Mixed Model Universal Software Thread-Level Speculation

Zhen Cao and Clark Verbrugge
School of Computer Science, McGill University
Montréal, Québec, Canada H3A 0E9
Email: zhen.cao@mail.mcgill.ca, clump@cs.mcgill.ca

***Abstract*—Software approaches to Thread-Level Speculation (TLS) have been recently explored, bypassing the need for specialized hardware designs. These approaches, however, tend to focus on source or VM-level implementations aimed at specific language and runtime environments. In addition, previous software approaches tend to make use of a simple thread forking model, reducing their ability to extract substantial parallelism from tree-form recursion programs such as depth-first search and divide-and-conquer. This paper proposes a Mixed forking model Universal software-TLS (MUTLS) system to overcome these limitations. MUTLS is purely based on the LLVM intermediate representation (IR), a language and architecture independent IR that supports more than 10 source languages and target architectures by many projects. MUTLS maximizes parallel coverage by applying a mixed forking model that allows all threads to speculate, forming a tree of threads. We evaluate MUTLS using several C/C++ and Fortran benchmarks on a 64-core machine. On 3 computation intensive applications we achieve speedups of 30 to 50 and 20 to 50 for the C and Fortran versions, respectively. We also observe speedups of 2 to 7 for memory intensive applications. Our experiments indicate that a mixed model is preferable for parallelization of tree-form recursion applications over the simple forking models used by previous software-TLS approaches. Our work also demonstrates that actual speedup is achievable on existing, commodity multi-core processors while maintaining the flexibility of a highly generic implementation context.**

Keywords*—*Thread-Level Speculation; Parallelization; Forking Model

I. Introduction

Thread-level speculation (TLS), or speculative multithreading (SpMT) is a safety-guaranteed approach to automatic or implicit parallelization. Speculative threads are optimistically launched at *fork points*, executing a code sequence from *join points* well ahead of their parent thread. Safety is preserved in this speculative model by buffering reads and writes of the speculative thread. Once the parent thread reaches the join point the latter may be *joined,* committing speculative writes to main memory and merging its execution state into the parent thread, provided no read conflicts have occurred. In the presence of conflicts the speculative child execution is discarded or rolled back for re-execution by the parent.

Thread-level speculation has received significant attention in terms of hardware development as a feasible technique for automatic parallelization [4], [17], [16]. Software-only designs have been proposed, and have the advantage of applying to existing, commodity multiprocessors, but the immediacy of application requires some tradeoff in terms of increased overhead and compilation complexity, with existing research efforts based on prototype, language-specific implementations [12], [10]. Realistic and convincing evaluation of such designs, however, requires consideration of a full compiler infrastructure, one that enables both deep investigation and application to a variety of compilation contexts.

Fundamentally, TLS approaches differ in terms of *forking models*: how they create and manage speculative threads. Two main forking models exist, *in-order*, and *out-of-order*, and existing software models have been primarily based on one or the other of these strategies, which allow for good exploitation of parallelism in loops and deep method calls respectively, as discussed in section II. These simple forking models, however, have limitations with respect to the ability to extract parallelism, and a reliance on pure in-order or pure out-of-order design limits the amount of parallelism that can be found in more complex programs, including ones that make extensive use of tree-form recursion, such as found in depth-first search and divide-and-conquer programs.

In this work we propose the Mixed-model Universal software-TLS (MUTLS) system to overcome both limitations of existing software-TLS approaches. First, MUTLS uses a *mixed* forking model to maximize the potential to extract parallelism in more general classes of programs. Second, MUTLS is universal in that it is language and architecture neutral. Our approach is to build a pure software TLS design using the popular LLVM compiler framework [1]. We integrate our design into LLVM's machine and language-agnostic intermediate representation (IR), enabling generic application of TLS to arbitrary input and output contexts. This has the advantage of providing a full and non-trivial compiler context for evaluating TLS, as well as allowing the full range of source and hardware pairings enabled by the LLVM framework.

Our design is demonstrated and tested by modifying front-ends for C/C++ and Fortran to support user-driven speculation. From this we are able to generate native executables (or JIT-based execution) for non-trivial benchmarks to evaluate performance, illustrating the potential of our approach as a means to explore and compare the use of TLS in different language contexts. Software TLS faces significant challenges in terms of balancing overhead concerns with the many possible design decisions possible in TLS implementation. Our system simplifies this research exploration by allowing for practical experimentation within an optimizing compiler context. This paper has the following specific contributions.

- We describe MUTLS, the first software-TLS implementation on a source language and target architecture independent

intermediate representation (IR). Our design is capable of adding TLS features to any LLVM input language and executes on any architecture supported by LLVM, significantly extending previous TLS systems which only support a single language context and/or architecture.

- We propose a tree-form, mixed forking model which incurs less cascading rollbacks than previous mixed forking models. We integrate it into the MUTLS system, demonstrating that complex mixed fork models can be hosted in a language and architecture independent software-TLS implementation.
- We perform a deep experimental analysis of performance using a programmer-directed approach to arbitrary point speculation. Our experiments demonstrate real speedup on both C++ and Fortran benchmarks, and show that a mixed model is preferable to in-order and out-of-order models for tree-form recursion applications. As far as we know, no previous software-TLS systems have experimented on tree-form recursion benchmarks.

II. Background on TLS Design and Fork Models

Traditionally, TLS approaches has been characterized by different *speculation models,* based on the selection of fork/join points. Loop-level speculation [4], [7], [17], [11] speculates on loop iterations, with loop iteration boundaries as fork/join joints. Method-level speculation (MLS) [3], [13] selects method (function) calls as fork points and speculates on their continuations. Arbitrary point speculation [16], [6] imposes no constraints on the selection of fork/join points. In principle all these forms are equivalent, although the required code transformations make conversion technically challenging.

Within any of these speculation models, different *forking models* can be used to define how the existence of multiple speculative threads is managed. Each of *in-order*, *out-of-order* and *mixed*, provides different choices, and has greater or lesser affinity for different program contexts.

In the in-order model, only the most recently speculated (most speculative) thread can fork a new speculative thread. This model is particularly appropriate for loop-based speculation, with future loop iterations forked in iteration order. The non-speculative parent begins the first iteration, forking a speculative child thread to execute the second, which can then fork a speculative grand-child to execute the third, and so on. Therefore, a typical scenario is that speculative threads are created in the order of their sequential execution: if the start of thread A would be prior to the start of thread B in sequential execution, then thread A is forked by its parent prior to B, and hence the name of this fork model. This model has the advantage that N threads can efficiently parallelize a loop of N iterations, but the disadvantage that if a speculative thread has to rollback all subsequently speculated threads should also rollback, as well as the constraint that parallelism not found in the most speculative thread may not be exploited.

