SEED: A Statically-Greedy and Dynamically-Adaptive Approach for Speculative Loop Execution

Lin Gao, Lian Li, Jingling Xue, *Senior Member, IEEE,* and Pen-Chung Yew, *Fellow, IEEE*

Abstract—Research on compiler techniques for thread-level loop speculation has so far remained on studying its performance limits: loop candidates that are worthy of parallelization are manually selected by the researchers or based on extensive profiling and pre-execution. It is therefore difficult to include them in a production compiler for speculative multithreaded multicore processors. In a way, existing techniques are *statically adaptive* ("realized" by the researchers for different inputs) yet *dynamically greedy* (since all iterations of all selected loop candidates are always parallelized at run time).
This paper introduces a SEED (Statically GrEEdy and Dynamically Adaptive) approach for thread-level speculation on loops that is quite different from most other existing techniques. SEED relies on the compiler to select and optimize loop candidates greedily (possibly in an input-independent way) and provides a runtime scheduler to schedule loop iterations adaptively. To select loops for parallelization at run time (subject to program inputs), loop iterations are prioritized in terms of their potential benefits rather than their degree of speculation as in many prior studies. In our current implementation, the benefits of speculative threads are estimated by a simple yet effective cost model. It comprises a mechanism for efficiently tracing the loop nesting structures of the program and a mechanism for predicting the outcome of speculative threads. We have evaluated SEED using a set of SPECint2000 and Olden benchmarks. Compared to existing techniques with a program's loop candidates being ideally selected a priori, SEED can achieve comparable or better performance while aututomating the entire loop candidate selection process.

Index Terms—Loop-Level Speculation, Thread-Level Speculation, Speculative Compilation

1 Introduction

The hardware support for thread-level speculation (TLS) on multicore processors [6], [26] allows speculatively parallel threads to be created from sequential code without having to prove they are independent. If a speculative thread executes incorrectly, a recovery mechanism is used to restore the machine state. This architectural support provides more opportunities to uncover thread-level parallelism in applications difficult to parallelize traditionally.

Both hardware and software approaches have been proposed. In loop-level speculation, each iteration of a loop forms a speculative thread that runs in parallel with the other iterations of the same loop or other loops (nested or otherwise). Many programs, however, contain a large number of loops. The challenge is to select the *right* subset of these loops, which may vary from input to input, and to fully exploit the *dynamic* parallelism available in those loops at run time with minimum overhead.

The pure hardware approaches [1], [4], [12] have the advantage of being transparent to users, being able to run legacy code without re-compilation, and being adaptive to input data. However, they are often limited in the amount of extractable parallelism due to the lack of source-level information and TLS-enhancing compiler optimizations. It is generally agreed that if we want to take full advantage of the hardware TLS support, software support is required to identify and optimize potential loops and avoid poor speculation.

Recently proposed compiler techniques on thread-level speculation for loops are also rather restrictive. In most of those schemes, all loops in a candidate set, denoted $\mathcal{LC}$, are speculatively parallelized in a greedy manner on the assumption that all loops in $\mathcal{LC}$ have been ideally selected (for some particular inputs). In most studies [2], [7], [8], [11], [16], [17], [29], $\mathcal{LC}$ for a program is determined either manually or based on extensive profiling (including dependence profiles), [7], [8], [29] or pre-execution of a parallelized program on some sample inputs [11] or real inputs [17]. Since the focus of these performance limit studies is on showcasing the best possible speedup of their techniques, the loops in $\mathcal{LC}$ are ideally selected in their experiments (for good reasons). Such idealized loop selection makes it difficult for them to be adopted in a production compiler. To automate this process, the compiler needs to optimize based on the average-case execution time (ACES) of a program, resulting in too few loops in $\mathcal{LC}$ if it is to conservative or too many if

- *Lin Gao, Lian Li and Jingling Xue are with School of Computer Science and Engineering, UNSW, Australia.*
- *Pen-Chung Yew is with Department of Computer Science and Engineering, University Of Minnesota.*

it is too greedy. In either way, parallelism cannot be adequately exploited.

In the case of thread scheduling, there are further difficulties in existing compiler techniques. First, all loops in $\mathcal{LC}$ are parallelized greedily even though some may not benefit from speculative execution, and some could even suffer a performance loss. This requires their loop selection to be unduly restrictive so that all loops in $\mathcal{LC}$ would produce good speedups during most or all program executions. Second, threads are spawned from a loop either in-order or out-of-order without considering the dynamic parallelism available in the program. Third, in the case of multi-level loop speculation [7], [11], [16], [17], a simple scheduling policy that prioritizes threads based on their degree of speculation (DOS) is used in most prior studies [2], [7], [11], [16], [17], [29]. In such DOS-based scheduling, threads are prioritized in favor of the least speculative threads. Therefore, threads that execute inner loop iterations are always given a higher priority than the threads that execute outer loop iterations (by squashing the latter when there is no more free core available). Furthermore, squashing a thread may trigger all its spawned threads to be squashed. As a result, this simple policy often fails to exploit parallelism that is dynamically and independently available in other loops nested in the same loop nest structure.

If existing speculative compiler techniques [2], [7], [8], [11], [16], [17], [29] were to be fully implemented in a production compiler, ACES-driven adaptive loop selection and DOS-based greedy thread scheduling would have to go together. However, such a combination could be problematic, especially when some of the nested loops are inside an IF-statement or have function invocations and non-constant loop bounds.

