

Distributed Models of Thread-Level Speculation

Cosmin Oancea
Department of Computer Science,
University of Western Ontario,
London, Ontario, Canada,
N6A 5B7
Email: coancea@csd.uwo.ca
Phone: +1 519 859-2345
Fax: +1 519 661-3515

Jason W. A. Selby
School of Computer Science,
University of Waterloo,
Waterloo, Ontario, Canada
N2L 3G1
Email: j2selby@uwaterloo.ca
Phone: +1 519 888-4567 x4822
Fax: +1 519 885-1208

Mark W. Giesbrecht
School of Computer Science,
University of Waterloo,
Waterloo, Ontario, Canada
N2L 3G1
Email: mwg@uwaterloo.ca
Phone: +1 519 888-4567 x6582
Fax: +1 519 885-1208

Stephen M. Watt
Department of Computer Science,
University of Western Ontario,
London, Ontario, Canada,
N6A 5B7
Email: watt@csd.uwo.ca
Phone: +1 519 859-2345
Fax: +1 519 661-3515

Abstract

This paper introduces a novel application of thread-level speculation to a distributed heterogeneous environment. We propose and evaluate two speculative models which attempt to reduce some of the method call overhead associated with distributed objects. Thread-level speculation exploits parallelism in code which is not provable free of data dependencies. Our evaluation of applying thread-level speculation to client-server applications resulted in substantial performance increases, on the order of 3 times for our initial model, and 21 times for the second.

Keywords: distributed systems, concurrency, generics, thread-level speculation

Cosmin Oancea will be available to present our research at the conference.

Submitted to PDPTA'05 - The 2005 International Conference on Parallel and Distributed Processing Techniques and Applications.

Distributed Models of Thread-Level Speculation

I. INTRODUCTION

This paper applies thread level speculation to an area in which it has not been previously attempted, namely distributed systems, and finds that, besides the obvious parallelization benefit, this may effectively reduce the communication and dispatch overhead inherent to such architectures.

Distributed Software Component Architectures (DSCA) provide a mechanism for software modules to be developed independently, using different programming languages. These components can be combined in various configurations, to construct distributed applications. [1] proposes generic component architecture “extension” that provides support for parameterized (generic) components, and can be easily adapted to work on top of various SCAs (CORBA [2], DCOM[3]).

There is increasing interest in the subject of automatically exporting generic libraries across their initial language boundaries. Our experiments have exposed part of C++’s STL and Aldor’s [4] BasicMath libraries for use across the Generic IDL (GIDL) [1] and Alma [5] frameworks respectively. This work has also revealed several performance issues. First, the overhead associated with inter-component communication stalls can be quite significant. In the context of a distributed application, the network and dispatching overhead may become dominant. This is especially true for object-oriented languages since they expose smaller average method length. Second, separate compilation of components hinders traditional compiler optimizations such as inlining.

This paper explores the novel application of speculative techniques to a distributed environment that address the aforementioned issues. We propose two models of Thread-Level Speculation (TLS) that can discover parallelism that is not exploitable using traditional parallelizing compiler techniques. Their application can yield substantial performance benefits, even in the case when the underlying hardware is not a multiprocessor.

The first model attempts to overlap the client-server communication overhead with useful computation performed on the server side in the form of speculation. This allows multiple remote invocations to be replaced with fewer calls that the server expands in multiple speculative iterations of the same code. We obtained speed-ups as high as 191% when the client and server share the same machine, and 353% in the distributed case.

The second model simulates “procedure inlining”. The server (master) runs a predictor program that approximates the code that was supposed to be executed by the client. The client validates the correctness of the predicted version of the program using results sent back by the server. This model obtains speed-ups as high as 1154% when the client and server share the same machine, and 2110% for the distributed case.

The remainder of this paper is organized as follows. In Sections II-A and II-B we provide an overview of our

GIDL and TLS frameworks respectively. We then describe the application of TLS to a distributed heterogeneous environment in Section III. Afterward, in Section IV we report and analyze the performance benefits of exploiting the parallelism enabled by TLS in order to speed-up client-server applications. Finally, we conclude with the contributions of this paper and future work in Section V.

II. BACKGROUND

A. Distributed Generic Multi-Language Architectures

There are a very few mainstream distributed heterogeneous software component architectures in use today. Most notable are CORBA [2], and Microsoft’s DCOM [3] (integrated in the recently emerged .NET framework). These architectures employ a specification language to describe the interfaces that the client objects call, and the object implementations provide, separating the specification and the implementation aspects of a module. Generic Interface Definition Language (GIDL) [1] is a *generic* extension of such a language (CORBA’s IDL [6]), that allows applications using parameterized or generic types to be exposed to a heterogeneous environment. It defines a common model for parametric polymorphism that can be meaningfully supported by various languages, and resolves different binding times and different semantics of parametric polymorphism in various programming languages. The GIDL model captures the notion of both qualified and unqualified type parameters, i.e., parameters restricted, or not, to satisfy particular interfaces (for example, the generic type `A` in `Test<A: BaseClass>` is restricted to extend the `BaseClass` interface). In the context of this paper the GIDL is layered on top of the CORBA SCA.