The out-of-order model usually applies to method-level speculation. As the non-speculative parent thread enters a function call, a new thread is forked to execute the method continuation. This process can continue recursively, resulting in speculative threads being forked in the order that the parent descends into nested method calls, and so joined in the reverse order, as the parent returns from each call. This approach avoids concerns of "increasing" speculation found in in-order models, and easily applies to more arbitrary code constructions such as C++ nested block statements. The out-of-order model, however, has the disadvantage of limited parallelism on loop-level speculation since the non-speculative thread has to complete an iteration before reaching the fork point again to speculate another thread. The inability to launch speculative threads from speculative threads prevents more than one iteration from executed speculatively, bounding parallelism to just two threads irrespective of loop dependencies.

Mixed model is by far the most powerful forking model. It maximizes parallelism opportunities by allowing all threads to speculate new threads, and thus has the strength of both in-order and out-of-order models. One scenario in which it outperforms in-order and out-of-order models is tree-form recursion, where in-order speculation can only extract the top-level parallelism and out-of-order can only descend into one branch, while a mixed model theoretically can fork a whole tree of threads.

The flexibility of the mixed model also involves different designs. One part is how the system organizes the speculated threads. Previous mixed model systems organize them in a simple linear form, as a sequence of execution of the program. This design has a similar disadvantage to the in-order model: if a sequentially earlier thread rolls back, then all subsequent threads roll back even if they present no conflicts, which is not rare since function calls usually indicate independent tasks. The approach we develop here uses a novel mixed model that organizes the threads in a tree-form, and only has cascading rollbacks within its subtree.

III. Related Work

Many compiler frameworks have been proposed for thread-level speculation. These typically require significant hardware support, with the most efficient and modern designs involving different, hybrid forms of software and hardware cooperation.

The bulk of these works focus on loop-level in-order speculation, for which a number of feasible performance models have been proposed. The Java runtime parallelizing machine (Jrpm), for instance, identifies the best loops to parallelize by analyzing speculative loops with dynamic compilation and a hardware profiler [4]. Du et al. propose a misspeculation-based cost-model driven compilation framework to select effective loops for speculative parallelization [7]. STAMPede is a cooperative approach with unified hardware support for TLS [17].

Research on hardware MLS has suggested MLS is more amenable to unstructured parallelism, as is often found in method-heavy programming contexts, such as object-oriented languages [3]. This direction includes work on specific aspects of hardware MLS, such as determining appropriate fork heuristics [20]. MLS can work well with an out-of-order design, but for simplicity of hardware implementation these works usually assume in-order speculation.

Mitosis [16] is a mixed model arbitrary point TLS system. It uses a profiling-based estimation model to find fork/join point pairs called SP/CQIP. The order of a new speculation is defined to be just after the last thread speculated on the same SP/CQIP pair. POSH [9] is another mixed model TLS

system targeting both loop- and method-level parallelism. It requires the compiler to insert the fork/join points such that threads speculated by the same thread are joined in reverse order, which is used to assign the order of the speculative threads. Therefore it relies on a correct compiler control-flow analysis to distinguish nested structures such as function calls or loop-nests. Given the order of threads, these systems treat speculative threads the same as in-order speculation.

In most models thread joining is a highly linear process, with the rollback of one thread potentially causing cascaded rollbacks of all subsequent speculative threads in execution order. García-Yágüez et al. propose a mechanism to avoid such cascades, rolling back only threads that have consumed values from an aborted thread [8]. Our design for tree-based rollback also reduces these dependencies, more coarsely, but without the need to build a thread dependency matrix.

Hardware-centric approaches generally use compiler-based fork heuristics to find appropriate fork/join points. Of course the same underlying techniques can be applied through manual specification based on programmer directives [14], a general design approach adopted by most software approaches, as well as by us to simplify our current implementation.

A. Software-only TLS

With the absence of readily available TLS-specific hardware and growing ubiquity of commodity multiprocessors, pure software-based approaches have seen increased attention.

Focusing on out-of-order MLS in Java, Pickett and Verbrugge describe the SableSpMT software TLS system, based on an interpreted virtual machine context [12]. This effort aimed primarily at facilitating TLS analysis, but other works have focused directly on showing speedup. Ding et al. propose the in-order, arbitrary-point speculation system *BOP,* showing that even a coarse-grained strategy based on spawning system processes (rather than threads) can generate speedups of 2.08 to 3.31 for 3 SPEC CPU benchmarks on an 8-core Intel Xeon 7140M machine, while still providing safety [6]. Other, more TLS-specific approaches have also been proposed. Oancea and Mycroft present an optimistic C++ library for in-order software thread-level speculation SpLSC [10] as well as an in-place implementation SpLIP aimed at independent loops that rolls back for all WAW, WAR and RAW dependencies [11]. SpLSC and SpLIP achieve speedups of 0.09 to 5.86 and 1.44 to 5.88 respectively, for 7 applications from SciMark2, BYTEmark and JOlden benchmark suites on an 8-core AMD Opteron 2347HE machine.

Safe futures [19] is an explicit, transaction-based speculative parallelism approach for Java. Safe futures are like MLS except that the future thread can be explicitly claimed (joined) in the continuation thread by the programmer. Safe futures apply logical semantics to order the threads and join as in-order speculation.

While design of all these software and library approaches have informed our overall design, we extend them by developing a true cross-language, cross-platform design with a tree-form mixed forking model, fully incorporated into a compiler context. Table I provides a summary of the main approaches, how they differ, and where our design is situated.

TABLE I. COMPARISON OF TLS SYSTEMS

		Language	Forking Model	Speculative Region
Hardware	Jrpm [4]	Java	in-order	loop iteration
	SPT [7]	C	in-order	loop iteration
	STAMPede [17]	C	in-order	loop iteration
	Mitosis [16]	C	mixed (*linear*)	arbitrary
	POSH [9]	C	mixed (*linear*)	nested structure
Software	SableSpMT [12]	Java	out-of-order	method call
	Safe futures [19]	Java	mixed (*linear*)	method call
	BOP [6]	C	in-order	arbitrary
	SpLSC/SpLIP [10], [11]	C++	in-order	loop iteration
	MUTLS	arbitrary	mixed (*tree*)	arbitrary

B. LLVM

Our approach is built on the Low Level Virtual Machine (LLVM) [1]. LLVM is a popular framework for compiler research of various forms. Tristan et al., for example, present an approach to translation validation, verifying intra-procedural optimizations within LLVM [18]. Other work on parallelism has also used LLVM. Work on enforcing deterministic scheduling has been based on LLVM traces [5], and Prabhu et al. build a commutativity based programming extension on LLVM enabling multiple forms of implicit parallelism [15]. As far as we are aware, however, our work is the first to use LLVM for speculative parallelism, although there has been recent work proposing to use LLVM to target IBM's BlueGene/Q TLS architecture [2].

