Software Thread-Level Speculation – An Optimistic Library Implementation

Cosmin E. Oancea [*]
Computer Laboratory, Cambridge University, UK
Cosmin.Oancea@cl.cam.ac.uk

Alan Mycroft
Computer Laboratory, Cambridge University, UK
Alan.Mycroft@cl.cam.ac.uk

ABSTRACT

Software thread level speculation (TLS) solutions tend to mirror the hardware ones, in the sense that they employ *one, exact* dependency-tracking mechanism. Our perspective is that software-flexibility is, perhaps, better exploited by a *family of lighter, if less precise* speculative models that can be combined together in an effective configuration, which takes advantage of the application's code-patterns.

This paper reports on two main contributions. *First*, it introduces SpLSC: a software TLS model that trades the potential for false-positive violations for a small memory overhead and efficient implementation. *Second*, it presents *PolyLibTLS*: a library that encapsulates several lightweight models and enables their composition. In this context, we report on the template meta-programming techniques that we used to achieve performance and safety, while preserving library's modularity, extensibility and usability properties.

Furthermore, we demonstrate that the user-framework interaction is straightforward and present parallelization timing results that validate our high-level perspective.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming; D.2.2 [**Software Engineering**]: Software Libraries

General Terms

Performance, Languages, Design, Algorithms

Keywords

Thread-Level Speculation (TLS), Template-Metaprogramming

1. INTRODUCTION

Static techniques for automatic program parallelization have yielded good results for Fortran-like, scientific applications [1] running on SMPs. Non-numerical applications however, tend to exhibit complex control flow, irregular data-structure access, and dynamic loop limits. Moreover, handling inter-procedural/modular analysis and object/pointer aliasing in mainstream implementation languages, such as C++ and Java, further complicates the static dependence analysis to the point of yielding very "pessimistic" results.

Thread-level speculation (TLS) architectures address these hindrances by resolving at run-time the unknown dependencies that would otherwise prevent program parallelization. One of their main shortcomings, the high inter-thread communication cost, has been alleviated to some degree by the emergence of the commercial chip-multiprocessors (CMP). Hardware-centric TLS proposals [5, 11, 22, 15, 21] are in general more effective than the software ones, but involve complex and expensive changes to the basic cache-protocol. In addition, the potential for speculative storage overflow restricts these frameworks to exploiting finer-grain parallelism than that naturally available [14]. As a result, commercial CMPs do not yet offer TLS support.

Consequently, software TLS solutions are still worth investigating. One trend in the software-centric TLS design has been to mirror the modified cache-coherence protocol that enables the hardware-centric solutions [20]. This has led to frameworks that are very precise in all cases, but feature a high memory/time overhead.

We take the orthogonal perspective of constructing a family of lightweight, if more imprecise, TLS models that can be easily composed to parallelize an application. A variable-based memory partitioning can be performed first, and static access-pattern hints and profiling information can determine the most effective model to be employed for each partition.

One *contribution* of this paper is SpLSC: one such model that, while being safe and detecting all violations, trades *false-positives*, which may cause unnecessary iteration re-executions (rollbacks), against the *space and time overhead.*

For best performance, software-TLS solutions should ultimately be encapsulated into the repertoire of a dynamic optimizing compiler. However, being in an early experimentation stage, we decided to integrate our TLS models in a library. We named the latter *PolyLibTLS*, to underline our polymorphic view for supporting software-TLS. This approach has several additional advantages:

- *Availability:* existing models can be more readily tested under any C++ compiler and `pthread` implementation. Alternatively, the library can be easily integrated in the repertoire of a dynamic compiler.
- *Extensibility:* new models can be easily plugged-in.
- *Composibility:* we can test the effectiveness of combining various models. The results may indicate an "optimal" strategy or the necessity of developing a new model based on different trade-off properties.

[*]Supported by the Natural Sciences and Engineering Research Council of Canada (NSERC)

- *Comparability:* TLS models share now the same environment; their strength and weaknesses can be better tested against each other.

This paper presents *PolyLibTLS*'s structure and design. At high-level, our library encapsulates components from three main domains: speculative models, thread models, and thread managers. The first specifies the data-dependence tracking schemes, the second implements the parallel code, while the third "manages" the thread life and handles the potential mis-speculations. Our design is, in spirit, similar to the one of STL [9] (C++'s Standard Template Library), as we use domain orthogonality to enforce modularity, usability, and extensibility. In our case, the orthogonality is achieved through the use of USpM: a recursive templated class which provides (i) a single unified interface encapsulating the TLS models to the threads/thread manager, and (ii) a non-redundant way of specifying the parallelization strategy to the user. Using templates this way to define USpM enables independent replacement of all three of: dependency-checking schemes, thread-scheduling models, and the application itself.

From a software engineering perspective, the paper's main contribution resides in showing how to preserve the desired properties without compromising *performance* and *safety*:

Performance: At a high-level, the TLS instance operations replace the original read/write accesses; this low-granularity requires a highly-tuned implementation.

One problem is that the association between the to-be-accessed address and the TLS instance protecting it is not known until run-time. We employ a mechanism in which the compiler/user "guesses" the TLS instance. The guessed path enables (static) aggressive optimizations, and hence efficient implementation. An incorrect guess defaults to the slower, unoptimized, run-time dispatch.

Each speculative model provides buffers (one per thread, owned by the thread) for speculative storage. Best performance is achieved when the latter are implemented in the thread domain, to eliminate the book-keeping overhead. *Another problem*, however, is that an naive implementation seriously impacts the system's ease-of-use. Instead, we employ C++ functors for specifying inheritance, to make this association transparent to the user.

Safety: We have developed a family of functors that statically enforce certain invariants that cannot be expressed under F-bounded polymorphism [3] (the one in Java).

The reasons for developing *PolyLibTLS* under C++ are twofold: First, C++'s partial template specialization, and in general its template meta-programming capabilities, are instrumental in achieving the above mentioned properties. Second, C++'s explicit pointer-use allows us to implement the TLS solution globally, at the address level, with the result that the speculative support is effectively decoupled from the data-structure to which it is applied.

The rest of the paper is structured as follows: Section 2 briefly surveys the software-TLS area, presents our TLS model and motivates our design trade-offs. Section 3 presents a library-use example, argues that the transformation is straightforward, and introduces the main library components. Section 4 explains the design choices, based on template meta-programming techniques, that enables the desired software engineering properties. Finally, Section 5 shows performance results that validate our TLS combinatoric approach and Section 6 concludes the paper.

