method are discussed in the Section 4.

The priority calculation is performed for every operation that can be scheduled by evaluating each exit, and determining whether the operation meets a critical need for the exit. If the scheduling of an operation meets a critical need, the operation helps the exit retire quickly. The high-level view of the priority calculation is shown in Figure 4.

If an exit_i has dependence height as its critical need, an operation_x meets that need if it has a late_time_i which is equal to the current cycle being scheduled. If operation_x meets the critical need, operation_x's helped weight and helped count get incremented for exit_i. A check is also made to determine if the meeting of a critical need for the current exit_i will help future exits meet a critical need.

The check is needed to accurately account for branch-to-branch dependence height. A future exit_j is helped if the branch unit is a critical need for it, the current exit_i precedes it, it shares a common control path with the branch for exit_i, and the delaying of the branch for exit_i would cause the future exit_j to also be delayed. The determination of whether delaying exit_i will also delay a future exit_j is not trivial. Figure 5 helps to illustrate the different cases that are handled by speculative hedge.

```
for (each unscheduled, available oper_x) {
  helped_weight_x = helped_count_x = 0
  for (each exit_i) helped_found_i = FALSE
  for (each exit_i where oper_x has a valid late time_i) {
    if (dependence_critical_for_i &&
      (oper_x's late time_i == current_cycle)) {
        helped_found_i = TRUE
        account_for_future_branches
    }
    if (issue_width_critical_for_i &&
      oper_x's late time is located before issue width
      problem ends) {
        helped_found_i = TRUE
        account_for_future_branches
    }
    /* the following is an example of what needs to be
        done for restricted resources */
    if (integer_alu_issue_width_critical_for_i &&
      oper_x is an integer alu operation &&
      oper_x's late time is located before integer alu
      issue width problem ends) {
        helped_found_i = TRUE
        account_for_future_branches
    }
  }
  for (each exit_i)
    if (helped_found_i) {
      helped_weight_x += profile_weight_i
      helped_count_x++
    }
  determine_min_late_time_diff_for_oper_x
}
```

Figure 4. Algorithm for dynamic priority calculation.

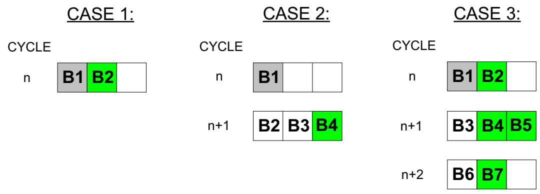

Figure 5. Cases for account_for_future_branches.

For each case, branch B1 is associated with exit$_i$. It is the exit that has a critical need met by operation$_x$. A determination is made to see if any future exits$_j$ will also be helped by operation$_x$. All branches shown are assumed to share a control path, and three branches can be issued in one cycle.

Case 1 is a basic example. In it, branch B2 is located after B1, and in the same cycle as B1. Therefore, if B1 gets delayed, B2 must be delayed too. B2's exit gets added to operation$_x$'s helped weight and helped count.

Case 2 is another basic example. Here, B1 is located by itself in a cycle, but the next cycle contains three branches. If B1 gets delayed, too many operations end up being located in the same cycle. So, B4 must be delayed also. B4's exit gets added to operation$_x$'s helped weight and helped count.

Case 3 is a combination of both cases 1 and 2. Here, the delay of B1, causes B2, B4, B5, and B7 to be delayed. Therefore, each one of them get their corresponding exit added into operation$_x$'s helped weight and helped count. In this cascading fashion, an operation$_x$ can help many exits just by meeting one exit's critical need.

The multiple branches-per-cycle problem is actually an important special case of a more general phenomenon. When predication is not used, branches have zero-cycle control dependences linking them. A similar case can also arise for other instructions. An example is a sequence of integer ALU operations that are linked by zero-cycle anti-dependences. If only one integer ALU operation can be issued in a single cycle, all the zero cycle anti-dependences can be changed to one cycle dependences in order to account for resource height. The fact making branches the most interesting case is that branches, when predication is not used, are always connected by these control dependences; no other instruction type has a zero-cycle dependence always linking them.

