Speculative Execution for Information Gathering Plans

Greg Barish and Craig A. Knoblock

University of Southern California / Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA 90292
{barish, knoblock}@isi.edu

Abstract

Although information gathering plans have enabled data from remote heterogeneous sources to be easily combined and queried, their execution performance suffers because access to remote sources is often slow. To address this problem, we have developed a method of speculative execution that increases the degree of run-time parallelism during plan execution. Our approach allows any information gathering plan to be automatically modified to support speculation in a manner that can lead to significant speedups, while ensuring that both safety and fairness are preserved. We demonstrate how speculative execution can be applied to a typical Internet information gathering plan to provide significant performance benefits.

Introduction

Improving the performance of information gathering plans remains an ongoing research challenge. Execution is often slowed by the time required to access remote sources. Plans that gather and extract information from Web sources are especially hard hit: they frequently need to interleave navigation with data retrieval and can thus require more I/O than other kinds of plans.

Consider the plan for the TheaterLoc information agent (Barish et al. 2000b) we recently developed. TheaterLoc integrates information from five Web sources, allowing users to query restaurants and theaters for any U.S. city and display their resulting locations on a dynamically generated map. TheaterLoc plan execution time averaged about 7 seconds – roughly 90% of that time was spent waiting for data from the underlying web sources. During that time, the CPU was idle and other local resources (such as network bandwidth) were under-utilized.

The inefficiency of information gathering plans has been a topic of current research on network query engines (Ives et al. 1999, Hellerstein et al. 2000, Naughton et al. 2001) and information agents (Barish et al. 2000a). Since it is impossible to control the performance of the network or of the remote sources, current research has instead focused on strategies for increasing the degree of run-time parallelism. Towards that end, various parallel execution techniques such as dataflow-style plan representation, data pipelining, and adaptive query execution have been proposed.

Despite the benefits of these techniques, data dependencies between operators can still hamper execution. For example, a query to a remote source often depends on the answer of a query to a previous source. Such binding-pattern style relationships require sequential execution and thus offer no opportunity for parallelization.

One solution to this problem is to engage in speculative execution, the process of executing instructions ahead of their normal schedule. Nearly all modern CPUs employ this technique as a means to address the I/O latencies associated with accessing local RAM. The underlying idea is that it is more efficient to probabilistically use an otherwise idle CPU than to not use it at all. Speculative execution has been shown to be one of the most effective means for increasing the level of instruction level parallelism (ILP) of a program (Wall 1991).

Just as it can increase the degree of ILP during program execution, speculation can also increase the degree of *operator-level parallelism* during the execution of information gathering plans. By speculating about the execution of future operators, it is possible to overcome stalls caused by earlier I/O-bound operators (e.g., those fetching remote data) and deliver better average performance. Further, applying speculative execution at a level much higher than that of machine instructions enables two additional benefits:

- **Significant performance improvement**. Instead of improving execution time in terms of *milliseconds*, a good result at lower levels of execution, speculative plan execution can result in gains of *seconds* – or at speeds relative to the slowest (I/O-bound) operators.
- **The opportunity to use more intelligent techniques for speculation**. CPU-level speculative execution occurs with very limited resources; consequently, the techniques for predicting program control and data flow are also limited. In contrast, plan-level speculative execution can leverage greater amounts of resources and reap the benefits of using more sophisticated machine learning techniques.

Contributions

In this paper, we describe an approach for speculative plan execution and demonstrate its effectiveness in improving the performance of a typical information agent. In particular, the main contributions of this paper are:

- An approach for speculative plan execution that yields arbitrary speedups while ensuring safety and fairness.

- Algorithms for automatically transforming any information gathering plan into one capable of speculative execution.
- Empirical results of applying speculative execution to one common type of information gathering plan.

We note that while our approach uses machine learning to decide *what* to predict, this paper focuses on describing the process (i.e., *how* to predict) and how it plans can automatically be modified for speculative execution. A separate paper will describe machine learning techniques for accurate and efficient data value prediction.

The remainder of this paper is organized as follows. Section 2 describes information gathering plans and provides an example. Section 3 describes the details of our technique for the speculative execution of such plans, including a discussion of how our technique ensures both safety and fairness during execution. Section 4 contains an algorithm that enables any information plan to be modified for speculative execution. Section 5 quantifies the impact of our approach on a typical Internet information gathering plan. Finally, section 6 discussed related work.

Information Gathering Plans

Information gathering plans retrieve, combine, and manipulate data located in remote sources. Such plans consist of a partially-ordered graph of operators $O_1..O_n$ connected in producer/consumer fashion. Each operator O_i consumes a set of inputs $\alpha_1..\alpha_p$, retrieves data or performs a computation based on that input, and produces one or more outputs $\beta_1..\beta_q$. The types of operators used in information gathering plans varies, but most either retrieve or perform computations on data.

Operators process and transmit data in terms of *relations*. Each relation R consists of a set of *attributes* (i.e., columns) $a_1..a_c$ and a set of zero or more *tuples* (i.e., rows) $t_1..t_r$, each tuple t_i containing values $v_{i1}..v_{ic}$. We can express relations with attributes and a set of tuples as:

$$R\ (a_1..a_c) = ((v_{11}..v_{1c}), (v_{21}..v_{2c}), \ldots (v_{r1}..v_{rc}))$$

Example Plan. To illustrate, consider the plan executed by an information agent called **RepInfo**. This plan, shown in Figure 1, gathers information about U.S. congressional officials. Given any U.S. postal address, RepInfo returns the names, funding charts, and recent news related to U.S. federal officials (members of the Senate or House of Representatives) for that address. RepInfo retrieves this information via the following web data sources:

- *Vote-Smart*, to identify officials for an address.
- *OpenSecrets*, for funding data about each official.
- *Yahoo News*, for recent news about each official.