2. SOFTWARE TLS MODELS

For the rest of the paper we use the following notations: `N` is the size of the data-structure that requires speculative support, `M` is the loop's number of iterations, `C` is the maximal number of threads executing concurrently, and `W` is the maximal number of write operations per iteration.

2.1 Background

Thread-level speculation is an aggressive parallelization technique that is applied to regions of code that, although contain a good amount of parallelism, cannot be statically proven to preserve the sequential semantics under parallel execution. (For simplicity we discuss loop-level parallelization, although the same techniques apply for any thread-partitioning consistent with the control-flow total order.)

With TLS, threads concurrently execute iteration of a loop out of sequential order even when these *may* contain a true dependency. They use software/hardware structures, referred as *speculative storage*, to record the necessary information to track the inter-threads dependencies and to revert to a *safe* point and restart the computation upon the occurrence of a *dependency violation* (*rollback recovery*). The thread assigned to the lowest numbered iteration of all is referred to as the *master* thread since it encapsulates both the correct sequential state and control-flow; the others are *speculative* threads since they may consume "dirty" values.

To guarantee correct execution, threads merge their changes into the *global non-speculative storage* only when it is determined that the locations it read-from and wrote-to do not generate a dependency-violation. The usual implementation is to have the threads buffer their writes and commit them (serially) when they become master. Hardware approaches employ a modified cache coherency protocol to detect the occurrence of inter-thread data dependencies 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 canceled iterations.

The rest of this section briefly reviews several software TLS solutions. Papadimitriou and Mowry use the memory paging hardware to implement the TLS dependency tracking mechanism [19]. Although the implementation is efficient, their results are pessimistic because the high access tracking granularity (page level) generates too many rollbacks.

Rundberg and Stenstrom propose the S-TLS framework [20] where read/writes to (*speculative*) locations that cannot be statically disambiguated are replaced with high-level calls to functions that simulate the data-dependency checking of a speculative cache-protocol. (Assigning value `y` to variable `x` corresponds to the call `specST(&x,specLD(&y,i),i)`, where `i` is the iteration number.) The approach strengths are that it employs a *parallel* commit phase (scalable) and that is very precise, as it minimizes the potential for false-dependencies and identifies violations at the write instruction level. The downfalls are a huge memory overhead ($O(\mathtt{M}\times\mathtt{N})$) and a non-constant speculative operation overhead that can be prohibitively high for certain access patterns.

Cintra and Llanos's [6] and Dang, Yu and Rauchwerger's [8] approaches improve on the latter by decreasing the memory overhead to $O(\mathtt{N} \times \mathtt{C})$ and the speculative instruction overhead (through a more refined data-structure). This is achieved through a sliding-window mechanism that employs a serial commit phase, and hence trades-off scalability.

For completeness we enumerate several other TLS-related solutions that rely heavily on compiler support. Kazi and

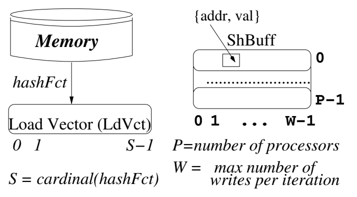

Figure 1: Speculative Storage Structure

Lilja propose a multi-thread pipelining framework [13] in which consecutive thread's stages compute the addresses of the write and read operations, respectively, and light synchronization is applied on the locations that fall in the intersection. Zilles and Sohi reduce communication overhead via a master slave model [25], in which the master executes an optimized (fast) approximation of the code, while the slaves verify its correctness. Finally, Chen and Olukotune's framework [4] exploits method level parallelism in Java.

2.2 Lightweight TLS Models

Our family of speculative models is designed to trade-off dependency-tracking precision for small memory overhead and performant, near-constant overhead of the speculative instruction (modulo memory-hierarchy effects). Furthermore, while other approaches implement the speculative support at the data-structure level, our design keeps them well-separated. The consequences are that switching between speculative (parallel) and sequential execution does not impose any overhead on the latter, and that aliasing is effectively handled.

2.2.1 Read Only Model (SpRO)

SpRO is the simplest, and most imprecise TLS model: the load operation returns the content of the to-be-read-address, while the write signals that the executing iteration is in error. SpRO's memory/read operation overhead is 0. Although very conservative, when combined with other models, SpRO is very effective when employed on variables that are only read, but that cannot be statically disambiguated by the compiler due to potential (improbable) aliasing relations.

2.2.2 Leightweight, Serial Commit Model (SpLSC)

SpLSC's design takes the perspective that if the current execution window is dependency-free then the threads can run in parallel and the violation-tracking scheme can be employed at the iteration's end. The speculative storage structure is depicted in Figure 1. For the moment assume that the map `hashFct` between memory locations and the dependency tracking vector (`LdVct`) is one-to-one. An `LdVct` entry stores the highest iteration number, corresponding to the sequential program order, that has read its associated address. The speculative write operation saves the `(address,value)` tuple in a buffer owned by each thread: `ShBuff[i]`. The implementation is straightforward and not shown here.

The speculative load operation (`specLD`) is presented in Figure 2. `SpecLD` receives as parameters the to-be-read address and the number of the iteration and the thread that performs the read.

The call to `markLD`, in line 2, implements the invariant that at any point, the maximal iteration number that has read an address should be inscribed in its associated `LdVct` entry. Two efficient, synchronization-free implementations

```
template<class T, class TH>      | //markLD is shorthand for:
T specLD( volatile T* addr,      | atomic{ if(LdVct[ind]<itNr)
        int itNr, TH* th )       |         LdVct[ind]=itNr; }
{                                | //or, in lock-free form:
1 int ind = hashFct(addr);       | do{ tmp = LdVct[ind];
2 markLD(ind, itNr);             |     if(tmp<itNr) {
3 if(!th->HasWritten(itNr, addr))|      LdVct[ind]=itNr;
4  return *addr;                 |      if(syncR[ind]>itNr)
5 else                           |       throw Dep_Exc(itNr-1);
6  return                        |      syncR[ind] = itNr;
7   th->getInternalWrite(addr);  |      tmp = itNr;
}                                | }   } while(LdVct[ind]!=tmp);
```

Figure 2: SpSC: Speculative Load

of `markLD` are shown on the right-side of Figure 2. (While the lock-free variant may in principle cause a rollback, this almost never happens in practice.)

The `if` expression (lines 3-7) handles iteration independent RAW dependencies. `HasWritten` is a light, conservative test that if unsuccessful guarantees that no same-iteration RAW dependencies exist. If so, the non-speculative memory is dereferenced and the value is returned. Otherwise the current thread may have written the needed value and its internal buffer is searched to return the proper value (`getInternalWrite`). This design choice is motivated by the fact that, in most cases, the compiler/user can eliminate such dependencies: `a[i]=x;...y=a[i];` $\rightarrow$ `a[i]=x;...y=x;` Although various lighter heuristics are available, a general solution for implementing `HasWritten` is to maintain a per-iteration write map against which the test is performed.