Operation$_x$ meets an exit's issue-width critical need, if the exit has issue width as a critical need, and operation$_x$'s late time is less than the late time associated with the issue width problem. The late time check is needed to ensure that the issue-width problem is really being addressed. As discussed in the previous section, an issue-width problem can be isolated to the beginning of an exit's dependence chain.

Finally, the checks needed for a restricted resource are also shown in Figure 4. They are identical to the issue width critical comparisons, with an extra check to ensure that operation$_x$ is the correct type of instruction. This same check can be replicated for any type of restricted resource that the scheduler should account for.

3.4. Potential Problems

As with any scheduling heuristic, there are some potential areas of concern with speculative hedge. Scheduling is an NP-complete problem, and obtaining an optimal solution for all cases is not feasible.

One problem area of speculative hedge has to do with the trade-offs made while selecting an operation to schedule. An operation may be chosen because it helps an important exit$_i$, but scheduling it may delay a less important exit$_j$. The problem arises, when a subsequent trade-off may mean that exit$_i$ gets delayed in order to help an even more important exit$_k$. Delaying exit$_j$, in hindsight, was unnecessary. Because the heuristic only looks at the situation for the current cycle, it may not always make the best decisions for the entire scheduling region as a whole.

This problem, while seen in practice, does not appear frequently enough to be a major issue. Also, it does not hurt the most frequently executed exits, and it causes only a short, unnecessary delay for the exits that it effects.

There is another problem related to scheduling trade-offs. This can happen when an important exit has dependence height as its critical need, and in the process of satisfying the dependence height need, it uses all the resources for a particular instruction type, like an integer ALU instruction. In addition, this situation lasts for several cycles. A less important exit can be retired immediately if it were allowed to issue an instruction whose type is being used up by the more important exit, but speculative hedge chooses to help the more important exit. When the problem is not solved for several cycles, the less important exit gets delayed several cycles.

This is an issue because a net loss may result. The cost of delaying the less important exit several cycles may not be offset by allowing the more important exit to be retired one cycle early. Speculative hedge only looks at the costs involved for the current cycle, and does not look at the costs that are involved in the future.

A final problem deals with speculative hedge's desire to not consciously schedule for an exit's retirement until it is absolutely necessary to do so. Speculative hedge will delay an operation on a path shown by profile information to be taken 100% of the time for an operation on a 0% path, if it determines that scheduling for the 0% path will not delay the 100% path. This works fine as long as the speculative hedge heuristic makes no misjudgments concerning the 100% path. If a misjudgment is made, the 100% path may get delayed unnecessarily.

This problem was seen in several places during the evolution of the speculative hedge heuristic. As the heuristic was improved, the problem became less pronounced. The main mechanism in the heuristic that helped alleviate the problem was the *determine_min_late_time_diff_for_oper* function. This function identifies operations that are most likely to be needed next, and helps ward off the potential problem.

4. Experimental Analysis

In this section, the effectiveness of the speculative hedge heuristic in minimizing the unnecessary delaying of exits is analyzed for a set of SPEC CINT92 benchmarks. First, speculative hedge's performance is compared against other scheduling heuristics on real input cases for a variety of machine configurations and compile-time assumptions. Then, an in-depth analysis is done to show exactly how well the speculative hedge heuristic controls unnecessary speculation for an exit that does not appear to be important based on the compile-time predicted data. Finally, the compilation-time implications of speculative hedge's dynamic priority scheme are addressed.

4.1. Methodology

The speculative hedge, dependence height and speculative yield, and successive retirement heuristics have been implemented in the IMPACT compiler [3]. This compiler inlined, coalesced into superblocks, and fully ILP optimized the six SPEC CINT92 benchmarks used in this testing. The benchmarks used are *espresso*, *li*, *eqntott*, *compress*, *sc*, and *cc1*.