B. Thread-Level Speculation

Thread-level speculation is an aggressive parallelization technique that can be applied to regions of code which cannot be parallelized using traditional static compiler techniques. Threads execute out of order, modifying their own state, and merge their changes into the global non-speculative state only when it is determined that the locations it read-from and wrote-to do not result in a data dependency violation. TLS, with its high inter-thread communication costs, is enabled by the emergence of chip-multiprocessors (CMP). CMPs contain multiple tightly-coupled processor cores on a single chip, which significantly reduce the costs of interprocessor communication. Their emergence has come about as the cost-benefit-ratio of instruction-level parallelism offered by superscalar VLIW processors has grown [7]. Even though commercial CMPs currently exist in the market [8], the cache coherency mechanism needed for speculation is not yet present.

TLS can be applied at the loop and method/function levels. At the loop level, speculative threads concurrently execute

iterations of a loop out of sequential order even when these *may* contain a true dependence. The thread assigned to the lowest numbered iteration is referred to as the *master* thread since it encapsulates both the correct sequential state and control-flow. It is the speculative cache coherency mechanisms job to detect the data dependencies across threads and initiate a rollback. In servicing a rollback the speculative state needs to be cleared and the threads affected by the violation are restarted to carry out the cancelled iterations. Method-level speculation overlaps the execution of a called method with the code downstream from the call-site. The region following the call is executed speculatively while the main thread executes the called method. In general, the downstream speculative region is quite small since data dependencies will occur between the parameters or return value of the two code segments. However, the length of a speculative region can be expanded through the use of value prediction. Simple, and efficient two-value and stride predictors can be applied to free up some possible dependencies with good results [9].

Even without hardware support, we set out to explore the benefits of TLS and implemented a software framework. Similar to [10], reads/writes of speculative locations are replaced with calls to functions which simulate the data dependency checking that would be present in a speculative cache protocol. However, our approach is at a much higher level than that of [10] who implemented their speculative framework in a mix of C, and assembly.

The initial idea behind our framework was to incorporate TLS into the repertoire of an adaptive dynamic optimizer such as JikesRVM[11]. Profiling could detect situations in which speculation might be applicable and even resolve statically unsolvable distance-vector equations which rely upon run-time values. This monitoring of the run-time state could be used to possibly reduce the number of dependence violations encountered by initiating threads separated by the observed dependence distance. The addition of TLS to a traditional parallelizing compiler could provide speed-ups where data dependence analysis fails to conclusively determine if dependencies exist across loop iterations. The access to the true run-time behavior of a program that a dynamic compiler has could as be used to direct the shape of the iteration space by identifying whether a block or cyclic iteration pattern is most applicable. Further adding to the adaptability of the system, profiling can be integrated into the rollback handler. The ratio of rollbacks to commits could be monitored and if an unacceptable threshold is reached, the run-time compiler could remove the speculative code. Many hardware based schemes suffer from the inability to control the amount of memory required by speculative threads in order to keep the main state isolated from the speculative state [12]. In our software approach we can resize or set an upper bound on the size of the speculative cache as needed.

In order for us to perform speculation in general Java programs (as opposed to very regular scientific applications) it is clear that a dynamic compiler carrying out the speculative transformations must be able to plug in speculatively aware versions of the Java class libraries. Specifically, in order to speculate on many common code sequences speculative

versions of the collection classes, such as `List`, are needed. Consider the common situation of iterating through a `List`. Given a speculative version of the `List` class, a dynamic compiler could replace the use of the sequential library with a speculative version which cuts the `List` into segments dependent upon the number of available processors. Each processor would then visit in parallel only its assigned part of the `List`, and dependency checking would be hidden behind the scenes in the implementation of the speculative `List` class.

III. DISTRIBUTED APPLICATIONS OF THREAD-LEVEL SPECULATION

This section introduces two TLS models, inspired by [10] and [13], which can be applied in a potentially multi-language, distributed environment. Performance improvements are derived from two aspects. First, the communication overhead is reduced by eliminating stalls between the client and the server, and secondly, by taking advantage of the server/client support for parallel execution. In most situations the second model yields better speed-ups compared to the first. However, in environments where security is of concern, the code migration aspect of the second approach might forbid its use.