Before starting a new iteration a thread waits until it becomes *master* and then commits its internal buffer to memory (the writes are serialized). For each `{addr,val}` tuple to be committed, `lastLoad = LdVct[hashFct(addr)]` is read. If for all such tuples, `lastLoad` is less than the master iteration number, then it is guaranteed that no RAW dependencies have occurred (no successor thread has missed one of the master's updates). The current thread receives a new iteration to execute, and another thread becomes master and starts its commit phase. Otherwise, a RAW dependency may have occurred and the rollback procedure is started to annul the incorrect iterations, as detailed in Section 3.2.3. (The serial commit phase transparently satisfies the WAW and WAR dependencies.)

It should be easy to observe that even if `hashFct` is not one-to-one (and maps different locations to the same entry), the dependency tracking is still safe since it conservatively regards a read from address `a` as a read from all locations in $\texttt{hashFct}^{-1}$`(hashFct(a))`. The memory overhead depends on the number of writes per iteration (`ShBuff` size) and on the range of addresses ($[\texttt{a}_{min}..\texttt{a}_{max}]$) read during `C` consecutive iterations (`LdVct` size). As demonstrated in the next section, the latter ensures that `hashFct` introduces very few false-positive violations. It follows that, as opposed to the reviewed approaches, in many cases the model's memory overhead is a small fraction of the original program data.

We conclude this section by discussing Welc, Jagannathan and Hosking approach [24] of integrating *safe futures* in Java. Their design is similar to ours in the sense that dependencies are tracked at the iteration level and a serial commit-phase is employed, resulting in a memory overhead comparable with ours. Their implementation differs in several ways: *First*, whenever an item of shared state is written, the thread stack is scanned and any reference to the original

```
// assume N%U==0 & U=2^t        | SpRO spA(A,M*N); SpSC spB(B,N);
for(int i=0,id=0; i<N; id++){   | USpM<SpRO,&spA,SpSC,&spB> spU;
  for(int k=0; k<U; k++)  {     | //sum += A[i][j];
    sum = 0.0;                  | sum += spU.specLD<SpRO,&spA>(
    for (j=0; j<M; j++)         |                 &A[i][j], id);
      sum += A[i][j];           | //B[i] += sum;
    sum = 1.0/(1.0+exp(-sum));  | spU.specST<SpSC,&spB>(&B[i],
    B[i] += sum;  i++;          |  spU.specLD<SpSC,&spB>(&B[i],id)
} }                             |   + sum, id);
```

Figure 3: (A)Loop Example (B)Applying TLS Diffs.

object is patched to refer to the copy instead. Compared with our scheme this solves the same-iteration RAW problem at a rather high overhead cost: the write has $O(\mathtt{W})$ complexity. *Second*, to track down dependencies, each thread maintains private read/write maps that register the shared data access. This provides better cache-behavior, but with the disadvantage that the serial commit phase needs to scan and reset all (`S`) vector's entries. (Our approach accesses `LdVct` exactly once per write, and needs not reset it.)

2.3 Motivating Example and Discussion

Consider the loop in Figure 3.A, where $\mathtt{U} = 2^t$ and the sizes of arrays `A` and `B` are `N*M` and `N`, respectively. The variable names and the loop access patterns hint that the arrays `A` and `B` are disjoint and that `A` is read-accessed only. (Profiling techniques, currently under investigation, can also be used to strengthen these hints.) Assume however that aliasing prevents the compiler to prove it and hence static parallelization cannot be employed. We target the speculative parallelization of the outer loop. The S-TLS model would use a speculative storage size of order $O(\mathtt{N}^2 \times \mathtt{M}^2)$, and thus yield poor performance (since it ignores these hints).

We achieve better speed-up and a small memory overhead by employing SpRO for `A` (`spA`) and SpLSC for `B` (`spB`). To compose them, each model instance is associated with a memory range and a check is done to ensure that these intervals are disjoint. By default, SpRO protects the remaining locations. (For example, if (start of) `B<A`, `spB` is applied on addresses in $[\mathtt{B}, \mathtt{A})$, and `spA` on the rest.)

A unified speculative model (`spU` of type USpM) can aggregate these instances and dispatch the load/store accesses that cannot be statically disambiguated by the compiler to the appropriate TLS instance. Figure 3.B shows the necessary modifications, and hints at optimistic meta-programming: the programmer "guesses" the correct interval; if unsuccessful USpM recovers by searching all active models for the one dealing with the given address. A variant of this approach (USpM-var), instead merely causes a full rollback when the guess is incorrect. As shown in Section 5.1, with USpM-var the compiler optimizes the library code better (quite efficient when all guesses are correct); however, the rollback's cost is of the magnitude of thousands of instructions.

Note that, even if the static/dynamic hints are inexact, *soundness is still preserved*: For example, even if `B` spans over the beginning of `A` (aliasing), the accesses through `B` that fall in the range of `spA` are handled by the latter, read-only model (even if the programmer has wrongly "guessed" `spB`). These accesses may result in false-positive violations, and may impact on performance, but not on correctness.

Let us examine now the memory overhead of the USpM model. SpRO is weightless. For `spB`, the `ShBuff` size is $O(\mathtt{U})$ (the number of writes per iteration). We denote by `C` the number of concurrent threads. Suppose `C` consecutive

```
/** 1. Constants and Speculative Model Instances **/
const int UROLL=16, SZ=8, TH_NR=4, SHIFT=4;
HashPow2 hash(SZ, SHIFT);                  /* LdVct size is 2^SZ */
typedef SpSCcore<HashPow2> SCcore;         SCcore core(hash);
typedef SpSC<HashPow2,&core,double> SC; typedef SpRO<double> RO;
SC spB(B,B+N);                             RO spA(A,A+N*M);

/** 2. Create the Unified Speculative Model (USpM) **/
typedef USpM<RO, &spA, 0, USpM<SC,&spB,IndRAW_HA,USSpMfp> > UM;

/** 3. Thread Manager and Thread instances **/
class ThManager : public TM_INH<UM,UROLL,NORMALIZED_IT,SC> { };
class InstTh    : public TH_INH<InstTh,ThManager,SIMPLE>   {
  private: int i, N;
  public:
    int  end_condition()  const { return (i>=N); }
    void set_IndVars()    { i = (this->id-1)*UNROLL; }
    int  iteration_body() {
      double  sum = 0.0;
      for (j=0; j<M; j++) sum+=this->specLD<RO,&spA>(&A[i][j]);
      sum = 1.0/(1.0+exp(-sum));
      specST<SC,&spB>(&B[i], specLD<SC,&spB>(&B[i])+sum); i++;
}   };
int main() {  /** 4. Main **/
  ThManager tm;
  for(int k=0; k<TH_NR; k++) tm.registerSpecThread(new InstTh());
  tm.speculate();
}
```

Figure 4: Library Use. (See also Section 3.2.2.)