In Figure 1, **Wrapper** operators gather data from Web sources, the **Select** operator filters federal officials from other types of officials, and the **Join** operator combines funding and news data into a single result. Wrapper operators are very common in web-based information gathering plans. They extract *structured* relations of data from *semi-structured* web pages.

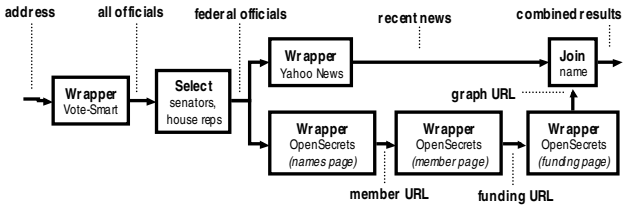


Figure 1: The RepInfo plan

At the start of execution, an input postal address is used to query the Vote-Smart source, returning the set of all officials for that location. The subsequent Select operator filters through just Senate and House officials. This subset is then used to query Yahoo News and OpenSecrets. Note that the latter requires additional Wrappers in order to navigate to the page containing the funding data. We describe why in the next section. The results of both are then joined, providing the result shown in Figure 2.

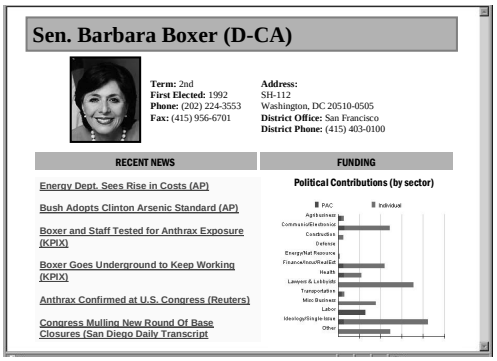


Figure 2: Results from executing RepInfo

Note that the plan in Figure 1 is one common type of information gathering plan. Similar plans that extract data from two or more distinct sources and then combine them together are common throughout the literature (Friedman & Weld 1997, Ives et al. 1999, Barish et. al. 2000b, etc.).

Execution

Execution of an information gathering plan commences when input required by the initial plan operators becomes available. For example, in Figure 1, querying the Vote-Smart source occurs as soon as an input street address is provided. The initial operator(s) (Wrapper, in this case) fire and, in turn, route output data to their consumers, causing them to fire. This process continues until all operators stop firing and plan output (i.e., the query answer) has been produced.

Note that operator execution order is not determined by an instruction counter (i.e., as is the case in von Neumann machines), but rather by the availability of data. Thus, like dataflow programs, information gathering plans can be as theoretically parallel as their data dependencies allow. For example, in Figure 1, the gathering of funding and news data is independent and can be executed concurrently.

This type of parallelism, evident at plan compilation time, is also known as *horizontal parallelism*.

Most systems that execute information gathering plans also support the pipelining (or streaming) of relations between producer and consumer operators in the sense that they allow tuples output by a producer to be consumed as soon as they are generated. When all tuples have been streamed from the producer to the consumer, the producer sends an *End-of-Stream* (EOS) token to indicate this state. Data pipelining enables successive operators along a given dataflow path of the plan to execute in parallel on a logical relation. This type of pipelined parallelism, evident at plan execution time, is also known as *vertical parallelism*.

The pipelining of data implies two additional attributes of operators in an information gathering system: that they (a) perform computations on partial input and (b) maintain state. For example, consider the Select operator, which filters tuples according to some specified criteria. As input tuples are received from a producing operator, they can be evaluated in terms of that criteria and output immediately. Select must also maintain state: it needs to retain its filtering criteria until the EOS has been received.

Performance Issues

Despite the use of horizontal and vertical parallelism, execution performance for information gathering plans remains hindered by I/O latencies and data dependencies between operators. In particular, slow producers starve their consumers. For example, in Figure 1, if Vote-Smart responds slowly, execution of Select will also be slow, as it requires an answer from its producer in order to execute.

Information gathering plans that use Web sites as data sources fare even worse because of the need to interleave navigation with retrieval. To illustrate, consider the three Wrapper operators required to obtain the funding graph from OpenSecrets. Given an official name, these wrappers are used to retrieve a bar graph showing the money various industries have contributed to that official's campaign.

Three wrappers are necessary because OpenSecrets does not allow this graph to be queried directly. Instead, the graph can be obtained only after navigating through the screens shown in Figures 3a-3d. In particular, for each official, we must submit the name of that official to the first page as shown in Figure 3a. This returns the set of campaign year URLs for the official we are querying (Sen. Barbara Boxer in this case). Each campaign year URL corresponds to a different year of candidate funding. Next, as Figure 3b shows, the campaign year of interest (2002) is selected, allowing us to obtain the member page for that year, shown in Figure 3c. However, the graph we want is only accessible by following the "Sectors" URL circled in Figure 3c, resulting in the final page that contains the graph, shown in Figure 3d. In short, obtaining the funding graph for each official requires the following three wrappers:

- One returning campaign URLs given an official name
- One returning a Sectors URL given a campaign URL
- One returning a funding graph given a Sectors URL

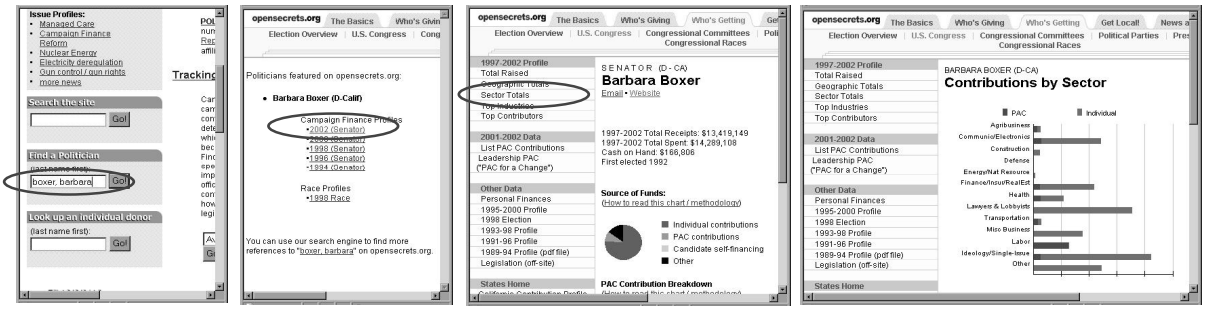
Thus, execution of the OpenSecrets part of the RepInfo plan is inefficient for two reasons. One is the inherent latency of accessing a remote source. The second is the need to navigate to the data we want, which requites multiple remote accesses and thus compounds the effects of I/O latency. Interleaving of navigation with retrieval is a problem unique to the Web and is one major reason why Web-based information gathering plans are slow.