Throughout this paper we assume that the server’s throughput is reasonable low (that is, the server has some idle time and is not over-run with clients requesting its services). Section III-A presents an overview of our approach, while Sections III-B and III-C introduce the two speculation models respectively.

A. Overview

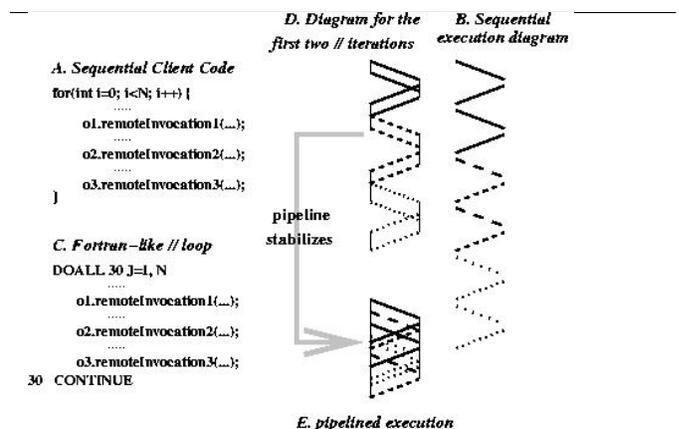


Fig. 1. An example of a simple object-oriented client program.

Figure 1.A presents an example of a general, object-oriented, client program, and Figure 1.B displays its normal (sequential) execution. However, if the loop can be executed concurrently, as evident in Figure 1.C, then the speed-up can be quite substantial. Figure 1.D shows the diagram’s temporal execution of the first two concurrent iterations. After some number of iterations, the *pipeline* stabilizes. Examining Figure 1.E, we see that the costs of the communication is ameliorated. The communication costs could be further decreased by

```

module TLSPackage {
  exception TLS_Dependence_Violation { long thread_num; };
  interface Speculative_Variable {
    void reset(in long tid, in long max_tid);
    void commitValueInFront(in long tid);
    void start_speculation();
  };
  interface Splittable_Variable<T: Splittable_Variable<T> > :
    Speculative_Variable {
    typedef sequence<T> Seq_T;
    Seq_T splitSpeculativeVariable(in long nr);
    void recombineIterators(in Seq_T s);
  };
};

interface GetValueObject {
  long getValue(); void setValue(in long val);
};

module IteratorPackage {
  interface Iterator<T> :
    TLSPackage::Splittable_Variable<Iterator<T>>{ // *
    long isEmpty(); void step();
    T value(); void resetIterator();
  };
};

module ContainerPackage { //...
  interface Vector<T: GetValueObject, C: Comparator<T> > :
    Container<T, C>, TLSPackage::Speculative_Variable{ // *
    T elementAt(in long i);
    void setElementAt(in T obj, in long i);
    T Spec_elementAt(in long i, in long thread_num); // *
    void Spec_setElementAt( // *
      in T obj, in long i, in long thread_num
    )raises (TLSPackage::TLS_Dependence_Violation); //....
  }; //....
}; //....

```

Fig. 2. GIDL specification. Lines marked with * denote TLS support

“inlining” the client code into the server. Additionally, server-side parallelism can be effectively exploited. This becomes more important as the granularity of a method increases.

Figure 1 displays an ideal Fortran DOALL parallelization of the program. However, this is not possible since the code is split and separately compiled between the client and the server. To achieve this, we employ our distributed TLS models that are discussed in Sections III-B and III-C.

B. Distributed Speculation Model

This section provides an overview of our TLS framework and describes its application to a distributed environment. Our model differs from that of a typical TLS scheme by the fact that the speculative variables may reside on a remote machine and therefore are not directly accessible by the client. However, the remote object whose methods uses these variables can act as a proxy for them. If the method’s parameters are also remote objects, than recursively, their server is required to provide parallelization support for the operations that are invoked upon them. If support for speculative parallelization is unavailable, and the code cannot be proven to be free of data-dependencies then speculation is not applied.

Figure 3 presents part of a two-client program that uses the services provided by a server that implements the functionality of the GIDL specification presented in Figure 2 (ignore for the moment the lines marked with * and the TLSPackage module). Assuming that the server’s code is available for analysis, note that the client code cannot be conservatively parallelized