(outer) iterations access addresses in the range $[\mathtt{a}_{min}..\mathtt{a}_{max})$ belonging to array `B` (this is discoverable by profiling). A good value for `S` is $\mathtt{a}_{max} - \mathtt{a}_{min}$; in the case of Figure 3 where accesses are disjoint we have $\mathtt{S}{=}\mathtt{C}{\times}\mathtt{U}$. Choosing `hash-Fct(x) = (x-B)`$mod$`S`, where `B` denotes the start address of array `B`, results in a `LdVct` of size `S`, and guarantees that two distinct addresses accessed in the same execution window are mapped into different `LdVct` indexes, and hence do not introduce false-positives (see Figure 1 and Section 2.2.2). The memory overhead of USpM is thus $O(\mathtt{U} \times \mathtt{C})$ (where typically $\mathtt{U} \times \mathtt{C} \ll N$). Furthermore, the ratio of operations serviced by SpLSC/SpRO per iteration is 1/`M`, and since SpRO is the lightest TLS model we achieve near optimal speed-up.

We conclude this section by noting that the application of TLS model composition, described in this paper, relies only on (approximate) profiling techniques and not on (exact) alias/inter-modular analysis. As demonstrated in Section 5, if the former are accurate enough, the low memory overhead and efficient implementation lead to good performance, even when suffering an occasional false-positive violation.

3. THE LIBRARY AT A HIGH LEVEL

The previous section has advocated the usefulness of lightweight TLS models composition. While TLS should ultimately be integrated in a compiler repertoire, the library solution is more suitable for this experimentation stage: parallelization can be tested over various optimizing compilers, the implementation is more readily available and extensible, and does not require time-consuming recompilations.

We named our library *PolyLibTLS*, from polymorphic TLS library. Section 3.1 demonstrates the user-library interaction, and Section 3.2 introduces at a high-level *PolyLibTLS*'s main components and the relations between them.

3.1 PolyLibTLS: Use-Example

Figure 4 shows the TLS code that parallelizes the loop in Figure 3. First, the speculative models instances are created (`spA`, `spB`), and the start/end addresses that determine their associated memory-partition are specified. Second, USpM is constructed (`UM`). USpM is a recursive templated class that encapsulates the speculative parallelization strategy. The third generic type, `IndRAW_HA` for example, indicates how to handle loop-independent RAW dependencies (see Section 2.2.2). In our case we choose a lightweight mechanism in which each thread maintains the highest address written so far by a given SpLSC instance. If the address to be read is higher than the latter, it is guaranteed that the address has not been written by the current iteration. `USpMfp` is the fixed point that terminates the recursion.

Third, the thread manager and thread classes are implemented. The "inheritance" functors accomplish an easy user-library interaction, and statically validate the composition of the three components. Deriving from `TM_INH` says: "Here is the USpM model and the granularity of the thread iteration. I need a thread manager that assigns each thread to execute `UROLL` consecutive iterations as one expanded iteration" (`NORMALIZED_IT`). The inheritance functor for threads is similar; the `SIMPLE` attribute specifies that the thread is managed directly by the thread manager and is not supervising other (slave) threads. Next, the user writes the speculative code for the original loop iteration (`iteration_body`), and specifies the loop end condition, and how to update the loop induction variable for each "expanded" iteration (`set_IndVars`). Finally, the `main` function creates and registers the speculative threads with the thread manager, and the speculative execution commences (`tm.speculate()`).

At first sight, the difference between Figures 3 and 4, in terms of number of lines of code (LoC) may question the framework's ease-of-use. We argue against such a view: First, as the loop's body LoC increases, the LoC difference gets smaller, and the loop's body modifications are straightforward: read/writes are replaced with `specLD`/`specST` calls. Second, except for the TLS model creation, applying our framework is no more difficult than what the programmer has to do in Java, for example. The additional advantage is that the user does not need to understand the whole program and ensure the parallelization's soundness; the TLS safety-net allows him to work based on his intuitions of the "local" code. Third, Figure 4 can be seen as the intermediate representation of a more simpler, loop annotation-based interface. Finally, although our approach may require a learning curve, it is fairly straightforward: it took us about 3 hours to write the TLS version for the benchmark tested in Section 5.

3.2 PolyLibTLS: Main Components

PolyLibTLS comprises three major kinds of orthogonal components: threads, thread managers, and speculation models. The unified speculative model (USpM) is the high-level inter-component glue: it aggregates different instances of speculative models, and exports a unified interface to the threads/thread managers. It also simplifies the library use.

3.2.1 Speculative Models

At the moment, the library supports three speculative models. We have already introduced two of them: SpRO and SpLSC. The third one, named SpLIP, is a lightweight model that executes its updates in-place, and hence is scalable (since it does not employ a serial commit phase). Its design and performance are analyzed elsewhere [17].

Figure 5 shows part of the interface of the SpLSC model. The implementation is split into two classes: `SpSCcore` han-

```
template<class HashFct> class SpSCcore {
  volatile WORD* LdVct;           HashFct hashFct;
  SpSCcore(HashFct obj)           { ...                   }
  WORD getIndex(WORD addr)        { return hashFct(addr); }
  void markLD(WORD ind, WORD itNr){ /*already presented*/ }
};
template<class HashF,SpSCcore<HashF>* core,class T> class SpSC {
  T *addr_beg, *addr_end; /* interval */     typedef T ElemType;
  WORD addr_diff;         /*=addr_end-addr_beg;*/
  WORD checkBounds(T* address)                    {
    WORD diff = (WORD)(address-addr_beg);
    if(diff<addr_diff) return core->getIndex(diff);
    else               return WORD_MAX;
  }
  template<class TH> T specLD
    (volatile T* addr,WORD itNr,TH* th,WORD ind){ ... }
};
```

Figure 5: SpLSC Interface

dles the `LdVct` access. If the code read-access pattern for two variables is identical, it is desirable that they share the same `LdVct` for the obvious reasons. Therefore we allow different instances of `SpSC` to share the same "core" (the second template argument). `SpSC` holds the memory address range to which this instance provides speculative support. `checkBounds` returns either the address' corresponding index in `LdVct` or the `WORD_MAX` value if the address is out of its scope. (`specLD` has already been discussed in Section 2.2.2.) Although not shown, `T` is restricted to be one of the basic types or a pointer type. Accessing more complex data-structure need to be broken down in terms of these components; otherwise the system's soundness is violated.