IV. DESIGN

MUTLS is purely based on the LLVM intermediate representation (IR). LLVM is a compiler framework that allows for multiple source languages and target architectures through the use of a generic, Static Single Assignment (SSA) based IR. This intermediate form is a type-safe, expressive assembly code which can be regarded as a universal abstract machine capable of representing all high-level languages.

There are many analyses and optimizations in LLVM. Each pass can specify its required and/or preserved analyses, so transformation passes can use analysis information and assume the IR already underwent specific transformations. There are two sorts of passes: LLVM IR based passes and Machine Function passes. Transformations of the former are purely based on the LLVM IR—i.e. from well-formed LLVM IR to well-formed LLVM IR. Machine Function passes are performed in the code generator for specific target architectures. The approach within this paper implements an LLVM-IR based pass and thus is inherently target independent.

The design of the system involves changes to LLVM back-end and its front-ends. The latter are minor, allowing easy portability of multiple language. Most of the complexity resides in the back-end, where we require code to support forking, buffering, joining and commit or rollback.

A. Front-End Design

Two built-in functions are added to the front-end for the user to specify fork and join points. These two functions have an argument p, which denotes the id of the fork/join point. Multiple fork points can have the same id while the join points cannot. Threads speculated at a fork point start execution from the join point with the same id. The fork point function has another argument *model* to specify the forking model of the

fork point. In order to avoid unnecessary rollbacks, we also add a *barrier point* which barriers the speculative thread if it is not in nested function calls (i.e. it is at the same level as was speculated). These built-in functions are transformed to the corresponding LLVM IR built-in functions, which are then processed by the back-end *speculator pass*. Examples of input C and Fortran programs are given in Figure 1.

```
void work(□ ) {
   □
   __builtin_MUTLS_fork(p, model);
   S1;
   □
   __builtin_MUTLS_join(p);
   S2;
   □
   __builtin_MUTLS_barrier(p);
   S3;
   □
}
```

(a) C program

```
subroutine work(□ )
   □
   call MUTLS_FORK(p, model)
   S1
   □
   call MUTLS_JOIN(p)
   S2
   □
   call MUTLS_BARRIER(p)
   S3
   □
end subroutine work
```

(b) Fortran program

Fig. 1. User-directed TLS source code. Before S1 the parent thread forks a speculative thread to execute S2, and synchronizes with it once it reaches that point. The speculative thread will stop before S3 if it reaches that point.

B. Back-End Overview

Our support for TLS in LLVM consists of two main parts: an LLVM speculator transformation pass, and a TLS runtime library. The LLVM speculator pass modifies the incoming IR based on the annotated fork and join points, delegating more complex behaviours to the TLS runtime library.

The TLS runtime library provides four modules: ThreadData, GlobalBuffer, LocalBuffer and ThreadManager. A ThreadData module maintains the status of a speculative thread, while GlobalBuffer and LocalBuffer modules manage individual buffering of global and local variables for speculative threads. The ThreadManager module maintains for each CPU one ThreadData, one GlobalBuffer and one LocalBuffer object, and interacts with the LLVM speculator pass.

The TLS runtime library is written in C++, and can be compiled into native static or dynamic libraries for linking into any executable. However, to enable further optimizations provided by LLVM such as inlining of library calls, we compile it into a bytecode library to link with the speculated LLVM bytecode of the source program.

The following subsections discuss the various steps involved in modifying the LLVM IR for TLS execution. We first present the basic code preparation, and then trace through the implementation design from fork to join, as well as give further details on memory management.

C. Preparation for Speculation

Target dependent information such as access to the actual instruction pointer (IP) and stack frame are generally not available in LLVM IR, nor are some low-level operations such as the ability to directly jump to another function. Maintaining the LLVM SSA-form after each transformation is also required. These issues make thread-level speculation for LLVM more difficult. We perform a number of IR-based code transformations to support TLS. These transformations are required for each function with annotation of fork and join points, as well as their nested function calls.

For each such function we perform 4 basic modifications. We need to (1) generate a speculative version of the function, (2) generate helper functions for interaction with the TLS runtime library, (3) split and number the basic blocks at appropriate points for synchronization between the speculative and non-speculative versions of the function, and (4) assign local buffering addresses (offsets from stack pointer) for local variables. Below we detail these steps.

(1) We first clone the function and add two integer parameters *counter* and *rank* to generate the *speculative function*. These arguments are used to direct the code entry to the correct starting point and track the CPU index. To ensure safety, every load and store operation within the function is replaced by a TLS runtime library call to `MUTLS_load_{int32, int64,` etc} and `MUTLS_store_{int32, int64,` etc}. Note that since we inline the TLS runtime library into the source LLVM code, these function calls are reduced to more efficient direct memory accesses.

(2) We generate a *stub* function with the suffix ".stub" as the entry point for speculative threads. This prologue function fetches arguments of the speculative function through the `MUTLS_get_regvar_{int32, ptr,` etc} library calls, and then calls the function with the arguments. These arguments are stored by the non-speculative parent thread in a generated *proxy* function, which has the same signature as the speculative function and stores the function arguments to the LocalBuffer object by the `MUTLS_set_regvar_{int32, ptr,` etc} library calls. The proxy function then calls the `MUTLS_speculate` library function with the stub function address to fork a new thread.

(3) Speculative termination and synchronization require a number of checkpoints to be inserted into the code. At each annotated fork point the basic block is split to generate a *speculation block*, and at each join point annotation the basic block is split to generate a *join point block*. Speculation is necessarily terminated at each external, indirect and exception handling function call through a *terminate point block* (other than for known, safe external calls such as *abs*, *log*, etc), and prior to the method return point through a *return point block*. Before each internal function call, a basic block is split to generate an *enter point block*, and inside each loop a block is split to generate a *check point block*. Except for the speculation block, all these blocks are potential *synchronization blocks*, and are numbered by the *synchronization counter* starting from 1, which is then used to build the speculation and synchronization tables, as discussed in subsections IV-D and IV-E.

(4) Registers cannot be used to transfer data between threads. Thus for each local (register and stack) variable live at the beginning of a synchronization block of the function, we allocate and assign an offset so we can save and restore such data through the runtime library at speculation and synchronization points.

A schematic of the transformation is given in figure 2. The following sections will discuss it in detail. We use C pseudo code instead of LLVM assembly for compactness.