due to the loop-carried true data-dependence of distance 1 in client A, and due to the indirect access of the vector’s vect elements in client B (see the lines marked ***). In both cases, profiling information combined with code analysis performed on the client may (non-conservatively) suggest that a region of rich-parallelism has been discovered. Suppose the if branch is *cold*, considering the *hot* path the code “resembles” a data-dependence free loop (modulo the data dependences introduced by possible object aliasing). Given these hindrances to parallelization our speculative framework can be employed.

```

// A
for(int i=0; i<dim[0]; i++) {
  GetValueObject gvo = vect.elementAt( new Long_GIDL(i) );
  int elem = gvo.getValue().getValue(); elem *= ...;
  if(elem>(-1)) gvo.setValue(new Long_GIDL(elem));
  else {
    GetValueObject gvol;
    if(i>0) {
      gvol = vect.elementAt( new Long_GIDL(i-1) ); //***
      elem = (long)gvol.getValue().getValue();elem*= ...;
    } else elem = ...;
    gvol = factoryImpl.createComparableObject
      (new Long_GIDL(elem));
    vect.setElementAt(gvol, new Long_GIDL(i));
  }
}

// B
for(; index_it.isEmpty().getValue()!=0; index_it.step()) {
  Long_GIDL ind = index_it.value();
  GetValueObject gvo = vect.elementAt(ind); //***
  int elem = gvo.getValue().getValue(); elem *= ...;
  if(isValidElement(elem)) {
    GetValueObject gvo = factoryImpl.createComparableObject
      (new Long_GIDL(elem));
    vect.setElementAt(gvo, ind); // ***
  }
}

```

Fig. 3. Two client code regions which are rich in speculative parallelism.

The client announces to the server that speculation is about to commence, and provides the required information regarding the speculative region. The TLS module used by the GIDL stub will invoke the target-language compiler (Java in our example) to compile the respective methods with support for speculation, thus generating some new (speculative-related) methods on the server side. While it is clear how this transformation would be implemented we are currently performing it by hand. Furthermore, it will modify the GIDL specification to also include speculation (lines marked with * together with the TLSPackage module in Figure 2), and re-compile it to update the client and server stubs.

Each interface that is found to contain at least one speculative method is required to inherit from the TLSPackage::Speculative_Variable interface (see Figure 2). Essentially, such an interface functions as a proxy for the speculative variables identified in its speculative-methods (as they do not have distributed support). Information received from the client will aid the server-side compiler to prune the number of variables that are considered speculative. However, if this is the only modification, the client-code labelled B in Figure 3 will generate many rollbacks due to the iterator step operation. To solve this, Iterator extends the Splittable_Variable interface, allowing each spec-

ulative thread to work with disjoint (separate) iterators (refer to Section II-B for speculative support for container classes).

```

T[] arr; TLS.Arrays.Spec_Arr_RefUID<T> spec_arr;
ArrayList<GIDL.TLSPackage.Speculative_Variable> Spec_Vars;
final public void start_speculation() {
    spec_arr=new TLS.Arrays.Spec_Arr_RefUID<T>(arr,1,1,ob_T);
    Spec_Vars.add(spec_arr);
}
final public void Spec_setElementAt(T ob, Long_GIDL a1) {
    arr[a1.getValue()] = ob;
}
final public void Spec_setElementAt(
    T ob, Long_GIDL a1, Long_GIDL th
) throws _TLSPackage.TLS_Dependence_Violation {
    int th_num = th.getValue();
    try {
        spec_arr.Speculative_Store(a1.getValue(), th_num, ob);
    } catch(TLS.Dependence_Violation exc) {
        throw new _TLSPackage.TLS_Dependence_Violation(th_num);
    }
}

```

Fig. 4. Part of the server-side speculative code for ContainerPackage::Vector

Figure 4 presents the `setElementAt` method and its speculative version `Spec_setElementAt`. Notice that the generated speculative code differs very little from the original. Specifically, it receives an extra parameter, the id of the thread executing the method (`th`). Second, the speculative operation is guarded by a `try-catch` block. If a violation is detected then the exception is forwarded as a GIDL exception onto the client. Finally, the container that may be the source of a data-dependence violation (`arr:T[]`) is replaced with a speculative version (in this case the `spec_arr:TLS.Arrays.Spec_Arr_RefUID<T>`). Figure 4 displays the implementation of the `start_speculation()` method exported by the `GIDL.TLSPackage.Speculative_Variable` interface. It initializes the variables on which data-dependence violations might occur, and stores them in a container. The `reset` and `commitValueInFront` methods (omitted from Figure 4 due to space constraints) traverse the list of speculative variables encapsulated by this class (`Vector`) and re-initializes them, or updates the original location that they shadow, respectively. These methods are invoked when handling a rollback or when speculation has succeeded and the speculative state should be merged with the true non-speculative state, respectively.