3.2.2 Threads

Figure 6.A presents the thread's interface and indicates its life cycle. `SpecThread` receives two type-parameters. `TM` is the type of thread manager that "supervises" the thread. (The unified TLS model, `USpM`, is statically known via `TM`.) `SELF` is the self-type: the user's implementation class that is derived from `SpecThread`. Its use allows the user-implemented methods (those prefixed with `me->`) to elide the overhead associated with virtual function calls, which can be significant. (In the C++ literature, this construct is known as "the curiously recurring template pattern" [7]).

The thread spends its life executing the `while(true)` loop. The `acquireResources` call in line 4 returns only when the thread is ready to start a new iteration. This implies that the former iteration (SpLSC) writes have been committed, and the resources needed to start a new iteration are available. The latter include for example a new iteration number (`id`), and a serial computation/prediction of (scalar) variables which if executed in parallel would generate frequent rollbacks (`execSynchCode` call in line 7, Figure 6.B). The mechanism is similar to Zilles and Sohi approach [25]. The thread then updates the induction variables that can be determined solely from the iteration number (line 5).

Empirical data indicate that TLS is best applied on iterations comprising hundreds-to-thousands instructions. The `for` loop (lines 6–10) expands the original iteration to achieve the desired granularity. The `iteration_body` call executes the original iteration code, augmented with speculative support for the variables that cannot be statically disambiguated.

The `catch` block (lines 11–17) is entered when one of three types of events occur: the current thread detects a dependency-violation, or satisfies the end of the loop condi-

```
 template<class SELF,class TM>  |template<class UM, class ATTR>
 class SpecThread : public      |class ThreadManager_SC
    ThBuffComp<SELF,TM::USpM> { | :public ATTR { typedef UM USpM;
  WORD UNROLL; TM* tm;          |
  void run() {                  | template<class TH> void
 1 SELF* me =                   | acquireResources(TH* th)      {
     static_cast<SELF*>(this);  |1  WORD old_id = th->getID();
 2 while( true ) {              |2  this->ATTR::nextBigIterId(th);
 3  try {                       |
 4   tm->acquireResources(me);  |3  /** wait to become master **/
 5   me->set_IndVars();         |4  while(master != old_id) ;
 6   for(int i=0;i<UNROLL;i++){ |5
 7     if(me->end_condition())  |6  UM::serial_commit(th);
         throw Exc();           |7  th->execSynchCode();
 8     me->iteration_body();    |8  master=ATTR::nextMasterID();
 9     tm->setNextID(me);       | }
 10  }                          |
 11 }catch(AnyExceptOrError e){ | template<class TH> int
 12  wait_to_become_master();   | rollbackProc(WORD safeId,TH*th){
 13  tm->rollbackProc           |  //kill all other spec threads;
        (getSafeId(e),me);      |  //request each TLS model to:
 14  if(dependence_except(e))   |  //  roll back its mem partition
 15   tm->respawnThreads();     |  //    to a safe point and
 16  else if(end_condition())   |  //  reset their spec. storage
         return;                |  //    (ShBuff, LdVct, etc)
 17  else throw e;              |  //execute several iters sequent
 } } }  ...  };                 |}};
```

Figure 6: A: Thread. B: Thread Manager

tion (line 7), or yields an exception/error. The treatment is similar. The current thread waits to become *master*. (The master trivially preserves both the data and control flow correctness.)

If so, it follows that the exception/end-of-the-loop are not mis-speculation artifacts. The thread manager is invoked (`rollbackProc` on line 13) to kill the other threads and to undo the successor thread's updates. In the case of a dependency-violation it also resets the speculative storage and executes one iteration sequentially to ensure the systems makes progress. Finally, speculation either continues (dependency-violation), or ends (exception/end-of-loop).

Otherwise, the event originated in a mis-speculation, and the current thread is eventually killed by the master thread who discovers it. (This mechanism also solves the case when a thread enters an infinite loop due to a "dirty" read.)

We have seen that SpLSC requires the thread to have buffering capabilities. This is achieved through the use of meta-programming in the definition of the `ThBuffComp<SELF,UM>` class. Ultimately, for each SpLSC instance in `UM`, `SpecThread` extends an appropriate buffering component (see Section 4.2).

Following Bhowmik and Franklin work [2] on multi-thread partitioning, an useful extension would be to classify the threads into *leaders* and *slaves*, where a *slave* can be spawn by a *leader* from anywhere inside his iteration. (The spawning is delayed until certain dependencies get resolved.)

3.2.3 Thread Managers

The thread manager has two main functions: First, it assigns new iterations and the associated resources to the threads it supervises while enforcing the necessary invariants.[1] For example, if a SpLSC instance is used, a thread may commit its writes only when he becomes *master*. Second, it implements the *rollback recovery* procedure.

The library supports two main types of thread managers: one that assigns a new iteration to a thread only after it has become master (`ThreadManager_SC`), and another one that allows threads to be more loose, as long as the distance between the highest and lowest concurrent executing iteration is less than a certain threshold. (The latter obviously cannot work with SpLSC instances.)

[1]Spawning new threads for each iteration carries a significant overhead, therefore we chose to re-use them.

Figure 6.B presents part of the thread manager's interface. It receives two template parameters: the unified speculative model, and a trait [16, 18] component, `IT_ATTR`, that specifies the iteration policy. This can be *normalized* or *interleaved*. With the former, `th1` executes iterations `{0,1,..,U}`, `th2` ∈ `{U,..,2*U-1}`. With the latter, `th1` ∈ `{0,4,8...}`, `th2` ∈ `{1,5,9...}` and so on. Note that, when SpLSC is used, the interleaved option is inappropriate, since the serial commit operation would incur too much overhead.

The `acquire_resources`'s implementation waits for the executing thread to become master (line 4) prior to committing his writes to non-speculative storage (line 6). The latter may throw a dependency-exception, which is caught in the thread's `run` function, and the rollback-recovery mechanism (`rollbackProc`) has already been discussed. Otherwise, the thread starts a new iteration and the master iteration is advanced (line 8). Note that our design is decentralized: the thread manager is not a thread but a piece of code that is executed by all speculative threads.

4. TEMPLATE META-PROGRAMMING: EASE OF USE AND PERFORMANCE

We start this section by comparing the STL's design with ours, in Section 4.1, and then Sections 4.2 and 4.3 present the template meta-programming techniques we have used to improve the framework's ease of use and performance.