To show the effectiveness of the speculative hedge heuristic for various issue width and functional unit constraints, four machine models are utilized. They are:

- 4_MIX – A 4-issue processor that contains 2-memory ports, 2-integer ALUs, 1-floating point ALU, and 1-branch unit. This machine model is similar to many of today's processors.
- 4_1BR – A fully-uniform 4-issue processor that contains only one branch unit. This machine model allows more scheduling freedom than 4_MIX, and shows how the scheduling heuristics perform when the functional units are not constraining resources.
- 4_2BR – A fully-uniform 4-issue processor that contains two branch units. This machine model forces the condition where branch resource height is tough to account for.
- SINGLE – A single-issue processor. This machine model forces issue width to be a major issue and speculation to be minimized in order to obtain good performance.

Table 1. Instruction latencies.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide(SGL)	8
branch	1 / 1 slot	FP divide(DBL)	15

Each machine model's latencies match those of the HP PA-RISC PA7100 processor, and the latencies are shown in Table 1. An enhanced version of the HP PA-RISC processor instruction set is used with a non-trapping version of instructions added, so speculative instructions cannot cause program termination. This allows the general speculation model to be utilized by the scheduler [3]. This model gives the scheduler the freedom and choices during scheduling to help differentiate the performance of the heuristics.

The analysis focuses on prepass scheduling, and the effect that the compile-time predicted behavior has on the quality of the schedules produced. The effects of register allocation, cache misses, branch mispredictions, etc. are factored out, so that a fair comparison can be made between the different scheduling heuristics. This means that the execution time of programs is determined statically, by knowing the schedule time for a particular exit and the number of times that the profile information shows that the exit is taken.

4.2. Results

This section compares the schedules obtained for the different benchmarks. The first set of results analyze the schedules obtained by the speculative hedge, dependence height and speculative yield, and successive retirement heuristics for the four different machine models and different compile-time profile assumptions. The different schedules are compared assuming that the actual program input matches the exact input used to generate one of the compile-time profile assumptions. The second set of results analyze the effectiveness of the speculative hedge and dependence height and speculative yield heuristics to schedule the first exit when they both assume that the last exit is always taken. Finally, the time needed to schedule with the speculative hedge heuristic is compared against the time needed to schedule with dependence height and speculative yield.

Evaluation for Real Execution of Programs. In this test, the effectiveness of the speculative hedge heuristic in controlling unnecessary speculation is analyzed. This is done by scheduling each benchmark on four different machine models with the different scheduling heuristics. In addition, the two profile-dependent heuristics, speculative hedge (SH) and dependence height and speculative yield (DHASY), are given three different compile-time profile assumptions. The different assumptions will be used to show that SH is less sensitive to profile variations than DHASY.

The three different profile-time assumptions are *REAL* (the profile data matches the program behavior for the real data that is input), *ALL1* (the profile data shows that all exits are equally likely), and *LAST1* (the profile data shows that only the last exit in the superblock is ever taken). Note that the critical path scheduling heuristic is the same as scheduling DHASY with the LAST1 profile assumption.

The results for the four machine models after scheduling with the different scheduling heuristics are shown in Table 2. The cycle count for SH-REAL assumes the same profile input used to generate the SH-REAL schedule is used in the actual run. The *% Diff* columns represent the difference the column's scheduling heuristic cycle count had to the SH-REAL's cycle count for the same benchmark and input data (a negative value means the column's heuristic required more cycles to execute).

SH is less reliant on the compile-time profile assumption than DHASY. The largest difference between SH-REAL and SH-ALL1/SH-LAST1 for any machine model and benchmark is 2%. This is contrasted with DHASY-REAL and DHASY-LAST1 (which is critical path) which vary by over 5% for at least one benchmark in every machine model, and vary by over 10% for at least one benchmark for the 4_2BR and SINGLE machine models.