As depicted in Figure 5, the client starts speculative execution by creating a thread-manager, and calling the `speculate` method on it. The thread manager calls the `start_speculation` method on all local speculative variables, and on all the remote objects that act as proxies for the speculative variables identified on the server. Furthermore, it creates a pool of speculative threads (registered to itself) and starts them. A speculative thread executes iterations corresponding to the sequential code, except that it now references local speculative variables and invokes the speculative handler methods. At the end of an iteration the speculative thread checks to see if any violations were detected by the other threads. If so, the thread transitions into the waiting state. Otherwise it is assigned a new id (sequential execution iteration number), and checks to see whether the terminating condition was met. If a thread catches a data-dependence violation exception (thrown by local code or by the server), it invokes the `rollbackSTs` method on its thread manager, which will set the manager’s `barrier_id` flag. In the end, only the lowest id thread that has detected a rollback will be alive. At this time, for each speculative variable the value generated by the thread with the highest id less than or equal to the id of the running thread is committed. Finally, all the speculative variables are committed, and cleaned up. Adaptability is built into the system by monitoring the ratio of rollbacks to commits. If a predefined threshold is passed then speculation is abandoned for sequential execution, otherwise the speculative threads are awakened and speculation continues.

C. Distributed Speculative-Inlining Model

The second speculative model presented here, inspired by [13], achieves a speed-up in a similar manner as procedure inlining. More precisely, the client provides the server (or vice versa) with a *predictor* program that approximates the code executed by the client. There are no constraints associated with the distilled program. However, in order to produce a good speed-up, it needs to achieve a high prediction accuracy. The server (*master*) runs the predictor program and sends back to the client, records of the live variables computed along the anticipated path through the client’s code. It is the client’s responsibility to validate the correctness of the master’s execution.

Our model differs from [13] in several ways. First, [13] expects the distilled program to be much faster (a straight line code segment of the dominant path) than the slave’s verification code. In our case, we prefer the *approximate* program to be as close as possible to the original (and hence less likely to contain a violation), because of the high cost associated with a rollback. Second, our implementation is adapted to a distributed environment, and therefore, is geared toward other goals, such as network, and dispatching overhead elimination. The parallelization of the predictor program becomes more important for us as the iteration granularity increases.

There are two situations when program distillation is most beneficial inside of our framework. The first is when a method returns a predictable value. Consider a local

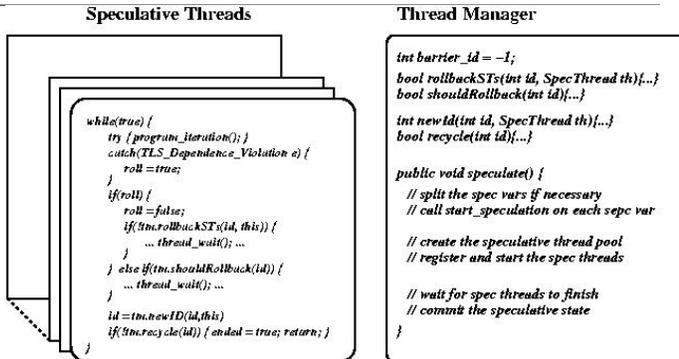


Fig. 5. The Interaction between the Speculative Threads and the Thread Manager

object which is used as a branch condition (see Figure 3.B: `if(client_obj.IsValidElement(...))`); in this case the *hot* branch will be added to the predictor but without the test (the test will be a remote invocation from the server point of view, and thus expensive). The second case, is when the deletion of a *cold* branch causes the number of speculative variables to drastically decrease, or the predictor code becomes conservatively parallelizable. In such a situation the server may even employ a standard parallelization model to achieve the greatest speed-up. In Figure 3.A, if the *true* branch from `if(elem > -1) ...` is found to be *hot* then a predictive program can be constructed by keeping the target, and removing the cold path. Further analysis by the server-side compiler of the predictor may conservatively discover that the vector’s element holder (`arr` in Figure 4) will not generate any data dependence violations.

The server side of the inlining speculative model is mainly composed from two communicating instances of our TLS framework, as shown in Figure 6.

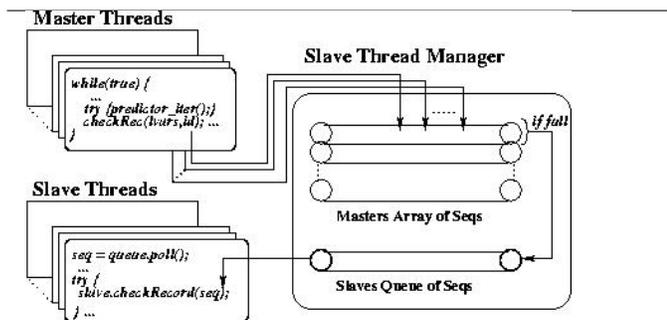


Fig. 6. “Inlining” - like Speculative Model. This figure presents the interaction between the Master/Slave Threads and the Slave Thread Manager

Master threads, registered to a master thread-manager, execute out of order iterations of the distilled program. At the end of every iteration, the live variables of the master threads are packed into a record residing in a predefined location in an array of sequences of records indexed by the thread’s id (viewed as a bi-dimensional array – the *Masters Array of Seqs* in Figure 6). Master threads are not permitted to over-write non-null records since this means that the record has not yet been committed because at least one thread is lagging behind. When a sequence is filled up, it is inserted into the *slave* queue (*Slaves Queue of Seqs* in Figure 6) and a new, empty sequence is placed in the table. The terminating condition of the master threads is dictated by the client’s code.