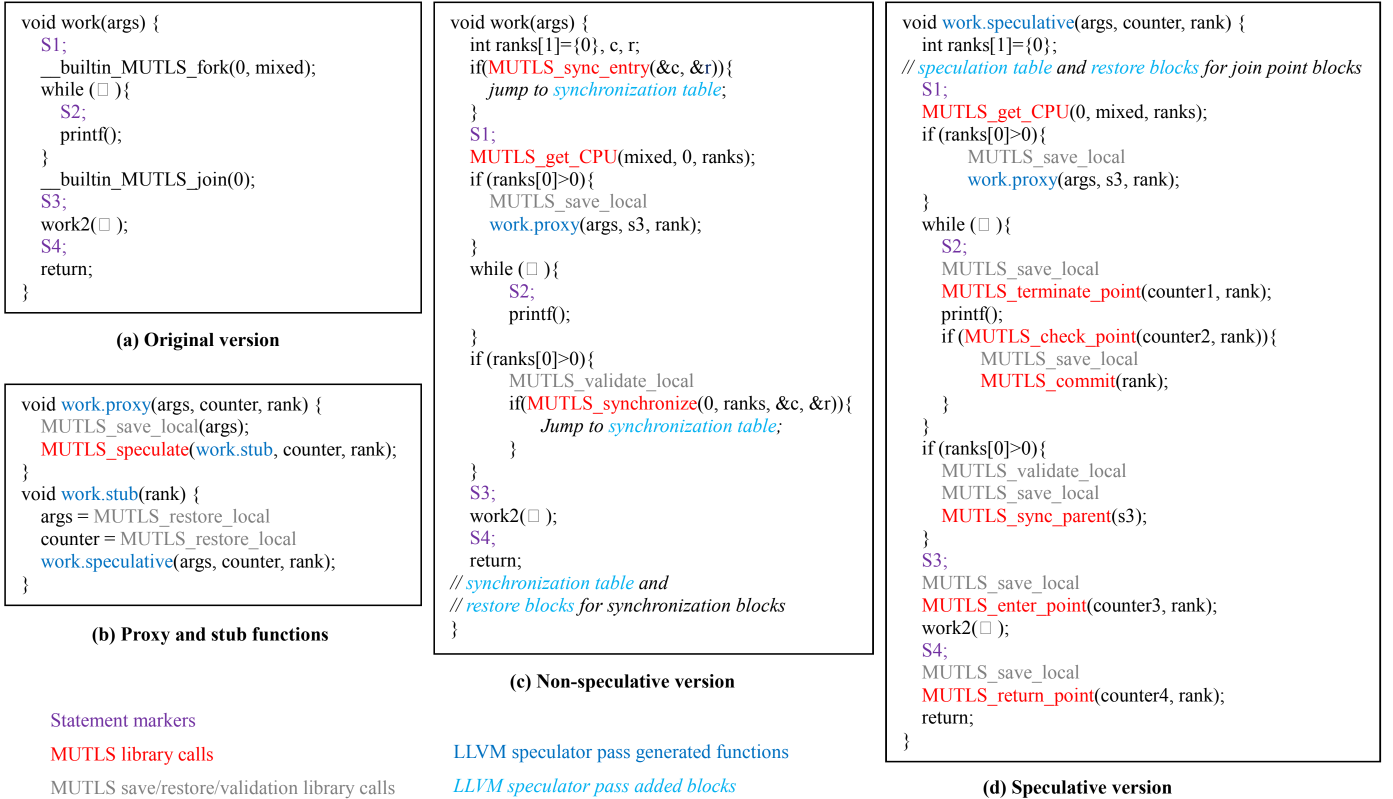

Fig. 2. Transformations performed by the Speculator Pass

D. Fork

Speculative threads are intended to be bound to virtual CPUs. Each virtual CPU is identified by *rank*, which is an ID number from 1 to the total number of CPUs. Since each function may contain multiple fork/join points, the speculator transformation pass allocates an integer array *ranks* on the stack to store the ranks of the child threads speculated in the current function frame. The length n_{ranks} of *ranks* is the number of join points in the function. At most one thread can be speculated on at each fork/join point id; if a fork/join point id is not speculated on, the corresponding entry in *ranks* is 0. This design allows multiple threads to be speculated on the same join point at different stack frames of the same function, enabling the ability to extract substantial parallelism from recursive function calls.

Each CPU can be RUNNING, IDLE or READY_TO_RECLAIM, and are initialized IDLE at the beginning of program execution. At each fork point, the LLVM speculator pass generates a library call `MUTLS_get_CPU` to assign a rank to the speculative thread, passing the forking model, fork/join point id and the *ranks* array. If no CPU is IDLE, speculation will not be performed; otherwise the child rank is stored in the corresponding entry of *ranks* and control is branched to the speculation block. The speculation block saves the local variables live at the beginning of the join point block as will be discussed in subsection IV-G4, and then forks a speculative thread by calling the proxy function generated in preparation step (2). In the proxy function `MUTLS_speculate` initializes the ThreadData object and sets the CPU state as RUNNING. The resulting speculative thread retrieves this data within the stub function, and then enters the actual speculative code.

The speculative entry point is conceptually somewhere within a function, but since LLVM does not allow branching directly to this starting point some gymnastics are performed to redirect entry control flow. For this we use the *speculation table*, implemented as a *switch* LLVM instruction that directs incoming control flow to the block indicated by the *counter* argument. A 0-counter indicates normal entry, while a non-0 value indicates some internal starting point. This approach bypasses the inability of LLVM to branch directly into a function, and also allows both initial and any subsequent (such as recursive) calls to the speculative function to coexist. Upon initial entry, local variables need to be initialized to the same values found in the non-speculative function at forking. For this we fetch the values previously stored within the runtime library and assign them to the corresponding local variables. This process is slightly complicated by the fact that LLVM IR is in SSA form, and so trivially re-assigning register variables is not possible. We thus add a separate *restore* block to assign the local values, and then branch into the actual entry point. Phi nodes are inserted at the beginning of the latter block to distinguish the different versions of the register variables.

E. Join

At each join point, the LLVM speculator pass adds instructions to check if a thread was speculated on the join point; if so it synchronizes with the speculative thread. This process is encapsulated by `MUTLS_synchronize`, which returns true/false if the speculative thread commits/rollbacks. If true is returned, the synchronization counter and the speculative thread rank

are returned by the arguments *c* and *r*, and control branches to the *synchronization table*. The synchronization table itself is a *switch* LLVM instruction that branches to the code blocks indexed by synchronization counter. Unlike the speculation table which has $n_{ranks} + 1$ entries, the synchronization table has an entry for each possible synchronization block.

A speculative thread needs to terminate if it may execute instructions unsafe to perform speculatively, or if the parent thread is waiting to join with it. The former case is enforced by adding a no-return runtime library call `MUTLS_terminate_point` at each terminate point block. The latter case, however, requires polling to determine whether or not the speculative thread needs to stop. This is implemented through calls to `MUTLS_check_point`, which are inserted prior to function calls and within inner loops to ensure that the non-speculative thread need not wait overly long. In either case, if validation fails during synchronization, the speculative thread rolls back within these functions; otherwise, it commits and passes in its synchronization counter and rank to indicate its continuation point and to identify itself. Live local variables needs to be saved in order for the non-speculative thread to restore the values after committing. For performance, this is not done before the `MUTLS_check_point` as check points are entered frequently. Instead, the speculator pass adds a *commit block* at each check point, which saves the local variables and calls `MUTLS_commit` to complete the commit process.