4.1 PolyLibTLS - STL

In spirit, the *PolyLibTLS* design resembles the one of the C++ Standard Template Library (STL) [9]. To achieve a high level of modularity, usability, and extensibility to its components, without impacting the code's efficiency, STL components are designed to be *orthogonal*. This is in contrast with traditional approaches where, for example, *algorithms* are implemented as methods inside *container* classes. In STL, the orthogonality of the former domains is achieved, through the use of iterators: the algorithms are specified in terms of iterators that are exported by the containers.

PolyLibTLS adheres to the same design principles:
Modularity and Extensibility: To develop and experiment new parallelization strategies, easy component-plug-in is a must. Moreover, library-use implicitly means to extend it: the thread must be at least provided with the iteration code.
Performance: At a high level, the library replaces memory location accesses to function calls that in addition track down dependencies. Working at such a low granularity requires a performant implementation to obtain any speed-up.
Safety: debugging multi-threaded applications is notoriously difficult, therefore it is important to have some static guarantees that the components are properly combined. F-bounded parametric polymorphism [3] is not enough in achieving this: some relations can be statically validated only with the use of meta-programming. (Eg: if USpM contains an SpLSC instance, then do not use an *interleaved* thread manager.)
Ease of use: it should be straightforward for the programmer to use the library, and it should also be easy to integrate the library in the repertoire of a dynamic compiler.

It comes then, as a natural consequence, that *PolyLibTLS's* components are also designed to be orthogonal. The orthogonality between the threads/thread manager and the TLS model instances is achieved through the use of the unified TLS model (USpM). In contrast to STL's iterators, the latter is not a run-time object but a recursive-templated class that contains only static members. *On the one hand*, USpM presents to the other components the simplified view of one speculative model, and carries out the obvious software-engineering benefits. More important, it allows static validation of component's composition and it generates faster code than the alternative run-time-oriented solutions.

On the other hand, this allows the user to separate concerns. Constructing USpM defines the parallelization strategy, as the program variables are mapped to customized speculative model instances. The next, orthogonal step, the thread implementation, is an "automatic" process of replacing variables reads/writes with calls to the `specLD`/`specST` functions. There is no redundancy in the user's work.

4.2 Meta-Programming: Ease of Use

C++ templates and partial specialization allows the compiler to perform static computations, encouraging a style known as template meta-programming [23]. Known to be Turing-complete, these techniques have been used in principle to provide static type checking to traditionally unchecked operations, or to achieve a better degree of optimization.

Boost MPL [10], one of the most well known library for meta-programming provides a high-level framework of compile-time algorithms, sequences and meta-functions. In our implementation we use the MPL's `enable_if`/`disable_if` constructs [12] developed by Jarvi, Willcock and Lumsdaine. `enable_if` takes two class parameters, the second with the default value `void`. The first type parameter has a static `value` member. If the latter evaluates statically to true, `enable_if` exports the second type-parameter through the name `type`. Otherwise it does not. The `disable_if` template has the reverse functionality. They are used as additional default parameter to control which functions to be considered for overload resolution and which not.

Figure 7 presents the definition of the USpM recursive template, and shows how different template-functors are used to statically guarantee safety and to ease the user-library interaction. The `validSpM` functor takes as parameter a speculation model, one of its instances, and a thread attribute and determines whether the combination is valid. We enumerate the valid compositions via partial specialization: the `value` static member is set to `true` and the corresponding thread bufferable component is exported via the `type` member. Otherwise no `type` member is defined and `value` is `false`. The `is_same_type` is "true" if the type parameters are the same and false otherwise.

USpM is either a recursive template in which the first three parameters form up a valid composition, as specified by the `validSpM` functor, or is the fixpoint type that ends the recursion (`USpMfp`). Note that otherwise the compiler will signal an error: either `Cont` parameter is not an USpM instance or the combination is not legal (at the definition of `V1` and `V2` respectively). The implementation also ensures that the user may instantiate the USpM's default parameter only with a valid type, otherwise type-checking fails (`is_same_type`).

Finally, the `TH_INHO` functor "returns" a type that inherits from all the bufferable thread components associated with the speculative model instances in the definition of USpM. Its use, in the `ThBuffComp` is transparent to the programmer.

```
/******* Functor: (SpecModel) -> BufferableThComponent ********/
template<class M, M* m, const WORD TH_ATTR> struct validSpM
    { static const int value=false; };
template<class T, SpRO<T>* m> struct validSpM<SpRO<T>,m,RO_ATR>{
    typedef Buff_RO<T,SpRO<T>,m> type;
    static const bool value=true;                              };
template<class H,SpSCcore<H>* c,class T,SpSC<H,c,T>* m>
        struct validSpM< SpSC<H,c,T>, m, IndRAW_HA_ATR >       {
    typedef BuffThSC< T, SpSC<H,c,T>, m, IndRAW_HA > type;
    static const bool value = true;                       };

/****************** is_same_type conditional *****************/
template<class T1, class T2> struct is_same_type
    { static const bool value = false; };
template<class T           > struct is_same_type<T, T>
    { static const bool value = true;  };


/**************************** USpM ***************************/
template< class M, M* m, const int ATTR, class Cont,
          class BuffThComp = validSpM<M,m,ATTR>::type >
class USpM :public BaseUSpM< M::ElemType,M,m,BuffThComp,Cont > {
  private:
    typedef enable_if< is_USpM_inst<Cont> >::type V1;
    typedef enable_if<
      is_same_type< validSpM<M,m,ATTR>::type, BuffThComp >
    >::type V2;
};

struct Dummy{}; Dummy dummy;
template<> class USpM<Dummy, &dummy, 0, void*,  int>
    : public USpMnever<Dummy, &dummy, 0> {};
typedef USpM<Dummy, &dummy, 0, void*,  int> USpMfp;

/*********** Functor: extends all BuffarableThComps ***********/
template<class UM> struct TH_INHO                {};
template<>         struct TH_INHO<USpMfp>        {};
template<class M, M* m, const int ATTR, class Cont>
  struct TH_INHO< USpM<M, m, ATTR, Cont> >        :
     public validSpM< USpM<M, m, ATTR, Cont>, M, m >::type,
     public TH_INHO<Cont>                        {};

/* SpecThread<SELF,TM> is subtype of ThBuffComp<SELF, TM:USp> */
template<class SELF, class UM>
  class ThBuffComp : public TH_INHO<UM>        {  ... };
```

Figure 7: Template Meta-Programming 1

Figure 8 presents the functor that, for user's convenience, creates the thread manager class (see Figure 4). If the iteration policy is interleaved, the implementation statically checks that USpM does not contain any SpLSC instances. The `isSCin` functor answers this question. Note that this check cannot be performed with F-bounded polymorphism.