In addition, SH-ALL1 and SH-LAST1, which were scheduled with compile-time profile assumptions that differed from the run-time execution, outperformed DHASY-REAL for most of the test cases. SH-ALL1 and SH-LAST1 executed more cycles than DHASY-REAL for only one benchmark in each of the 4_MIX, 4_1BR, and 4_2BR machine models. SH-ALL1 and SH-LAST1 performed better because they were able to account for resources while scheduling, and minimize the effect of exits that were delayed unnecessarily. Even though SH-ALL1 and SH-LAST1 may not have helped some of the execution paths that were important at run-time (because they did not have that information), they were able to minimize unnecessary exit delays. On the other hand, DHASY-REAL did delay some less important exits unnecessarily, and this resulted in its schedule taking a few more cycles to execute than the schedules for SH-ALL1 and SH-LAST1.

To understand how speculative hedge performs for special cases, the SINGLE and 4_2BR machine models must be examined. The SINGLE machine model case shows how well speculative hedge performs when issue width is a severe problem. Speculative hedge generates schedules that are as good as, or better than successive retirement. This highlights speculative hedge's ability to control speculation, so exits are not delayed unnecessarily.

In the 2_BR machine model case, speculative hedge's method of accounting for the branch resource height allows it to be insensitive to the accounting problems for branch-to-branch dependences that effect DHASY. The zero cycle branch-to-branch dependence assumption made by DHASY leads to its profile sensitivity problems. For this machine model, DHASY-LAST1 (critical path) varies an average of over 5% when compared to DHASY-REAL, while SH-LAST1 only varies an average of 0.4% when compared to SH-REAL.

Evaluation of the unnecessary delaying of an exit. The results shown so far have been for real execution cases. These results can be affected by a minority of the superblocks present in the benchmarks. This makes it difficult to show how large a problem the dependence on profile information can be. To illustrate the potential problem, speculative hedge and DHASY are scheduled assuming that the last exit is always taken. Then, a comparison is made between their resulting schedules showing how well they handle the first exit for all the superblocks in the six benchmarks. The first exit was chosen because it has the best chance of being delayed unnecessarily by over-speculation.

The results are shown in Figure 6, and they were obtained for the 4_MIX machine model. The cycle difference subtracts the scheduled cycle for SH-LAST1's first exit to the cycle obtained for DHASY-LAST1's first exit. The zero cycle's column represents instances where SH-LAST1 and DHASY-LAST1 scheduled the first exit in the same cycle, and the cycle was later than could have been obtained if the first exit was scheduled to retire as quickly as possible. As shown in the figure, DHASY does a worse job in retiring the first exit of the superblocks for all the benchmarks.

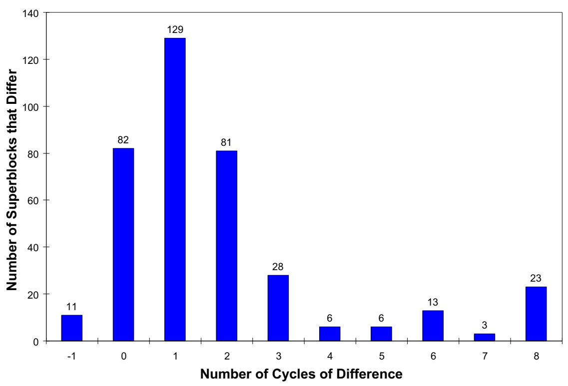

Figure 6. Cycle differences between SH-LAST1 and DHASY-LAST1 for the first exit.

Table 2. Results when input data matches input used for REAL profiling.