Efficient thread polling and synchronization is performed using a simple flag-based barrier. The ThreadData object for each speculative thread maintains two volatile variables: `sync_status` which is set to SYNC if the speculative thread is notified to synchronize, and `valid_status` which is set to COMMIT or ROLLBACK after validation and global buffer commit/rollback. Both variables are initialized to NULL during the fork process. When `MUTLS_synchronize` is called, the non-speculative thread locates the corresponding ThreadData object as discussed in subsection IV-F. It then sets `sync_status` to SYNC and busy-waits for the `valid_status` to be non-NULL. The speculative thread busy-waits for `sync_status` to be SYNC if it enters a terminate point, or simply returns if it enters a check point and encounters a NULL. Once both threads have stopped the speculative thread can validate and commit/rollback.

F. Mixed Forking Model

The mixed forking model assumes that the direct children of a thread follow the out-of-order model; that is, a later speculated thread represents logically earlier sequential execution. It also assumes that a thread subtree represents a continuous interval of execution with its root representing the earliest logical execution, and that different thread subtrees represent disjoint intervals of execution. As a result, a "reverse in-order traversal" of the thread tree follows the sequential execution order. This assumption is similar to previous work [19], [9]. However, unlike their approaches, the runtime system does not rely on the mixed-model assumption to be correct.

Each thread maintains a *children* stack storing the ranks of its direct children. When forking a thread, it pushes the new thread rank into *children*. In the `MUTLS_synchronize` library call, it pops a child rank from *children* and checks if it is equal to the corresponding rank stored in the *ranks* array. If it is not, then it means the program did not follow the mixed-model assumption. In this case, it sets the `sync_status` of the child thread to be NOSYNC and the process continues until the rank is found or *children* is empty. In either case, the corresponding entry in *ranks* is set to 0 to allow speculation on that point again. If *children* is empty, the child thread has already rolled back and false is returned. Otherwise, it appends the *children* of the child thread to the *children* of the non-speculative thread, synchronizes with the child thread and returns the rank in the argument. Note that even if the child thread is invalid and requires rollback, its children are still preserved in the *children* of the non-speculative thread. This process is different from previous software approaches, having the advantage that local conflicts do not incur global rollbacks.

G. Memory Buffering

Non-local (static and heap) and local (register and stack) variable accesses of a speculative thread are buffered in its GlobalBuffer and LocalBuffer objects, respectively. GlobalBuffer is flat since non-speculative and speculative threads share the same address space (the same addresses for non-local objects). The LocalBuffer is organized as an array of stack frames, with each frame containing a RegisterBuffer and StackBuffer for storing register and stack variables.

1) Address Space Registration: Memory buffering should guarantee that invalid addresses are not accessed. Also required is to identify whether an address is in GlobalBuffer (non-local and non-speculative stack variables) or on the speculative stack (speculative stack variables). We solve the problem with an address space registration mechanism. The address space (the start and the end addresses) of each static and heap object is registered in the runtime library during the creation and deletion of the object, that is, at the beginning of program execution for a static object and memory allocation and deallocation for a heap object. Adjacent spaces can be merged to improve performance. The stack address space of a thread are the addresses between its base and current stack pointers, and is registered in its LocalBuffer object. A speculative thread is rolled back if it reads/writes an address not in the global and local address spaces.

The current implementation of heap memory registration is to intercept language-specific memory management library calls in LLVM IR, for example, "malloc" in C, "_gfortran_internal_malloc" in Fortran and "_Znwm" in C++. We do not allow speculative threads to allocate/deallocate memory as they may roll back. We realize that this approach is somewhat ad-hoc and difficult to deal with customized allocators. We plan as future work to solve the problem by hooking into OS system calls or handling page fault signals.

2) Global Buffer: Each GlobalBuffer object maintains two maps: a read-set and a write-set, with writes to the global address space redirected into the write set. Global loads either return the value from the write-set if found there, from the read-set if previously read, or by loading the value from memory and saving it in the read-set (first time).

Conflicts only occur when a speculative thread reads data from an address before the non-speculative thread writes data to the address. Therefore, the validation process iterates

through the read-set of the GlobalBuffer object, comparing data with the corresponding values in main memory; if they are not equal, then validation fails and the buffer is discarded. Otherwise, validation succeeds and the data in the write-set is committed.

As the addresses of the global read and write operations can be arbitrary, and there may be an arbitrary number of read and write operations, the read and write-set maps must be efficient. Normal hash maps frequently increase in size as data is inserted, causing dynamic memory allocation and deallocation. Our design is instead to use static memory. The map has a byte array *buffer* of a multiple of the WORD size, a pointer array *addresses* and an integer stack *offsets*, all containing a maximum of N elements. The two arrays together implement a hash map while the stack guarantees that validation, commit and finalization operations of threads accessing a small amount of data are fast.

The *addresses* are initialized zero at the beginning of program execution. Given an address, the find/insert operation uses an efficient hash function to calculate its *offset* in *buffer* as the lower bits of the address, and the array *index* in *addresses* as the offset divided by the WORD size. Then *addresses[index]* is checked; if it is zero, meaning an empty slot, the address is inserted into the *addresses* array, data of WORD size is inserted at *buffer+offset*, and the offset is pushed onto *offsets*; otherwise, if *addresses[index]* equals the address, meaning the address has been inserted, then the data is accessed in *buffer*, otherwise the hash buffer is conflict. In the buffer conflict case, we store the address and data in a temporary buffer and the speculative thread will wait to be joined at the next check point. If the temporary buffer is used up, the speculative thread rolls back, although this is rare since check points are entered frequently. During validation, commit and finalization, *offsets* is traversed to find addresses and data accessed by the speculative thread.

Different size data accesses can be encountered. Assuming a read/write of *data* of *size* size at address *p*, the MUTLS memory buffering supports read/write operations if *size* is larger than, equal to, or smaller than WORD, given that one of *size* and WORD is a multiple of the other, and that *p* is aligned by *size*. If *size* is larger than WORD, we split the address into several WORD pointers and split *data* before write or reconstruct *data* after read operations. To support the case that *size* is smaller than WORD, a byte array *mark* with the same size as *buffer* is needed. First, a normalized address *np* is calculated by making the lowest WORD bits of *p* 0. If *addresses[index]* equals *np*, then the data is read/written at *buffer+offset*, and *size* bytes from *mark+offset* are set to `0xFF` if it is a write; otherwise, if it is an empty slot, then WORD bytes of data are read from *np* and written to the buffer and *size* bytes from *mark+offset* are set to `0xFF` if it is a write; otherwise, it is a buffer conflict. Validation of the read-set validates all read data, while commit of the write-set only commits data marked by *mark*. An optimization for commit is that if WORD size data of *mark* is -1, then the WORD bytes of data in *buffer* can be committed all at once.