4.3 Meta-Programming: Performance

Figure 9 shows the implementation of the USpM `specLD` operation that will dispatch the call to the appropriate speculative model instance. The user hints that it is most likely that the template parameter `m1` is the right instance. The implementation checks this by calling `checkBounds`. If the test fails, the execution takes the slower path of checking the model instances in order for the proper address range (`specLDslow`). If no suitable range is found the default is a SpRO-type behavior – conservative approach. If the test succeeds (`index<WORD_MAX`) then `specLDrec` is called.

Static overloading resolution will find out that exactly one of the two `specLDrec` functions matches. This is because only one of the third, default parameter type is well formed (the `enable_if`, `disable_if` are called on the same argument, and hence cannot be both correct or wrong). The `is_same_object` "condition" is true *iff* `m1` and `m` are identical. Note that an optimistic `specLD` call on a model instance

```
/************ isSCin: is any SpSC instance in UM? *************/
template<class UM>struct isSCin{static const bool value=false;};
template<>struct isSCin<USpMfp>{static const bool value=false;};

template<class M, M* m, const int ATTR, class Cont>
 struct isSCin< USpM<M,m,ATTR,Cont> > : public isSCin<Cont> { };
template<class H,SpSCcore<H>* c, class T,SpSC<H,c,T>* m, int A>
 struct isSCin<SpSC<H,c,T>,m,A> {static const bool value=true;};

/********* Thread Manager Inheritance Functor TM_INH **********/
template<class UM, int UROLL, int IT_POLICY> struct TM_INH  { };
template<class UM,int U> struct TM_INH<UM,U,INTERLEAVED_IT,SC> :
     public ThreadManager_SC<UM,UROLL,InterleavedChunks<UROLL> >
  { typedef disable_if< isSCin<UM> >::type V; };
```

Figure 8: Template Meta-Programming 2

```
template<class T, class M, M* m, class Buff, class Cont>
class BaseUSpM                                                    {
  template< class TH > static void serial_commit(TH* th)      {
      m->serial_commit(static_cast<Buff*>(th), th->getID());
      Cont::serial_commit(th);                                }

  template< class M1, M1* m1, class T1, class TH >
  static T1 specLD( volatile T1* addr, TH* th )               {
      WORD index = m->checkBounds((T1*)addr);
      if(index<WORD_MAX)
           return specLDrec <M1, m1>(addr, th, index);
      else return specLDslow<T1, TH>(addr, th);              }

  template< class M1, M1* m1, class T1, class TH >
  static T1 specLDrec (
    volatile T1* addr, TH* th, const WORD index,
    disable_if< is_same_obj<M1, M, m1, m> >::type*=NULL
  ) { return Cont::specLDrec<M1,m1, T1, TH>(addr,th,index); }

  template< class M1, M1* m1, class T1, class TH >
  static T specLDrec  (
    volatile T* addr, TH* th, const WORD ind,
    enable_if< is_same_obj<M1, M, m1, m> >::type* =NULL
  ) { return m->specLD(addr,th->getID(),(Buff*)th,ind);     }  };
```

Figure 9: Optimistic (Static) Dispatch for specLD

that is not aggregated in the USpM will be statically faulted. The overloading resolution also fails if the types `T` and `T1` are not identical when `m1` and `m` are. When no unsafe casts exists in the user code, we can thus allow two models to service non-disjoint memory partitions, as long as their basic-types are different (now we can handle `typedef struct{int i; float f;} arr[N]`). Otherwise, TLS instances are applied at `WORD` level on disjoint intervals.

Finally, the hinted speculative instance is tracked down at compile time, while the unnecessary recursive calls to `specLDrec` are completely compiled away (the timming differences with the hand-optimized code are inconclusive). The thread implements a similar policy, hence the user may conveniently call the TLS operations on either USpM or thread.

5. PERFORMANCE RESULTS

All the tests referred to in this section were performed on a four-*Opteron*-processor SUN SMP machine running *Fedora Core 6*. We used the *gcc-3.4.4* compiler with `-O2` optimization level and the *pthread* library. Section 5.1 evaluates the speculative overhead on loops with trivial bodies (*gcc*'s profile-directed optimizations are used). Section 5.2 presents speed-up result for loop kernels from real benchmarks.

5.1 Load/Store Overhead on Null Benchmark

Table 1 shows the speculation overhead, computed as the inverse-ratio between the (sequential) timings of the code as written and with read and writes re-expressed as calls to `specLD` and `specST`, for four simple programs. Table entries are of the form `x/y` corresponding to the implementation approaches USpM/USpM-var introduced in Section 2.3.

	Model	size=10^3	size=10^4	size=10^5
LoopRO	SpRO	2.06/1.46	2.22/1.66	1.17/1.09
LoopRO	SpLSC	4.33/4.10	4.36/4.08	2.05/1.93
LoopWO	SpLSC	6.44/5.60	5.48/4.87	2.72/2.49
LoopRW	SpLSC	11.2/11.1	6.58/6.52	2.81/2.69
Figure 3	Both	9.83/8.67	4.95/4.47	2.19/2.00

Table 1: Read/Write (Sequential) Overheads.

```
/**LoopRO**/ for(int i=0; i<N; i++) { sum += C[i] + D[i]; }
/**LoopWO**/ for(int i=0; i<N; i++) { A[i] = i; B[i] = i; }
/**LoopRW**/ for(int i=0; i<N; i++)
             { d1=A[i]; d2=B[i]; A[i]=d1+d2; B[i]=d1-d2;  }
```

Figure 10: Null Benchmark for Overhead Evaluation

Our mini benchmark contains four programs. The first three are presented in Figure 10, while the fourth is our introductory example (Figure 3). In all programs, all array accesses have been protected with speculative support. The first three applications use a single model: either SpLSC or SpRO. The fourth uses two, as depicted in Figure 3. We used a hash function of cardinality 2^6 and a thread buffer of size 100. The size of the arrays was varied between 10^3 to 10^5. Since the speculative memory overhead is fixed (does not depend on the original-data size), increasing the array sizes will accentuate the memory-hierarchy related overheads of the original read/write operations, with the result that the (relative) speculative overhead decreases. With one exception, the figures seem to indicate just that. Note also that a speculative write is significantly more expensive than a read, due to value-buffering.