The slave threads poll a sequence from the slave-queue (if not empty, otherwise yield and try again). They request the client (that now acts like a server) to verify the current sequence containing several live-variable records. A slave-thread’s exit condition is reached when all of the master-threads are dead and no data in the slave-queue requires verification. No explicit synchronization is required between the master and slave threads except for guarded access to the slave-queue.

The client is responsible for verification. If any of the instructions that were not part of the *predictor* program (branch conditions excluded) are reached, or a *cold* branch

excluded from the predictor is taken, then a violation has occurred. The client throws a dependence-violation exception that will be caught by the corresponding slave thread on the server-side. The slave thread manager will handle the rollback as described in the previous section, additionally it will set the `barrier_id` flag of the master thread manager to the id of the thread that detected the violation. Thus all of the master-threads are going to be in a waiting-state (all have an id greater than `barrier_id`, otherwise the corresponding sequence wouldn’t have reached the client), and finally, only one slave-thread (the thread with the lowest id that detected a rollback) is running. Only then are the speculative variables committed and reinitialized. Control is then handed to the client which sequentially performs the iterations corresponding to the records in the received sequence.

```

module MasterSlavePack {
interface Master1<
  T: GetValueObject,
  C: ContainerPackage::Comparator<T>
> {
  void runMaster(in long i, in long j,
    in long s, in long l,
    in long sps, in long ms,
    in ContainerPackage::Vector<T, C> vect
  );
};
interface Slave1<T: GetValueObject> {
  struct LiveVariables {
    T elementAt_result; long thread_nr;
    long getValue_result;
  };
  typedef sequence<LiveVariables> seq_LV;
  void checkRecord(in seq_LV lv)
    raises(TLSPackage::TLS_Dependence_Violation);
  void performRollbackIteration(in seq_LV lv);
}; }
//...

```

Fig. 7. GIDL specification support for the inlining speculative model

Figure 7 presents the GIDL specification, corresponding to the client program displayed in Figure 3.A, that is needed by our “inlining speculative model”. When a client discovers a suitable code region for speculation, it locally creates and runs a slave checking-server (type `Slave1<...>`). The `Master1<E, C> createMaster1(Slave1<E> slave)` method creates a remote-object that upon invoking the `runMaster` method will create the server-side two-level TLS architecture described above. The `checkRecord` method in the `Slave1` interface performs the speculation validation. If a dependence violation exception is thrown the client is requested to sequentially execute several iterations (`performRollbackIteration(...)`).

As noted in the beginning of this section the inlining model almost always yields better speed-ups compared to the first approach. This is due to the fact that the number of remote calls performed by the two models is $1/(MasterCheckingSeqSize * NrOfRemoteCallsPerIt)$ in favor of the inlining speculative model. However, client code may reference many objects distributed across many servers, among which some may not support code exchange via a common intermediate representation (IR). Moreover, security issues may disallow the sharing of certain pieces of code or data. In this case, a combination of the two models

TABLE I

1ST ARCHITECTURE (OVERLAPPING COMMUNICATION)
 NR = CLIENT THREAD POOL SIZE,
 G = “REMOTE” METHOD GRANULARITY (INSTRUCTIONS)
 nMc SPEED-UP COMPARED TO SEQUENTIAL.
 n = NO. MACHINES, c = CLIENT VERSION
 $nMcR$ AS ABOVE, BUT WITH 1% ROLLBACK RATE.

Nr	G	1M1	1M1R	1M2	1M2R	2M1	2M1R	2M2	2M2R
4	10	1.35	1.30	1.30	1.23	2.23	2.05	2.05	1.98
8	10	1.55	1.51	1.56	1.52	3.01	2.72	3.24	2.71
16	10	1.65	1.53	1.62	1.53	3.36	2.76	3.36	2.68
32	10	1.91	1.47	1.69	1.44	3.22	2.37	3.46	2.27
4	10^3	1.31	1.28	1.30	1.28	2.09	2.03	2.13	2.03
8	10^3	1.51	1.45	1.53	1.48	3.12	2.72	3.16	3.07
16	10^3	1.62	1.46	1.62	1.46	3.29	2.94	3.47	2.66
32	10^3	1.73	1.48	1.70	1.35	3.53	2.31	3.53	2.17
4	10^4	1.25	1.23	1.32	1.26	2.25	2.03	2.04	1.86
8	10^4	1.36	1.27	1.50	1.38	2.71	2.35	2.78	2.39
16	10^4	1.41	1.24	1.55	1.32	2.83	2.35	3.17	2.41
32	10^4	1.44	1.25	1.63	1.24	2.73	2.01	3.41	2.05