3) Local Buffer: The LocalBuffer is used to transfer local (register and stack) variables between parent and child threads during fork and join though `MUTLS_(set|get)_(regvar|stackvar)_*` library calls. At preparation step (4) of subsection IV-C, the speculator pass assigns an offset for each register and stack variable. `MUTLS_(set|get)_regvar_*` passes the offset and the register value to the RegisterBuffer object, which in turn stores the register value in a static array. If there are too many variables and the assigned offset exceeds the array size, the speculator pass reports an error and speculation fails. `MUTLS_(set|get)_stackvar_*` is a similar case for stack variables, except that they also pass the address and size of the stack variable, and copy the stack data.

The above is complicated by the potential presence of stack pointers. If such a pointer is used in a speculative context, and the thread commits, the pointer will be invalid since the speculative version of the stack variable no longer exists. Instead, it should point to the non-speculative version. We propose a pointer mapping mechanism to solve the problem. During commit of pointers through `MUTLS_get_(regvar|stackvar)_ptr` calls, the value of the pointer is checked and if it is in the stack address space of the speculative thread, it is mapped to point to the corresponding variable in the non-speculative thread. Since the non-speculative and speculative functions may have different stack layouts, we cannot use a constant offset for mapping of all variables. The implementation thus records stack variable addresses in a hash map during `MUTLS_(set|get)_stackvar_*` calls, and calculates the offset of the pointer value. A complication to this occurs when type-casts between pointers and integers are present in a function—the pointer mapping mechanism may be unsafe as integer values may be used in various instructions including I/O. Our current implementation is to disallow pointer-integer type-casts unless the value is inside the global address space that is not mapped. Before type-cast instructions between pointers and integers, `MUTLS_ptr_int_cast` library calls are inserted which barrier the speculative thread if the pointer/integer value is not in the global address space.

Stack variable loads/stores in nested frames of a speculative function directly access the function's stack, since they do not affect the non-speculative thread and the stack acts as buffer itself. These variables are committed (copied) to the stack of the non-speculative thread by `MUTLS_get_stackvar_*` calls. Stack variables at the bottom frame are accessed as non-local data, the non-speculative version of which are buffered in GlobalBuffer during `MUTLS_(load|store)_*` calls.

4) Register Variable Validation: Local register variables live at the beginning of the join point block need to be initialized as they would be when normal execution reaches the join point. If the variable is not live at the fork point, then it should be predicted by the forking thread, otherwise the speculative thread retrieves an uninitialized value. Induction variables and expressions can also be made live by code transformation. When the forking thread reaches the join point, it should validate that the live local variables were correctly speculated, which is implemented by the `MUTLS_validate_local_{int32, ptr, etc}` library calls. The speculative thread will rollback if this validation fails.

H. Stack Frame Reconstruction

For simplicity we currently restrict speculative threads from returning from their entry function, but do allow them to call

and enter new functions. Even this is non-trivial, however, since the non-speculative parent thread may then need to reconstruct equivalent stack frames as part of a successful join, but stack frames are not available at the LLVM IR level, and may also differ in layout due to the extra parameters added to speculative versions. We propose a stack frame reconstruction scheme to address this problem. First, we need to explicitly track stack frames as the speculative thread descends into a call chain. At each enter point block, `MUTLS_enter_point` is called to register a new stack frame in the LocalBuffer object for the nested function call. This is matched at each return point block, where `MUTLS_return_point` is called to pop the stack frame. (Note that this checks to ensure the speculative thread is returning from a nested function call, and not its entry point.) The non-speculative parent must then generate a corresponding call chain, restoring frame data as it descends. This is initiated by the synchronization process, and fully enabled by a library call `MUTLS_synchronize_entry` inserted at the top of each non-speculative function reachable from a speculative one. This function inspects the LocalBuffer object, recognizes the existence of further frames, and if so restores the current frame data and directs the thread to the correct call point in the current function. This process continues until the non-speculative thread has replicated the entire call chain and frame state.

V. Experimental Results

We implement the MUTLS[1] system on llvm 2.9 with the llvm-gcc-4.2.2.9 front-end. For experimentation we used an AMD Opteron 6274 machine with 64 2.2GHz processor cores (4×16-core, 8×2MB L2 cache) and 64GB memory. The operating system is 64-bit Red Hat Enterprise Linux.

The benchmarks we experiment with are summarized in Table II. 3x+1 computes the well-known 3x+1 problem that avoids memory access during the computation, and thus serves as an idealized benchmark for our software-TLS system. We include 4 tree-form recursion applications to evaluate the mixed forking model: fft, matmult, nqueen and tsp. Matmult is a block-based matrix multiplication like Strassen's algorithm. This gives us a mix of CPU- and memory-intensive computation. Note that the computation/memory intensiveness is not characterized by the total memory used, but by the memory access frequency (density), defined as the number of read/writes divided by the program runtime $\rho = N_{rw}/T$. Thus, although matmult has time complexity $O(N^3)$ and space complexity $O(N^2)$, it is still considered memory intensive.

A. Speedup

Given the execution time of a parallelized program on N cores T_N, and of the original sequential program T_s, the absolute speedup is defined as T_s/T_N. The results of computation- and memory-intensive applications are shown in Figures 3 and 4, respectively. The runtime of each benchmark is determined by the arithmetic mean over 10 runs.

As expected, the computation-intensive benchmarks perform much better than the memory-intensive ones. We achieve absolute speedups of 51.8, 33.6, 31.9 and 49.0, 31.9, 19.5 for 3x+1, mandelbrot and md of C and Fortran programs,

[1]MUTLS is available online at http://www.sable.mcgill.ca/~zcao7/mutls

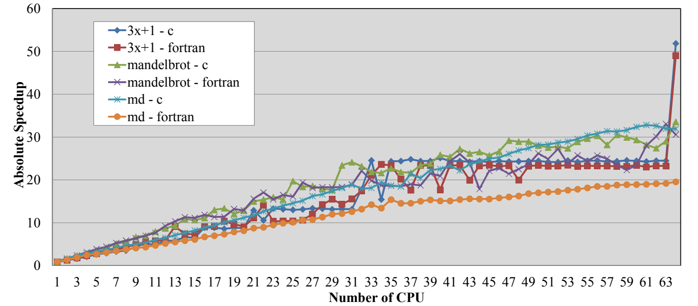

Fig. 3. Performance of Computation-Intensive Applications

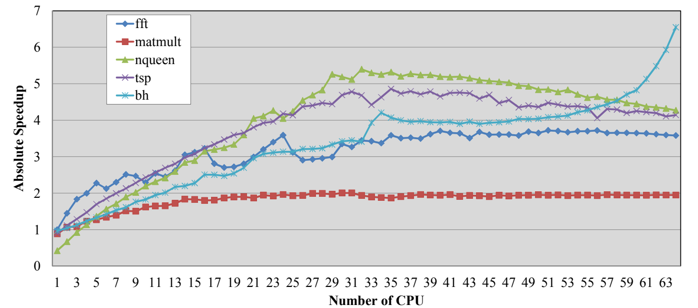

Fig. 4. Performance of Memory-Intensive Applications

respectively. The lower scalability of the Fortran programs is mainly because of their additional memory buffering overhead, e.g., the shapes of arrays being allocated on the stack. The reason that speedups of 3x+1 between 32 and 63 cores are generally stable and jump up at 64 is our workload distribution strategy, which splits the computation into 64 loop iterations, and thus at least two iterations are computed sequentially.