Speculative Plan Execution

To address the performance dilemma, we describe a new form of run-time parallelism: *speculative plan execution*. The general idea is to use *hints* received at earlier points in execution to generate speculative input data to dependent operators and pre-execute them. Thus, consumer operators that are dependent on slow producers can be executed in parallel with those producers, using the input to those producers as hints for how to execute.

The mapping between hints and the predictions they lead to occurs over time. This relationship can either be cached or learned from earlier executions. The key difference between caching and learning here is that the former allows us to predict based on input we have seen before whereas the latter potentially allows us to predict based on hint input we have *never* seen. We describe the details related to learning how to predict data in a separate paper; here, we simply focus on the mechanism for speculative execution and how its application can be automated.

To better understand the concept of speculative execution, let us return to the RepInfo plan and hypothesize about one scenario for speculation. Consider



(a) (b) (c) (d)

Figure 3: Interleaving navigation with retrieval at OpenSecrets.com

the retrievals of the funding graph from OpenSecrets and the news headlines from Yahoo News. Both operations occur in parallel, but both are dependent on the representative names fetched from Vote-Smart. If this site is slow, performance of the rest of the plan suffers.

With speculative execution, however, we could use the input to Vote-Smart (i.e., the address) to predict the inputs for the OpenSecrets and Yahoo News wrappers. For example, we could learn that certain features of the address (such as city or zip code) are good predictors of the representative names that Vote-Smart will return. This would give us a reasonable basis upon which to predict queries to OpenSecrets and Yahoo – even for input we have *never* seen (i.e., same city, but different street address).

In this example, note that we are not limited to speculating about only one set of representatives – in fact, there is no reason why we cannot speculatively execute retrievals for multiple groups of representatives to improve our chances for success. For example, from prior executions, we could learn that *city*="Marina del Rey" and *zip*="90292" correspond to two different congressional districts, one represented by Jane Harman, the other by Dianne Watson. Identifying the correct one (based on street address) occurs during the processing of the Vote-Smart Wrapper. However, the capability to issue multiple predictions allows us to have the best of both worlds and confirm only those predictions that turn out to be correct. Speculatively executing the same path with multiple data is often useful when hints map to multiple answers.

The above scenario would allow us to execute the retrievals to the Vote-Smart, OpenSecrets, and Yahoo in parallel. Since all three tasks are almost entirely I/O-bound, true concurrent execution occurs. In fact, if the Vote-Smart source responds more slowly than OpenSecrets or Yahoo News, we can also execute the Join operation that follows these latter two retrievals. However, we have to be careful about what happens after that point – specifically, we cannot allow data to exit the plan until we have ensured that our earlier prediction was correct. In summary, while the above hypothetical scenario for speculation is thus fairly simple, it raises three important requirements. Specifically, it is important to:

- **Define a process**: It is important to specify how speculative execution actually works – what triggers it, how are predictions made, etc.
- **Ensure safety**: We must limit speculative execution from triggering an unrecoverable action (such as the generation of output or the execution of an operator affecting the external world) until our earlier predictions has been verified. Thus, we must confirm speculation
- **Ensure fairness**: Speculative execution should not be prioritized at the same level as normal execution. Its resources demands should be secondary. For example, the CPU should not be processing speculative instructions while normal execution instructions await.

In the next subsections, we describe our approach in terms of each of these three requirements.

Process

Our process for enabling speculative execution involves augmenting a plan with two additional operators. The first, **Speculate**, is a mechanism for using hints to predict inputs to future operators, and later for correcting or confirming those predictions. The second operator, **SpecGuard**, halts the flow of speculative data beyond "safe points" in a plan until earlier predictions can be confirmed or corrected.

We show how to deploy these operators in Figure 4, a transformation of RepInfo for speculative execution. The plan in the figure executes in the manner described above – that is, it uses the input address to predict the names of the federal representatives for that address. As the figure shows, a Speculate operator (denoted by SPEC) receives its hint (the address) and uses it to generate predictions about representative names. These names, in turn, drive the remainder of execution, while the first part of execution continues. Note that the final Join can also be executed – the only requirement is that a SpecGuard operator (denoted by GUARD) be the last operator in the plan. It prevents speculative results from propagating until Speculate has confirmed its predictions. We now describe the Speculate and SpecGuard operators in more detail.