is the preferred solution (if the code possesses high-level parallelism). The *master* is selected by identifying the remote object that is invoked most frequently. Predictive programs corresponding to the functionality of the servers that support a common communication IR and allow code migration will be also inlined into the master. If the code exposes parallelism, the execution time may be further decreased by concurrently executing speculative iterations of the master thread. We can see that one application may create a hierarchy of inlined speculations and overlapping speculative iterations (first model).

IV. RESULTS

Automatic library translation across language boundaries is an area yet to be explored. Unfortunately, it is lacking in formal benchmarks that can accurately measure the performance effects associated with porting a non distributed application into a distributed environment. We implemented a GIDL-server which exhibits functionality similar to that found in the STL of C++ (for example, containers, iterators, etc). Our tests are based on variations of the two examples used throughout this paper. The “remote” method granularity was varied from 10 to 10000 instructions (notice that each iteration performs between 3 and 5 remote calls). Our tests were carried out on two configurations. One configuration ran on a single machine which acted as both client, and server (2.4GHz P4/512 Mb). Another configuration employed two machines on the same local network (both 800MHz P3/256Mb RAM). All the machines we have used are running Linux.

We applied our TLS framework to distributed programming in the anticipation that speed-ups could be obtained by overlapping network stalls with speculative computation, thereby minimizing idle times. Table I shows the speed-ups obtained by employing our first distributed TLS model compared to sequential program execution. In a rollback-free (“ideal”) execution, employing a higher number of client threads generates a better speed-up (32 client threads achieve a 1.91, 1.69, 3.22, 3.46 times speed-up). Our framework is rollback-tolerant in the sense that it gracefully accommodates a 1%

TABLE II

2ND ARCHITECTURE (“INLINING”-LIKE SPECULATION)
 G = “REMOTE” METHOD GRANULARITY (INSTRUCTIONS)
 SS = SLAVE SEQUENCE SIZE,
 nMc SPEED-UP COMPARED TO SEQUENTIAL.
 n = NO. MACHINES, c = CLIENT VERSION
 $nMcR$ AS ABOVE, BUT WITH 1% ROLLBACK RATE.

G	SS	1M1	1M1R	1M2	1M2R	2M1	2M1R	2M2	2M2R
10	1	3.02	2.31	4.69	3.27	5.86	4.70	8.96	6.58
10^3	1	2.88	2.22	4.20	3.06	4.96	4.67	10.22	9.21
10^4	1	1.96	1.32	2.86	1.88	3.76	2.26	5.19	2.99
10	10	9.59	3.20	11.54	3.65	15.57	4.75	21.10	6.18
10^3	10	7.35	1.77	9.33	2.54	14.05	2.52	14.83	2.86
10^4	10	2.97	0.71	4.13	0.89	3.83	1.10	5.62	1.57

rollback probability. In examination of the cost of a rollback, we notice that the performance difference with respect to the ideal case decreases with the size of the thread pool. This is due to the greater number of inter-thread dependencies resulting in redundant work and increased synchronization overhead. The observed number of threads that provided the best speed-up was either 8 or 16.

Our second model clearly yields substantial performance benefits compared to the the first model as demonstrated in Table II. There are two main reasons for this. First, we have eliminated CORBA’s inherent remote-call dispatch costs by “inlining” the client code into the server. All remote calls in the initial code are now handled locally. Second, the network overhead is reduced by batched communication of the live variables. The server is configured to use 15 concurrent slave threads in order to “pipeline” the remote-client checking phase.

In an ideal (rollback-free) execution scenario, the application of this model obtains impressive speed-ups. On a single machine, execution time was 9.6 and 11.5 times faster, and 15.6 and 21.1 times faster over a distributed network with a method granularity, and slave sequence size of 10 (slave sequence size represents the number of records sent in a batch for the client to check for correctness). However, for a 1% rollback probability, the corresponding speed-up decreases dramatically (3.20 – 6.18). This is because, in our implementation, the rollbacks are handled by asking the client to sequentially execute the iterations associated with the sequence of records that have generated the violation (10 in our case). We are currently working on enhancing our architecture to better handle the rollback situation by sequentially executing only the “guilty” iteration. However, the rollback handling will remain expensive (see results in Table II for sequence-size 1) and influence our predicted program to be more “correct” than “distilled”.