For the memory-intensive benchmarks, we achieve maximum speedups of 3.72, 2.01, 5.40, 4.86 and 6.55 for fft, matmult, nqueen, tsp and bh, respectively. For fft and matmult, the small threads speculated in deeper recursive calls cause significant amount of idle time, as discussed in subsection V-B. Larger problem sizes may relieve the problem, although a larger amount of data also requires more memory buffering for the speculation not to overflow, which would result in longer startup times for the programs.

B. Analysis of Parallel Execution

We are more concerned with the non-speculative thread since (1) it is on the critical path, (2) it does not rollback, and (3) it is the fastest thread, since it does not have speculative buffering overhead. As a result, we define two indicators: *critical path efficiency* which is the useful work time divided by the runtime of the non-speculative thread $\eta_{crit} = T_{worktime_nonsp}/T_{runtime_nonsp}$, and *speculative path efficiency* which is the sum of the work time divided by the sum of the runtime of the speculative threads $\eta_{sp} = T_{worktime_sp}/T_{runtime_sp}$. The two efficiencies are illustrated in Figures 5 and 6.

3x+1 and mandelbrot have almost perfect critical path efficiency and the highest speculative path efficiency, which corroborates their highest speedups. md has speculative path efficiency close to that of 3x+1 and mandelbrot, but its

TABLE II. BENCHMARKS

Benchmark	Description	Amount of Data	Pattern	Language	Characteristics
3x+1	3x+1 problem in number theory	40M integers (enumerate)	loop	C/Fortran	Computation intensive
mandelbrot	mandelbrot fractal generation	512×512 image, maximum 80000 iterations	loop	C/Fortran	
md	3D molecular dynamics simulation	256 particles, 400 iteration steps	loop	C/Fortran	
bh	Barnes-Hut N-body simulation	12800 bodies	loop	C++	Memory intensive
fft	recursive Fast Fourier Transform	2^{20} doubles	divide and conquer	C	
matmult	block-based matrix multiplication	1024×1024 matrices	divide and conquer	C	
nqueen	N-queen problem	14 queens	depth-first search	C	
tsp	travelling sales person (TSP) problem	12 cities	depth-first search	C	

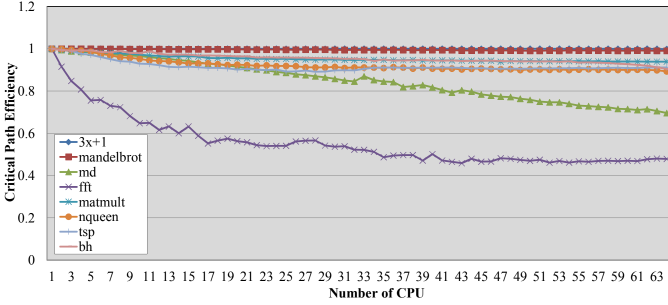

Fig. 5. Critical Path Execution Efficiency

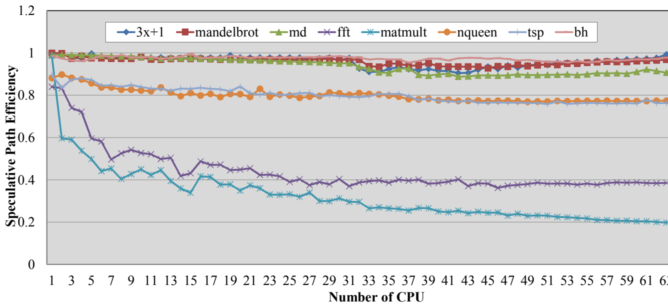

Fig. 6. Speculative Path Execution Efficiency

critical path efficiency drops continually, which explains its lower speedups. An interesting fact is that the critical and speculative path efficiency of nqueen and tsp are nearly the same all the time, showing similar characteristics of the two benchmarks. matmult has the third highest critical path efficiency of 94%-100%, illustrating the benefit of data reuse (spatial locality) of efficient memory buffering using the block-based approach. However, its low speculative path efficiency significantly weakens its speedups.

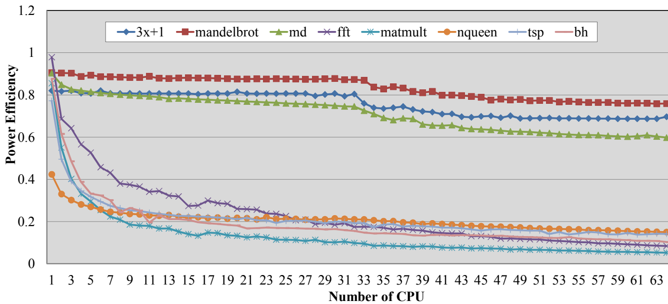

Fig. 7. Power Efficiency

Power consumption is a major concern in modern systems as well. Although we cannot measure power usage precisely, we can approximate power *efficiency* as the sequential runtime divided by the sum of the runtime of the non-speculative and all speculative threads $\eta_{power} = T_s/(T_{runtime_nonsp} + T_{runtime_sp})$, giving us an inverse measure of relative waste. This is shown in Figure 7. As expected, the computation-intensive applications are much more power efficient, up to 60% to 76% at 64 cores. Among the memory-intensive ones, nqueen and tsp have the highest power efficiency of 15% and 14%, respectively, followed by bh (10%), fft (8.4%) and matmult (5.3%).

The parallel execution coverage is defined as the sum of the speculative path runtime divided by the critical path runtime $C = T_{runtime_sp}/T_{runtime_nonsp}$. All our benchmarks have significant parallel execution coverage of 23.1 to 60.7, as expected of the mixed forking model.