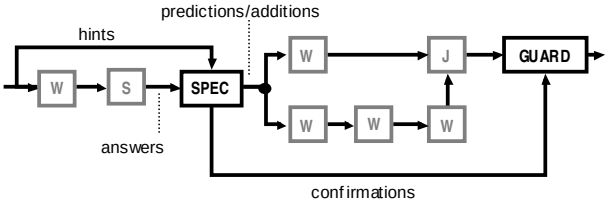


Figure 4: RepInfo modified for speculative execution

The Speculate Operator. The inputs and outputs of the Speculate operator are summarized in Figure 5. As the figure shows, this operator receives *hints* (input data to an earlier operator in the plan) and uses those hints to generate data *predictions* (used as input to operators later in the plan). Later, Speculate receives *answers* to its earlier predictions from the operator normally producing this data. Using these answers, *confirmations* can be generated to validate prior predictions. Any data errantly predicted is not confirmed and data that was never predicted is communicated via the *predictions/additions* output. For example, in Figure 4, suppose that an address is used to predict representatives X and Y. OpenSecrets and Yahoo News information is collected and combined for each, but

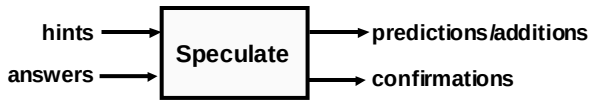


Figure 5: The Speculate operator

held up at the SpecGuard operator. At the same time, suppose that the Speculate operator receives an answer from Select that indicates that the real representatives were X and Z. It can subsequently confirm only X to SpecGuard and propagate Z through the OpenSecrets, Yahoo News, and Join operators. Thus, because Speculate operates at the tuple level, corrections to its predictions are fine-grained and require only the minimum amount of additional work be done to correct a mistaken prediction.

The SpecGuard Operator. The functionality of SpecGuard is similar to that of a relational Select operator – it acts as a filter to a set of incoming tuples. Figure 6 illustrates its inputs and outputs: *probable_results* are the incoming speculative tuples, *confirmations* are generated by the Speculate operator, and *actual_results* are the filtered (correct) results. SpecGuard is used in our scheme to guard against the release of unconfirmed or errant tuples beyond a safe point in the plan. The main way it differs from a standard relational Select is in how it uses the *confirmations* data as a filter to halt all *probable_results* tuples until they have been confirmed.

Figure 6: The SpecGuard operator

Once again, note that our approach exploits the fine-grained property of execution that data pipelining provides. By basing our production of verified results on confirmations – instead of *errors* – we can output correct data as soon as possible, without waiting for the remaining corrections to be processed. SpecGuard can also continue to wait for corrections until it receives an EOS (controlled and propagated by Speculate).

Ensuring Safety

Ensuring safety during speculative execution means preventing errant predictions from affecting the external world in unrecoverable ways. As described above, the SpecGuard operator facilitates safety by only producing confirmed results – however, it must still be correctly placed in a transformed plan. To maximize speculative execution benefits while ensuring correctness, SpecGuard is placed as far as possible along a speculative path, occurring just prior to plan output or an "unsafe operator". In Figure 4, SpecGuard is located just prior to plan output.

Ensuring Fairness

Fairness refers to the need to control resources such that normal execution is prioritized over speculative execution. For information gathering plans, the primary two resources to be concerned about are processing power (CPU) and network bandwidth. Fairness with respect to the CPU can be ensured by the operating system. During execution, operators for information gathering systems are typically associated with threads and processing occurs at the tuple-level. By maintaining a pool of standard-priority "normal threads" and a pool of lower-priority "speculative threads", the former can be used to service normal execution while the latter can be used for speculative execution. Standard operating system thread scheduling thus ensures that speculative CPU use never supersedes normal CPU use.

In terms of bandwidth, there exists a large body of computer networking research on resource reservation. In addition to hardware-based (e.g., network switch bandwidth provisioning) and software-based (e.g., TCP/IP socket configuration) methods, network resources can also be controlled by limiting the number of speculative threads. A fixed number of such threads limits simultaneous connections and bounds the amount of speculative bandwidth concurrently demanded.

Optimistic Performance

The maximum, or optimistic performance benefit resulting from speculative execution is equal to the minimum possible execution time of a transformed plan. Calculating this requires computing the minimum execution times for each of its independent sequential flows and then choosing the maximum value of that set. Using the minimum execution time for each flow implies all predictions are correct and no further additions are needed.

For example, consider the optimistic performance of the plan in Figure 4. This plan shows three paths of concurrent execution: the Vote-Smart path p_x, the OpenSecrets speculative path p_y, and the Yahoo News speculative path p_z. If we assume that all network retrievals take 1000ms and all computations (such as Select and Join) take 100ms, the resulting flow performance is:

$p_x = 1000 + 100 = 1100$ ms
$p_y = 1000 + 100 = 1100$ ms
$p_z = 1000 + 1000 + 1000 + 100 = 3100$ ms

Since the original execution time of the plan (using these assumed values) would have been 4200ms, the potential speedup due to speculative execution is 4200ms/3100ms = 1.35. Note that if Vote-Smart had been slow, say 3000ms, overall performance would have been slower (6200ms) and potential speedup (6200/3100 = 2.0) greater.

Achieving Better Speedups

While a speedup of two allows us to halve our execution time and produce noticeable results, we can in fact do better. At first, it might not seem possible – since all speculation must be confirmed, execution time appears bound by either the time to perform speculative work or the time to process its confirmation. However, two additional techniques can be used to improve performance such that speedups well beyond two are possible.

Early confirmation. Normally, we need to confirm speculation immediately after the answer is produced.

However, if we consider that some operators are deterministic, and always produce the same results for a given input, we can theoretically confirm speculation prior to its normal production. For example, as Figure 6 shows, we normally need to confirm our guesses about federal officials at the same point in the plan where those officials are produced – as in the case of the plan in Figure 4, following the Select operator. However, Select is deterministic: for a given set of input data and filtering criteria, it always produces the same output. Thus, we could confirm our speculation at least one operator earlier in Figure 4, thus enabling a potential speedup of 6400/(3100-100) = 2.13. Obviously, the increased speedup depends on the number of deterministic operators preceding the data being predicted and their individual execution times. The reliability of early confirmation via deterministic operators is based on prior work related to inference rules for functional dependencies (FDs), specifically the *transitive rule* which states:

$$\forall \text{ relations X, Y, Z: } \{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$$

If we consider that each *relation* of intermediate results can be assigned a unique hash value, then the relationship between intermediate results input to and output from a deterministic operator is identical to the relationship between two sets of attributes whose tuples are dependent.