Table I and Table II show that for both our models, the speed-up decreases when the method-granularity increases. However, in this case, taking advantage of the machine’s (potential) parallelism becomes very important. The final version of this paper will include tests executed on a parallel server.

To summarize this section, the performance gain for our first model (with respect to the sequential client program execution time) depends on the size of the thread pool, on the remote method granularity, and on the rollback ratio. The best speed-

ups, for a rollback-free execution, are obtained with 32 client threads and range from 144% to 191% when the client and server share the same machine, and from 353% to 341% for the distributed case, when the method granularity varies from 10000, to 10 respectively. For a 1% rollback rate, the best speed-ups are obtained using a number of threads between 8 and 16. They range from 127% to 153% for the single machine case and from 235% to 276% when the client and server are across a local network, for a method granularity of 10000 and 10 respectively.

The second model mimics “procedure inlining” and is very effective in eliminating the distributed system overhead. For a rollback-free execution we obtained speed-ups between 297% and 1154% for a single machine space, and between 383% and 2110% for the distributed case, for a method granularity of 10000 and 10 respectively. We also notice that a 1% rollback rate will substantially decrease these speed-ups, therefore we prefer a more “correct” rather than a more “distilled” predictor.

V. CONCLUSION

This paper has examined the potential for thread level speculation in a new area: the environment of distributed software components. We have found that substantial speed-ups may be achieved from this level of parallelism.

We propose two TLS models employed in a distributed setting that substantially reduce the network and call dispatch overhead. Additional speed-up is achieved when the underlying hardware is a multiprocessor. This becomes more noticeable as the remote method granularity increases.

The first model performs concurrent speculative iterations on the client, overlapping with communications. The second model mimics procedure inlining to eliminate distributed system overhead.

The performance gain depends on many factors. For the first model speed-up ranges from $1.4\times$ to $1.9\times$ on a single machine, and about $3.5\times$ when distributed. For the second model speed-up ranges roughly between $3\times$ and $11.5\times$ on one machine, and between $3.8\times$ and $22.1\times$ when distributed. Allowing a 1% rollback rate gives a somewhat smaller speed up for the first model, and substantially decreases speed-up for the second model.

For the final version of this paper, we plan to enhance our “inlining” like speculative architecture to better handle the rollback situations, and to investigate how a multi-processor server may influence the speed-ups as the method granularity increases. The creation of speculatively aware container classes proved to be a highly beneficial idea and warrants further investigation to determine other commonly used libraries where thread-level speculation can be exploited.

REFERENCES

- [1] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt, “Parametric polymorphism for computer algebra software components,” in *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computation*, 2004, pp. 119–130.
- [2] OMG, “Common Object Request Broker: Architecture and specification,” OMG Specification,” Revision2.4 (October 2000), 2000.
- [3] [Online]. Available: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/introremoting.asp>
- [4] S. M. Watt, “Aldor,” in *Handbook of Computer Algebra*, J. Grabmeier, E. Kaltofen, and V. Weispfenning, Eds., 2003, pp. 154–160.
- [5] C. Oancea and S. M. Watt, “A framework for using Aldor libraries with Maple,” in *Actas de los Encuentros de Algebra Cmoptuacional y Aplicaciones*, 2004, pp. 219–224.
- [6] OMG, “Common object request broker architecture — OMG IDL syntax and semantics,” OMG Specification,” Revision2.4 (October 2000), 2000.
- [7] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, “The case for a single-chip multiprocessor,” in *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. ACM Press, 1996, pp. 2–11.
- [8] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy, “Power4 system microarchitecture,” *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–26, 2002.
- [9] Y. Sazeides and J. E. Smith, “The predictability of data values,” in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1997, pp. 248–258.
- [10] P. Rundberg and P. Stenstrom, “An all-software thread-level data dependence speculation system for multiprocessors,” *The Journal of Instruction-Level Parallelism*, 1999. [Online]. Available: <http://www.jilp.org/vol1>
- [11] M. G. Burke, J. D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, “The Jalapeno dynamic optimizing compiler for java,” in *JAVA ’99: Proceedings of the ACM 1999 conference on Java Grande*. ACM Press, 1999, pp. 129–141.
- [12] M. Prvulovic, M. J. Garzar, L. Rauchwerger, and J. Torrellas, “Removing architectural bottlenecks to the scalability of speculative parallelization,” in *ISCA ’01: Proceedings of the 28th annual international symposium on Computer architecture*. ACM Press, 2001, pp. 204–215.
- [13] C. Zilles and G. Sohi, “Master/slave speculative parallelization,” in *Micro-35 Proceedings*. ACM, 2002.