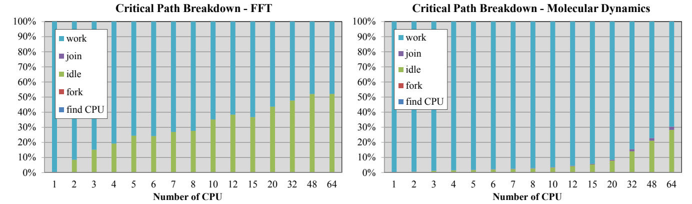

Fig. 8. Critical Path Breakdown - fft and md

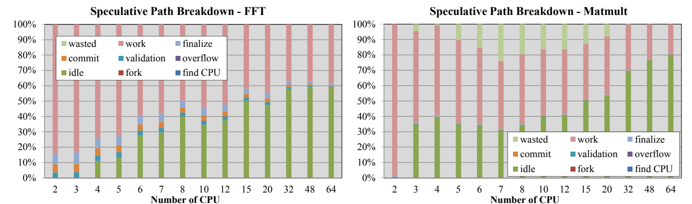

Fig. 9. Speculative Path Breakdown - fft and matmult

To further understand the overhead, we breakdown the executions of the least efficient benchmarks: fft and md for the critical path, and fft and matmult for the speculative path. The results are presented in figures 8 and 9. It can be seen that almost all overhead of the critical path is the idle time spent on synchronizing with the speculative threads, mainly waiting for them to validate and commit their memory buffers. The speculative path is more interesting. For fft, validation, commit and finalization accounts for 17% of the total speculative runtime with few cores, decreasing as more cores are used. Idle time accounts for most its time, however, up to 59% at 64 cores; this is partly because we fork a thread to execute the second recursive call and barrier it after the call, preventing it from accessing parent data and causing unnecessary rollbacks. In our experiments, matmult is the only benchmark that exhibits rollbacks, which start to occur from 3 cores and peakat 23%

at 7 cores. Though we split the computation into 4 sub-tasks each multiplying one sub-matrix, if the sub-tasks split their own sub-tasks, then different “sub-sub-tasks” may read/write the same data and cause rollbacks. Like fft, though, idle time still dominates, again due to speculative execution barriers.

C. Comparison of Forking Models

A comparison for different forking models is illustrated in figure 10, normalized to the speedup of the full mixed model. With more than 8 cores, the mixed model beats in-order and out-of-order models for almost all benchmarks. The only exceptions are in-order nqueen from 8 to 20 cores and tsp from 12 to 14 cores. With less than 8 cores, generally the DFS benchmarks perform better under a mixed-model while divide-and-conquer ones prefer in-order or out-of-order models.

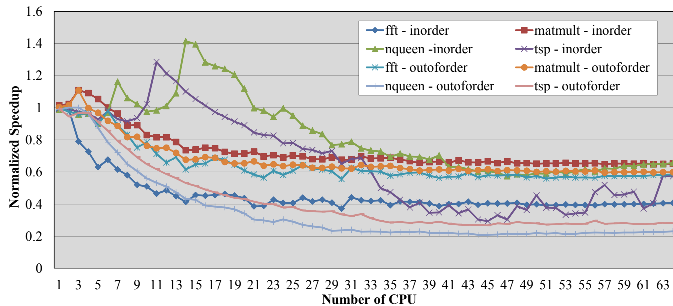

Fig. 10. Comparison of Forking Models

D. Rollback Sensitivity

Most of our benchmarks (other than matmult) do not generate rollacks due to their embarrassingly parallel properties. Here we intentionally make the MUTLS system randomly cause rollbacks with specific probabilities in order to see the effect on performance. We characterize this as Rollback Sensitivity, which is the relative slowdown with respect to the non-rollback scenarios. The results are shown in figure 11. It can be seen that programs with better speedups are more sensitive to rollbacks, especially for low rollback probabilities. We also note that for most memory-intensive workloads, 5% rollbacks can preserve at least 70% speedups.

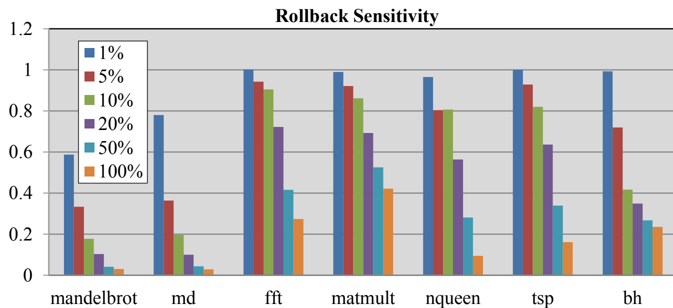

Fig. 11. Rollback Sensitivity

VI. Conclusions and Future Work

The complexity of even a relatively plain implementation of TLS makes exploration of the many possible design choices difficult, a problem exacerbated by the potential for interaction with other aspects of an optimizing compiler and execution environment. The MUTLS system is intended to improve that situation, providing a full-featured TLS compiler framework that accommodates a wide variety of input and output contexts. With a mixed forking model, MUTLS has more potential to extract parallelism from tree-form recursion applications.

Future work involves more fully fleshing out the design, adding in features and capabilities that enable deeper exploration of different aspects of TLS. This includes value prediction, different automatic fork heuristics, as well as other improvements to the interface that simplify usage. A direct and very practical application of our TLS design is for ready incorporation into the execution context of more dynamic languages such as JavaScript, Matlab, and Python, where parallelism likely exists but is difficult to extract using traditional, conservative analyses and optimizations.

References

[1] LLVM (low-level virtual machine). http://llvm.org.

[2] A. Bhattacharyya. Using combined profiling to decide when thread level speculation is profitable. In *PACT'12*, pages 483–484, 2012.

[3] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT'98*, pages 176–184, Oct. 1998.

[4] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA'03*, pages 434–446, June 2003.

[5] H. Cui, J. Wu, C. che Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In *OSDI'10*, 2010.

[6] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI'07*, pages 223–234, June 2007.

[7] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI'04*, pages 71–81, June 2004.

[8] A. García-Yágüez, D. R. Llanos, and A. González-Escribano. Squashing alternatives for software-based speculative parallelization. *IEEE Transactions on Computers*, pages 247–279, May 2013.

[9] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS compiler that exploits program structure. In *PPoPP'06*, pages 158–167, Mar. 2006.

[10] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE'08*, pages 23–32, May 2008.

[11] C. E. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *SPAA'09*, pages 223–232, Aug. 2009.

[12] C. J. Pickett and C. Verbrugge. SableSpMT: a software framework for analysing speculative multithreading in Java. In *PASTE'05*, pages 59–66, Sept. 2005.

[13] C. J. Pickett and C. Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC'05*, volume 4339 of *LNCS*, pages 304–318. Springer-Verlag, 2005.

[14] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *PPoPP'03*, pages 1–12, 2003.

[15] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *PLDI'11*, pages 1–11, June 2011.

[16] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI'05*, pages 269–279, June 2005.

[17] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems (TOCS)*, 23(3):253–300, Aug. 2005.

[18] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. *PLDI'11*, pages 295–305, June 2011.

[19] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA'05*, pages 439–453, Oct. 2005.

[20] J. Whaley and C. Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *ICPP'05*, pages 147–156, June 2005.