Cascading speculation. We are not limited to speculating about only one input. In fact, there are many cases where we can pursue multiple paths of speculation in parallel and then confirm them at once, resulting in very high speedups when our predictions are correct. To illustrate, consider a longer sequence of operators, such as that in Figure 7. Using our earlier assumption, the execution of 10 successive Wrappers normally takes 10 seconds. Predicting input *f* allows the first and last halves of the plan to execute concurrently, resulting in a new execution time of 5 seconds and a speedup of 2.

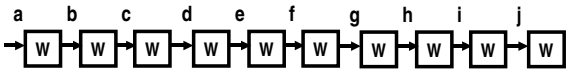


Figure 7: Longer sequence of operators

However, suppose that we used A as a hint to speculate about B, *the speculation of B* as a hint to C, and so on. This is shown in Figure 8 (Each Speculate operator is denoted by an **S**; SpecGuard by a **G**). Note that in the case of cascading speculation, only one SpecGuard is required.

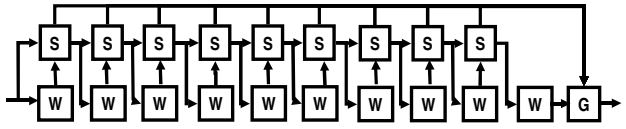


Figure 8: Cascading speculation

Since all wrappers require the same amount of time to execute and are all I/O-bound, they would act simultaneously and their confirmations could be processed at once. Thus, the resulting execution time would simply be the duration of a single wrapper call plus the overhead for speculation and the time to process confirmation. Even if we assume that the overhead and confirmation somehow requires an additional 100ms, execution would still only require 1000+100=1100ms, a speedup of 9.09.

Figure 9 shows a version of the plan in Figure 4 further modified for cascading speculation. If we make our earlier assumption that each network retrieval takes 1000ms and computations each require 100ms, then the five flows require (1100, 1100, 1000, 1000, 1100) and thus the optimistic speedup is 3200/(1100+100) = 2.67.

Automatic Plan Transformation

In the previous section, we described how speculative plan execution can yield significant performance gains. However, the augmentation of the example plan was done manually. In this section, we present algorithms that enable the automatic transformation of any information gathering plan into one capable of speculative execution.

Our overall goal is to maximize the theoretical average performance gain resulting from speculative execution. At the same time, we also need to be wary of the *overhead* (or cost) of speculative execution – this is primarily the additional execution time required because of the increased context switching that thread-level speculation demands. In particular, if the execution time for the unmodified plan P was represented as $T(P)$ and we identify a set of possible transformed plans $P'_1..P'_m$, then we are interested in the execution time $T(P'_i)$ as well as the overhead of speculation $C(P'_i)$ for each of these plans. Given this information, we then want to find the plan P_{best} such that:

$$P_{best} = \text{MIN}\,(T(P'_i) - C(P'_i)), \qquad 1 <= i <= m$$

Identifying P_{best} requires enumerating the set of candidate transformations and the calculating the most efficient one.

The Set of Candidate Transformations

One natural way to derive the set of candidate plan transformations is to use a brute force approach that generates a unique transformation for every case where a prior operator input can potentially be used as a hint to predict any future input along the same path.

Let $p_1..p_s$ represent the number of unique execution paths in a plan. Next, consider a single execution path, p_j, composed of operators $O_b..O_d \subseteq O_1..O_n$. To simplify, let us assume that each operator has only one input; thus, the

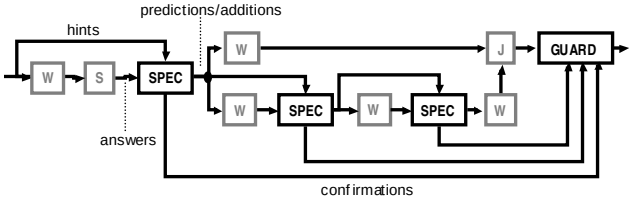


Figure 9: RepInfo with cascading speculation

set of inputs is represented by $\alpha_b..\alpha_d$. Then, the number of possible speculative transformations $S(\alpha_k)$ using the hint α_k, such that $b <= k < d$, can be calculated as:

$$S(\alpha_k) = (d-b)^2 - (d-b-1) \qquad \text{(Eq.1)}$$

In combining the set of possible transformations for a given input with the set of inputs along a given path p_j, we see that $S(p_j)$, the total number of speculative transformations for that path is

$$S(p_j) = \prod_b^d S(\alpha_k) \qquad \text{(Eq.2)}$$

Finally, to compute the total number of candidate speculative transformations *S(P)* for a given plan *P*, we need to combine the number of paths with the number of possible transformations per path. Specifically:

$$S(P) = \prod_1^s S(p_j) \qquad \text{(Eq.3)}$$

Thus, although a brute force approach allows us to consider every possible schedule for speculative execution, it quickly results in a large number of candidate transformations as the set of operator inputs increases.

The Most-Expensive-Path (MEP) Algorithm

We can reduce the size of the candidate transformation set substantially by leveraging Amdahl's law, which states that program execution time is a function of its most latent sequence of instructions. That is, it is not worthwhile to consider the transformations based on inputs or operators that *do not exist* along this path because any improvement does not allow overall execution to be any faster.

Thus, we can use a *most-expensive-path* (MEP) approach that identifies the most latent sequence of operators in an information gathering plan and focuses the generation of candidate transformations on that path. The MEP approach is well-suited for information gathering, a process that frequently requires the parallel retrieval of information that is later combined. Conceptually, an MEP-based transformation algorithm for a given plan *P* would:

1. Find all paths of *P* and their execution costs.
2. Identify the *MEP*.
3. Optimize the *MEP* for speculative execution.