We have also observed that the overhead decreases faster for `LoopRW` than for `LoopRO` or `LoopWO`, even if these programs use the same data. Our guess is that, perhaps, the instruction-level cache is the issue here, as `LoopRW` uses both load/store speculative instruction while the other two use either load or store. The USpM-var approach overheads (2^{nd} number in each entry) are much reduced in several cases; if the interval prediction is very accurate, this technique may pay off (otherwise use USpM since a full-rollback cost is in the range of thousands of instructions).

Finally, we note that when the arrays' size reaches 10^5 elements, the speculation overhead is less than three, and hence we have potential speed-up on a four processor machine, even when all accesses require speculative support. The figures obtained for the loops involving store operations are deceptive: SpLSC is not scalable. Hence the ratio between the whole iteration cost and the cost of the writes per iteration is in effect a bound on the number of processors that may contribute to speed-up. We expect SpLSC to perform well when this ratio is high.

5.2 Speed-Ups on Non-Trivial Benchmarks

Table 2 shows the parallelization speed-ups, computed as the ratio between the sequential and parallel execution timings, obtained on several loops from the BYTEmark and SciMark benchmarks. We assumed that only privatisable variables are statically disambiguated, all global variables

Seq/Spec	Hand Par	SpLSC	+SpRO	+ 1% RR
IDEA Cipher	3.91	2.25	3.22	3.19
IDEA DeKey	3.89	1.85	3.09	3.00
SparMatMult	2.11	1.25	2.00	1.95
NeuralNetFW	2.04	1.19	1.90	1.86
NeuralNetBW	1.52	0.75	1.26	1.21
FFTtransf	1.99	0.83	0.83	0.80

Table 2: Parallelization Speed-ups on Loops from SciMark & BYTEmark Benchmarks (4 Processors)

(arrays) and parameters are protected with TLS support. The second column presents the *optimal* speed-up, where the program is hand-parallelized without speculative support. The third column shows the results obtained when SpLSC was employed alone. The fourth column refers to the case when the SpRO model was used for the variables that are very rarely written. The fourth column shows the obtained speed-up when we artificially introduce a 1% rollback ratio (w.r.t. the number of executed iterations). The speculative storage size for all applications is well under 1% of the original data size. Note that speculative-model composition attains between 80–93% of the optimal speed-up, while SpLSC in isolation reaches only the 47–60% level.

Finally, we observe that `FFTtransf` is not suitable for SpLSC, since it features an equal number of read and writes, little independent computation and a code-fragment that requires serial execution. In this case, the speed-up from SpLSC saturates at only two processors due to the non-serial commit phase (see Section 5.1 end). In comparison, the scalable SpLIP [17] model yields a healthy 1.83 speed-up.

6. CONCLUSIONS

The shortcomings of software-level solutions have motivated us to develop *PolyLibTLS*, a library that encapsulates several lightweight TLS models. It promotes *flexibility*: the most suited model is associated, based on code properties, with each variable, while a unified speculative model combines the various model instances.

This paper has presented one such TLS model and the high-level library structure. Since debugging multi-threading applications is notoriously difficult, we have shown how to use template meta-programming to statically verify certain invariants that cannot be expressed via F-bounded polymorphism. Furthermore, we have shown how to use inheritance-functors to make certain associations transparent from the user, thus significantly simplifying the framework's use.

A usable software-TLS framework must not compromise efficiency to achieve a modular and extensible design, especially not when the mapped operation granularity is a read/write instruction. We have presented an *optimistic* dispatching technique, well suited for a TLS solution, that, while being safe, reduces to a static and hence efficient dispatch when the guess is correct. The desired properties are thus preserved. Finally, we have shown results that demonstrate that our strategy of composing lightweight speculative models to parallelize an application is effective.

Acknowledgements

The first author would like to thank Stephen Watt for insightful discussions on this topic and the ORCCA lab at Western Ontario for providing access to its parallel machine.

7. REFERENCES

[1] V. S. Adve. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. *SC'95*.

[2] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *SPAA'02*.

[3] P. Canning, W. Cook, W. Hill, and W. Olthoff. F-Bounded Polymorphism for Object Oriented Programming. In *FPCA'89*, pages 273–280.

[4] M. K. Chen and K. Olukotun. Exploiting Method Level Parallelism in Single Threaded Java Programs. In *PACT'98*.

[5] M. K. Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *ISCA-30*, June 2003.

[6] M. Cintra and D. R. Llanos. Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In *PPoPP*, June 2003.

[7] J. O. Coplien. Curiously Recurring Template Patterns. In *C++ Report*, pages 24–27, 1995, February.

[8] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *IPDPS*, 2002.

[9] A. S. David R. Musser, Gillmer J. Derge. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley (ISBN 0-201-37923-6), 2001.

[10] A. Gurtovoy and D. Abrahams. The Boost MPL Library, http://www.boost.org/libs/mpl/doc/index.html.

[11] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for Chip Multiprocessor. In *ASPLOS'98*.

[12] J. Jarvi, J. Willcock, and A. Lumsdaine. Concept-Controlled Polymorphism. In *Generative Programming and Component Engineering*, volume 2830 of LNCS, pages 228–244. Springer Verlag, September 2003.

[13] I. H. Kazi and D. J. Lilja. Coarsed-Grained Thread Pipelining: A Speculative Parallel Execution Model for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 12(9), September 2001.

[14] S. W. Kim, R. E. Chong-Liang Ooi, B. Falsafi, and T. N. Vijaykumar. Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution. In *PPOPP'01*

[15] Sun Microsystems. MAJC architecture tutorial. In *White Paper*, 1999.

[16] N. C. Myers. Traits: a New and Useful Template Technique. In *C++ Report*, June 1995.

[17] C. E. Oancea and A. Mycroft. A Lightweight In-Place Model for Software Thread-Level Speculation. *In preparation.*

[18] M. Odersky and M. Zenger. Scalable Component Abstractions. In *OOPSLA*, pages 41–57, 2005.

[19] P. Papadimitriou and T. Mowry. Exploring Thread-Level Speculation in Software: The Effects of Memory Access Tracking Granularity. Technical report, CMU, 2001.

[20] P. Rundberg and P. Stenstrom. An All-Software Thread-Level Data Dependence Speculation System for Multiprocessors. *The Journal of Instruction-Level Parallelism*, 1999.

[21] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *ISCA-22*, pages 414–425, June 1995.

[22] J. G. Steffan, C. G. Colohan, A. Zhai, and T. Mowry. A Scalable Approach for Thread Level Speculation. In *ISCA-27*, 2000.

[23] T. Veldhuizen. Using C++ Template Metaprograms. In *C++ Report, Vol 7, No.4*, pages 36–43, 1995, May.

[24] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *OOPSLA*, pages 439–453, 2006.

[25] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Micro-35 Proceedings*. 2002.