Machine Model	Benchmark	SH - REAL # of cycles	SH - ALL1 % diff	SH - LAST1 % diff	DHASY - REAL % diff	DHASY - ALL1 % diff	Crit. Path[a] % diff	Succ. Ret. % diff
4_MIX	espresso	152405400	-0.5	-0.9	-0.2	-0.6	-2.1	-0.7
	li	14486519	0.0	0.0	0.0	-0.7	-0.3	-0.4
	eqntott	423624658	0.0	0.0	0.0	0.0	0.0	0.0
	compress	25767022	-0.1	-0.1	-0.6	-3.1	-8.2	-2.6
	sc	38204165	0.0	0.0	-0.5	-0.9	-0.3	-3.7
	cc1	49349343	-0.2	-0.1	-0.3	-0.7	-0.2	-1.1
4_1BR	espresso	146674421	-0.1	0.0	-1.4	-1.2	-1.0	-1.6
	li	13702023	0.0	0.0	-0.8	-1.3	-0.2	-1.1
	eqntott	401064104	0.0	0.0	0.0	0.0	0.0	0.0
	compress	24174638	-0.5	-0.5	-0.4	-3.4	-6.3	-2.9
	sc	36938586	0.0	0.0	-0.1	-0.3	0.0	-2.5
	cc1	47497736	0.0	0.0	-0.1	-0.2	-0.1	-0.7
4_2BR[b]	espresso	129275735	-0.1	0.0	-1.0	-1.9	-7.2	-2.3
	li	12389729	-0.7	-0.7	0.0	-2.2	-1.3	-2.0
	eqntott	313622471	-0.4	-0.4	-1.0	-4.0	-6.4	-0.9
	compress	22011735	-0.3	-0.3	-0.6	-7.1	-11.4	-2.7
	sc	30904853	-1.2	-1.1	-1.4	-2.0	-5.8	-4.6
	cc1	41450987	0.0	0.0	-0.4	-0.6	-3.7	-1.3
SINGLE	espresso	330192736	-0.2	-0.2	-1.6	-3.1	-11.2	-0.2
	li	27242181	-0.6	-0.6	-0.3	-1.9	-4.4	-1.9
	eqntott	826767804	0.0	0.0	0.0	-0.1	-0.1	-0.1
	compress	56980582	-2.0	-2.0	-0.7	-6.3	-15.6	+0.1
	sc	84079061	-0.5	-0.4	+0.2	-0.8	-1.0	-0.7
	cc1	99468982	-0.3	-0.2	-0.1	-1.4	-3.1	-0.4

[a] Critical path is the same as DHASY-LAST1.
[b] All branch-to-branch dependences for DHASY are assumed to be 0.

An evaluation of speculative hedge's and DHASY's first exit schedule times to the earliest time that the first exit could be retired (which would be obtained if the profile information showed that it was the only exit ever taken) provides more insight into this issue. It confirms that not only does DHASY delay more exits unnecessarily, it also delays them longer than speculative hedge. An examination of the detailed data shows, that compared to the case where the first exit was retired as soon as possible, speculative hedge delayed 125 first exits for an average of 1.3 cycles, while DHASY delayed 379 first exits for an average of 2.4 cycles.

Evaluation of compilation time. The compilation time for speculative hedge must be addressed because of its use of a dynamic priority calculation. The priority calculation is performed on every operation that can be scheduled (which means there are no resource conflicts and all the operation's incoming dependences have been satisfied), and gets recomputed before any operation is scheduled. In addition, the priority calculation's run time is proportional to the number of exits that have yet to be retired. This leads to a worst-case run time for speculative hedge of $O(n^2m)$, where n = number of operations and m = number of exits. Measurements, during prepass scheduling of the SPEC CINT92 benchmarks, showed that speculative hedge had to compute priorities for an average of 2.8 operations before an operation got scheduled. This leads to a run-time of $2.8nm$ in practice. The worst-case run time for DHASY's static priority calculation is $O(nm)$.

The speculative hedge heuristic was implemented in the IMPACT compiler. Timing measurements were made during prepass scheduling. Besides the calculation of operation priorities and the actual scheduling, the only other computationally-intensive functionality performed was the data-flow analysis required to compute the live-out variables. IMPACT uses a standard, iterative data-flow algorithm for the analysis which has a worst-case run time of $O(b^2)$, where b = number of basic blocks [1]. For the benchmarks used in this paper, prepass scheduling took 26% to 49% longer for speculative hedge than it did for DHASY.