Although this algorithm allows the MEP to be improved, what of the new MEP in the resulting transformed plan? Theoretically, it too can be improved by speculation – in fact, the refinement process can continue until it is no longer possible to optimize the MEP of the plan. The iterative refinement of plans for speculative execution is an important asset because it provides an anytime property and thus allows refinement to be bounded by some fixed time, if necessary. The detailed form of the algorithm above is called SPEC-REWRITE and is shown in Figure 10.

The figure shows that our refinement of the original plan consists of continually identifying the MEP for the current refinement and then optimizing that path via TRANSFORM-PATH (described shortly). Refinement stops when the MEP cannot be improved any further. This process enables us to focus our transformations only on the plan flow that determines overall execution time. In contrast to a brute force approach, it requires consideration of less candidate transformations and has an anytime property.

```
SPEC-REWRITE (plan)
{
  mep ← Ø
  curPlan := plan
  do
    bestCandPath ← Ø
    maxPathCost ← 0
    planPaths ← FIND-ALL-PATHS (curPlan)
    foreach path p ∈ planPaths
      curPathCost ← PATH-COST (p)
      if (curPathCost > maxPathCost) then
        mep ← p
        maxPathCost ← curPathCost
      if (mep != Ø) then
        candPaths ← GENERATE-CANDIDATES (mep)
        bestCandCost ← maxPathCost
        foreach candidate c ∈ candPaths
          curCandCost ← SPEC-PATH-COST (c)
          if (curCandCost < bestCandCost) then
            bestCandCost ← curCandCost
            bestCandPath ← c
          if (bestCandPath != Ø) then
            curPlan ← REPLACE-PATH (curPlan, mep,
              bestCandPath)
  while (bestCandPath != Ø)
  return curPlan
}
```

Figure 10: The SPEC-REWRITE algorithm

Enumerating Candidates

Having defined an algorithm that focuses transformation on a single path (the *MEP)*, our next task is to generate and compare the set of candidate transformations. This set includes each member $S(p_j)$, as defined by (Eq.2).

Figure 11 shows GENERATE-CANDIDATES, an algorithm returning the set of possible MEP transformations for speculative execution. The algorithm iterates over the set of *possible hints* (i.e., any input preceding the current operator) and builds a set of speculative paths using that hint (i.e., all combinations of predicting future inputs).

```
GENERATE-CANDIDATES (path)
{
  candSet ← Ø
  isFirst ← TRUE
  foreach operator op in path
    if (isFirst) then
      isFirst ← FALSE
    else
      possibleHints ← FIND-POSSIBLE-HINTS(op, path)
      foreach hint h in possibleHints
        curSpecPaths ← BUILD-SPEC-PATHS(h, op)
        candSet ← candSet ∪ curSpecPaths
  return candSet
}
```

Figure 11: The GENERATE-CANDIDATES algorithm

Evaluating Candidates

In terms of comparing the candidate set, we are interested in identifying the fastest *average* execution time of any of the possible speculative path transformations. We consider the average time because our candidate evaluation needs to account for speculation failure as well as success.

If we consider that m represents the original MEP and that $m'_1...m'_q$ are the set of possible MEP transformations, then the average execution time of that transformation $T(m'_i)$ requires two important statistics: the average execution time of each operator in that transformation and a measure describing the predictability of the inputs.

The average execution time $T_{avg}(O_i)$ of a single operator on m'_i can easily be determined by keeping a log of prior executions. Calculating the predictability of a particular future input to an operator given the existence of a prior input merely requires that we also keep a record of how these inputs correspond. For this paper, we assume that, for each of the collective set of inputs $\alpha_1..\alpha_x$, we are able to ascertain $P(\alpha_h|\alpha_g)$ where $g<h$ and thus: the probability that a prior input a_g can predict a future input α_h.

To see how to use average execution time and predictive probability to evaluate $m'_1...m'_q$, let us consider evaluating the sample path shown in Figure 11. Without any speculation, the original cost of this execution path is the sum of its individual operator average execution times:

$$T(m) = T_{avg}(Op_1) + T_{avg}(Op_2) + T_{avg}(Op_3) + T_{avg}(Op_4)$$

To calculate the execution time of each possible speculative transformation on that path, we must use average execution times and predictive probabilities to determine average path execution times. For example, Figure 12 shows one of the many possible speculative transformations of Figure 11.

Notice that per invocation, there exist four possible execution scenarios: (a) S1 and S2 are successful predictors, (b) S1 is successful but S2 is not, (c) S2 is successful but S1 is not, and (d) both fail. To compute the average execution time of the transformation in Figure 12, we calculate the time required by each of these four scenarios multiplied by the likelihood of their occurrence.

For example, if $P(\alpha_2|\alpha_1)=0.6$, $P(\alpha_3|\alpha_2)=0.7$, and each operator requires 1000ms to execute (assume SpecGuard requires 100ms), then we can figure the average execution time by first calculating the time $T(S_i)$ required for each scenario and the likelihood $L(S_i)$ of that scenario occurring:

(a) $T(S_1)=2000+100=2100$, $L(S_1)=0.6*0.7=0.42$
(b) $T(S_2)=3000+100=3100$, $L(S_2)=0.6*0.3=0.18$
(c) $T(S_3)=3000+100=3100$, $L(S_3)=0.4*0.7=0.28$
(d) $T(S_4)=4000+100=4100$, $L(S_4)=0.4*0.3=0.12$

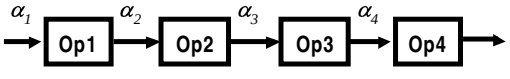


Figure 11: A simple operator sequence

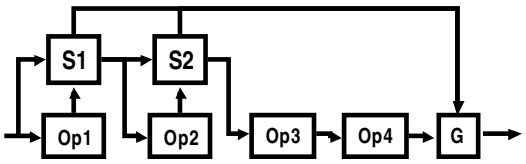


Figure 12: One possible transformation of Fig. 11

Thus, the average execution time of this transform is:

$$(0.42*2100)+(0.18*3100)+(0.28*3100)+(0.12*4100)$$
$$= 2800ms + \text{SPECULATIVE-OVERHEAD}(m'_i)$$