5. Conclusion

5.1. Summary

This paper described a new path-oriented, scheduling heuristic, *speculative hedge*, which attempts to minimize speculation so paths of execution do not get delayed unnecessarily. This ensures that speculative hedge is much less sensitive to variations between compile-time predicted behavior and actual run-time behavior. This is done by accounting for dependence height and processor resources while scheduling. Previous scheduling heuristics that only accounted for dependence height while scheduling would delay exits unnecessarily, and perform poorly at run time when the program executed differently than expected. The speculative hedge heuristic was implemented in a superblock/hyperblock scheduler to illustrate its effectiveness.

The paper investigated the performance of speculative hedge for six programs from the SPEC CINT92 benchmark suite for four different machine models. Speculative hedge was scheduled with three different sets of compile-time predicted behaviors, and the largest variation for any combination of machine model and compile-time predicted behavior was 2%. In contrast, *dependence height and speculative yield*, using the same sets of compile-time predicted behavior, had variations of over 5% for at least one benchmark in every machine model, with three cases having variations of over 10%. In addition, an analysis of how the first exits of superblocks can be delayed for different profile inputs showed that *dependence height and speculative yield* delayed some superblock first exits by up to eight cycles more than speculative hedge.

5.2. Future Work

Future work includes fixing the speculative hedge heuristic so it can account for multiple, independent paths of control. This problem must be fixed so fully-resolved predicates [15] can be dealt with effectively. In addition, some of the potential problem areas discussed earlier may be tackled if new applications show that they are more of an issue than they were in SPEC CINT92.

Another area of future research will be a coupling of speculative hedge and register allocation. Currently, speculative hedge does not account for register pressure while scheduling. Allowing speculative hedge to control speculation and make intelligent decisions when register pressure is an issue should prove useful for a number of different situations.

A final area that needs to be addressed is making the path selection process done by the compiler less dependent on profile information. Speculative hedge makes the scheduling of paths within a scheduling scope less dependent on profile information, but it can only minimize the penalty incurred when path selection does a poor job. By lessening the profile sensitivity of path selection and utilizing speculative hedge, a complete solution to the profile sensitivity problem of scheduling can become reality.

Acknowledgments

The authors would like to thank John Gyllenhaal for his suggestions regarding the presentation of results. The authors would also like to thank all the members of the IMPACT research group and the anonymous referees whose comments and suggestions helped to improve the quality of this paper.

This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Intel Corporation, Advanced Micro Devices, Hewlett-Packard, SUN Microsystems, NCR, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] R. A. Bringmann. *Compiler-Controlled Speculation.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[3] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 266–275, May 1991.

[4] C. Chekuri, R. Motwani, R. Johnson, B. Natarajan, B. R. Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with applications to super blocks. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996.

[5] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox. Hardware-based profiling: An effective technique for profile-driven optimization. *International Journal of Parallel Programming*, 24(2):187–206, April 1996.

[6] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C-30:460–477, July 1981.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30:478–490, July 1981.

[8] J. A. Fisher. Global code generation for instruction-level parallelism: Trace scheduling-2. Technical Report HPL-93-43, Hewlett-Packard Laboratory, 1501 Page Mill Road, Palo Alto, CA 94304, June 1993.

[9] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu. Superblock formation using static program analysis. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[10] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

[11] W. H. Kohler. A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems. *IEEE Transactions on Computers*, C-24:1235–1238, December 1975.

[12] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[13] C. V. Ramamoorthy and M. Tsuchiya. A high level language for horizontal microprogramming. *IEEE Transactions on Computers*, C-23:791–802, August 1974.

[14] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 63–74, December 1994.

[15] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 57–69, December 1995.

[16] M. D. Smith. Architectural support for compile-time speculation. In D. Lilja and P. Bird, editors, *The Interaction of Compilation Technology and Computer Architecture*, pages 13–49. Kluwer Academic Publishers, Boston, MA, 1994.