We summarize this average cost calculation in the SPEC-PATH-COST algorithm shown in Figure 13. As shown, the algorithm determines the average execution cost of a path by combining the overhead of speculation on that path with the summation of each scenario execution time multiplied by the likelihood that it will occur.

```
SPEC-PATH-COST (specpath)
{
  cost ← SPECULATIVE-OVERHEAD(specpath)
  execScenarios ← BUILD-SCENARIOS(specpath)
  foreach scenario s in execScenarios
    cost ← cost + PATH-COST(s) * LIKELIHOOD(s)
  return cost
}
```

Figure 13: The SPEC-PATH-COST algorithm

Experimental Results

To demonstrate the performance benefits of speculative execution, we applied our transformation algorithms to a real version of the RepInfo information gathering plan. We used RepInfo because it represented a common data integration task, similar to plans described in (Friedman & Weld 1997, Ives et al. 1999, Barish et al. 2000b).

Methodology

We experimented with four different RepInfo execution paradigms: normal (original plan), best-case (optimistic) speculation, worst-case (pessimistic) speculation, and average speculation (predictions correct 50% of the time).

To execute RepInfo, we used the Theseus information agent execution system (Barish et al. 2000a). Theseus is an information agent execution system, able to execute information gathering plans in a dataflow-style manner. To achieve high concurrency, Theseus uses threads to process operator firings (as they occur) and supports pipelined I/O between plan operators. Theseus was modified to support the Speculate and SpecGuard operators, running these operators at a lower priority than other operators. Theseus consists of about 15,000 lines of code and is written entirely in Java.

We ran Theseus on a Dell Latitude PC containing an 833MHz Intel Pentium III processor, 256MB of RAM, running Windows 2000 (Professional), and connected to our local LAN using a 10Mbps Ethernet card.

Normal Execution Performance

The original RepInfo plan was shown earlier in Figure 1. After building and executing this plan ten times, we found the following average execution times for its operators:

Operator	Time (ms)
Wrapper (Vote-Smart)	2010
Select	10
Wrapper ($OpenSecrets_1$)	2250
Wrapper ($OpenSecrets_2$)	2110
Wrapper ($OpenSecrets_3$)	2380
Wrapper (Yahoo News)	1250
Join	10

$OpenSecrets_1$, $OpenSecrets_2$, and $OpenSecrets_3$ correspond (respectively) to the wrappers for obtaining the names page, the member details, and the funding graph. Thus, the performance of each RepInfo path was:

Path	Time (ms)
VoteSmart→Select→$OpenSecrets_{1,2,3}$→Join	8770
VoteSmart→Select→Yahoo→Join	3280

Identifying the longer of the two paths gave us the normal execution time of the plan: 8770ms.

Speculative Execution Performance

After ten initial runs using identical input, we used SPEC-REWRITE to automatically transform the RepInfo plan into the plan shown in Figure 9, which we called **RepInfoSpec**. We then measured the performance of this new plan under optimistic, pessimistic, and average speculative success.

Transformation. As specified by SPEC-REWRITE, the MEP of RepInfo was first identified and then rewritten to a more efficient form, capable of speculative execution. The MEP of the resulting plan was also identified and improved. This process continued until it was no longer possible to improve the MEP. Below, we describe part of the first iteration of the SPEC-REWRITE algorithm; due to space constraints, we summarize but omit the details for the remainder of the steps.

For the first iteration of the plan, SPEC-REWRITE needed to consider the effect of how statistics related to the predictability of future inputs could impact average plan execution time. In particular, the following data was potentially predictable, in various combinations:

- The set of public officials o_p, consumed by Select
- The set of federal officials o_f, consumed by Yahoo and $OpenSecrets_1$
- The member URL m for each federal official, consumed by $OpenSecrets_2$
- The funding URL f associated with each member page, consumed by $OpenSecrets_3$
- The funding graph g for each federal official, consumed by Join

For purposes of illustration, we consider a subset of the combinations predictable: the effect of predicting each of the above data given the input address a. Using a measure of likelihood gained through applying machine learning techniques, the efficiency of the different scenarios was:

Scenario	L(S)	Average Execution Time (ms)
$o_p \mid a$	.86	(6760+10)*.86 + (8770+10)*.14 = 7051
$o_f \mid a$	.86	(6750+10)*.86 + (8770+10)*.14 = 7043
$m \mid a$	.16	(4270+10)*.16 + (8770+10)*.84 = 8060
$f \mid a$	.16	(6380+10)*.16 + (8770+10)*.84 = 8398
$g \mid a$	.16	(8760+10)*.16 + (8770+10)*.84 = 8778

Since it has the shortest average execution time, the best candidate is the scenario $o_f \mid a$, where address is used to predict the set of federal officials. As it turns out, Figure 9 shows that this scenario ends up being accepted as one of the eventual plan transformations. Continued application of the algorithm works in a similar fashion and eventually results in the plan in Figure 9. The final MEP consists of just the $OpenSecrets_3$ wrapper – however, since there is no further improvement possible, the algorithm terminates and returns the resulting plan.

Optimistic Speculation. To measure RepInfoSpec under best-case conditions, we used the same input that had been used in earlier runs of the original RepInfo plan. Thus, the probabilities of predicting all operator inputs was 100%. The resulting performance was 2400ms, indicating a speedup of 8770/2400 = 3.65.

Pessimistic Speculation. To measure the performance of RepInfoSpec under pessimistic conditions, we run it using new input. All speculation was thus incorrect and the plan had to execute operators in their original order in addition to suffering the overhead of speculation. The resulting plan performance was about 8790ms, a difference not noticeable when compared to original plan execution time.

Average Speculative Execution Performance. Finally, we considered the theoretical average performance time of RepInfoSpec under conditions where our predictions were only correct 50% of the time. To do so, we simply halved the execution times of optimistic and pessimistic versions of RepInfoSpec and calculated the resulting sum: (0.50 * 2400) + (0.50 * 8790) = 5595ms, resulting in a speedup of 8770/5595 = 1.57. Figure 14 summarizes and compares all four execution paradigms.

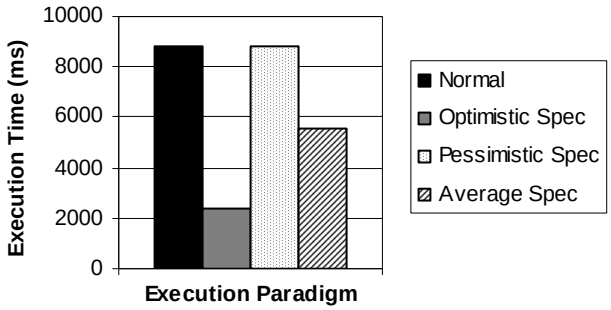


Figure 14: Impact of speculative execution on performance

Related Work

The approach described here represents a completely new way to execute information gathering plans. In the context of existing work on network query engines (Ives et al. 1999, Hellerstein et al. 2000, Naughton et al. 2001), it can be considered a new form of adaptive query execution. It is somewhat related to the partial results strategy of Niagara (Shanmugasundaram et al. 2000) in the sense that it involves execution of future operators based on unverified input data. The work here is also related to the dynamic operator ordering facilitated by Telegraph eddy (Avnur & Hellerstein 2000) in the sense that both involve out-of-order execution. The difference between the work here and these existing techniques is that speculative execution can be applied anywhere in a plan (not just near aggregate operators), is dependent on synthetic data, and can be cascading. Prior techniques operate on real data (thus limited by the latency of prior operators) and are only relevant to certain operators or *moments* during execution.

In a narrow sense, speculative execution can be thought of as a prefetching strategy (Adali et al. 1996, Godfrey & Gryz 1997). Like prefetching, speculation allows data to be retrieved from a source earlier than it normally would. However, unlike prefetching, speculative execution of information gathering plans does not lead to the problem of *stale data* – since it prefetches when plan execution starts, it can be seen as always retrieving the most recent data.

Finally, our approach to speculative execution has been inspired by both its historical and recent success at the system-level. In addition to work on classic processor branch prediction, (Chang & Gibson 1999) showed how speculation can improve file system performance, and (Hull et al. 2000) showed how "eager" execution can reduce the database latencies of e-commerce workflows. It is also worth noting that while there is a long history of speculative execution under von Neumann architectures, its use in dataflow machines has never been studied.

Conclusion and Future Work

In this paper, we have described an approach to the speculative execution of information gathering plans. We have shown how this approach represents a new form of run-time parallelism that can lead to significant execution speedups without sacrificing fairness or safety during execution. In addition, we have presented algorithms that enables any information gathering plan to be automatically transformed into one capable of speculative execution.

We are very encouraged by our initial results and plan to focus our future efforts on the problem of learning how to predict data. Although a simple caching scheme can be used to map hints into predictions, we believe that a more space-efficient and intelligent alternative strategy exists. Specifically, by applying standard machine learning techniques, we can learn classifiers or functions that enable us to not only efficiently store how hints map to predictions, but also enable us to predict data based on hints that we have never previously seen.

Acknowledgements

The research reported here was supported in part by the Air Force Office of Scientific Research under Grant Number F49620-01-1-0053, and in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152. Greg Barish was supported in part by a fellowship from Intel Corporation. Views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

Adali, S.; Candan, K.; Papakonstantinou, Y.; and Subrahmanian, V. 1996. Query caching and optimization in distributed mediator systems. *SIGMOD-1996.*

Avnur, R. and Hellerstein, J.M. 2000. Eddies: Continuously adaptive query processing, *SIGMOD-2000.*

Barish, G.; DiPasquo, D.; Knoblock, C.A.; and Minton, S. 2000. A dataflow approch for information management. *International Conference on Artificial Intelligence (ICAI-2000).*

Barish, G.; Knoblock, C.A.; Chen, Y-S.; Minton, S.; Philpot, A.; and Shahabi C. 2000. The TheaterLoc virtual application. *Proc. Innovative Applications in Artificial Intelligence (IAAI-2000).*

Chang, F. and Gibson, G. A. 1999. Automatic I/O hint generation through speculative execution. *Proceedings of Third Symposium on Operating Systems Design and Implementation (OSDI-99).*

Friedman, M., and Weld, D. 1997. Efficient execution of information gathering plans. *IJCAI-1997.*

Godfrey, P. and Gryz, J. 1997. Semantic query caching in heterogeneous databases. *KRDB at VLDB 1997.*

Hellerstein, J.M.; Franklin, M.J.; Chandrasekaran, S.; Deshpande, A.; Hildrum, K.; Madden, S.; Raman, V.; and Shah, M.A. 2000 Adaptive query processing: technology in evolution. *IEEE Data Engineering Bulletin*, 23(2).

Hull, R.; Kumar, B.; Llirbat, F.; Zhou, G.; Dong, G.; and Su, J. 2000. Optimization techniques for data-intensive decision flows. *Proc of Intl Conf on Data Engineering (ICDE-2000).*

Ives, Z.G.; Florescu, D.; Friedman, M.; Levy, A.Y; and Weld, D.S. 1999. An adaptive query execution system for data integration. *SIGMOD-1999.*

Naughton, J.; DeWitt, D.; Maier, D.; et al. 2001. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, 24(2)

Shanmugasundaram, J. Tufte, K. DeWitt, D.J.; Naughton, J.F.; and Maier, D. 2000. Architecting a Network Query Engine for Producing Partial Results. *Proc of WebDB-2000 Workshop.*

Wall, D.W. 1991. Limits of instruction-level parallelism, *Proc of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-91).*