The thesis of this work is that their roles should be reversed: dynamic runtime scheduling should be adaptive and driven by a cost-benefit analysis while static loop selection should be greedy and thus easily automated in a compiler. In particular, loop selection should be done in an input-independent manner: only the loops that would not benefit from speculative execution are excluded from $\mathcal{LC}$ based on a simple *compile-time analysis*. This new approach not only makes it easier to be implemented in a production compiler but could also exploit more dynamic parallelism available in a program at runtime.

This paper makes the three main contributions:

- We introduce SEED, a more practical approach to parallelize multiple loops in sequential code (Section 2). The novelty lies in allowing the compiler to select and optimize loop threads greedily, but relying on a runtime scheduler to maximize performance adaptively by prioritizing loop threads based on their *potential benefits* rather than their *degree of speculation*.
- We present an implementation of SEED including a new thread scheduling scheme to realize its *statically-greedy but dynamically- adaptive* strategy (Section 3). Speculative threads are scheduled based on their potential benefits guided by a simple yet effective cost model. The model comprises a mechanism to efficiently trace the loop nesting structure in the program and a mechanism to predict the outcome of speculative threads.
- We have implemented SEED in GCC and compared it with the existing multi-level loop speculation approaches that rely on ideally selected loop candidates and DOS-based thread scheduling (Section 4). SEED can achieve comparable or better performance validated using a set of selected Olden and SPEC2000 benchmarks while automating the entire loop candidate selection process.

2 The SEED Methodology

In this section, we first examine in more detail the limitations of existing TLS-based compiler techniques by way of examples and then outline the key components in our SEED approach.

To effectively exploit the speculative thread-level parallelism in a program, we need to both maximize parallelism and increase its coverage, i.e., the percentage of code executed under speculative execution. In many programs, inner loops tend to have higher parallelism but lower coverage while outer loops cover more of the program but have lower parallelism [14]. To be fully automatable and as competitive as (and even better than) existing compiler techniques, SEED aims to overcome their limitations by maximizing both parallelism and coverage in a program.

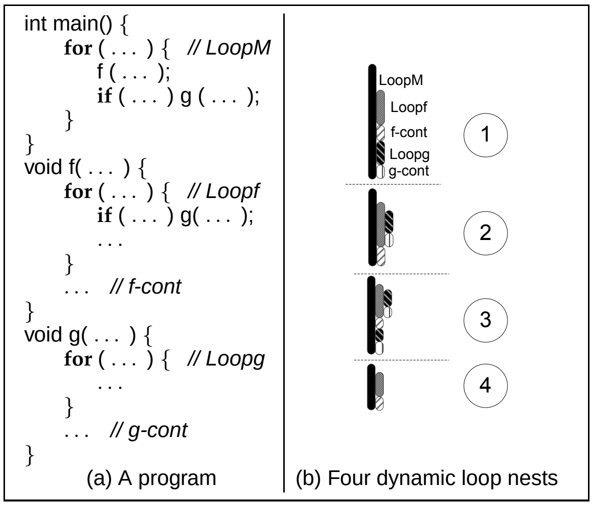
```
for (; node; node=node→list) {
    do {   // DOALL
        othernode = . . .
        } while (k < j)
    ++othernode→from; // input-dependent dependences
}
```

Fig. 1. Dynamic parallelism available at both loops.

Figure 1 gives a double loop abstracted from em3d in the Olden benchmark suite. The outer loop is a DOACROSS loop with its degree of parallelism depending on the runtime values of `othernode` computed in the inner DOALL loop.

Let us look at the limitations of existing TLS-based compiler techniques [2], [7], [8], [11], [16], [17], [29]. With ACES-based loop selection, the outer loop may or may not be selected to be parallelized, i.e., included in $\mathcal{LC}$. In either case, its parallelism and coverage cannot be effectively exploited for some of its executions depending on the runtime values of `othernode`. In the two extreme cases, the outer loop is parallelized (as it is included in $\mathcal{LC}$) when it is actually DOSEQ and sequentialized when it is DOALL. Furthermore, when both loops are parallelized, DOS-based scheduling will prevent outer loop iteration threads from being executed when there are not even enough cores to execute the inner loop as validated by experiments (to be discussed in Section 4).

In our SEED approach, we expect both loops in Figure 1 to be greedily selected (i.e., included in $\mathcal{LC}$) for parallelization since the inner loop is DOALL and the outer loop is expected to have some DOACROSS parallelism for some program executions. However, the parallelism in both loops will be adaptively exploited at run time. When the program is currently executing the i-th outer loop iteration, let us consider the scenario in which this iteration and outer loop iterations $i+1$ and $i+2$ are independent but outer loop iteration $i+3$ may be squashed due to some dependence violation (caused by accesses to `othernode`). Then, roughly SEED will allocate two cores to execute iterations $i+1$ and $i+2$ and the remaining cores to execute iteration i and all its inner iterations. In general, both inner and outer loop iterations are executed simultaneously depending on their potential benefits. Thus, unlike existing techniques, both parallelism and its coverage can be maximized at the same time.



```
int main() {
    for ( . . . ) {  // LoopM
        f ( . . . );
        if ( . . . ) g ( . . . );
    }
}
void f( . . . ) {
    for ( . . . ) {  // Loopf
        if ( . . . ) g( . . . );
        . . .
    }
    . . .  // f-cont
}
void g( . . . ) {
    for ( . . . ) {  // Loopg
        . . .
    }
    . . .  // g-cont
}
```

Fig. 2. Dynamic loop nesting.

The limitations of existing compiler techniques become more apparent when we consider programs with their loop nesting structures being formed dynamically due to complex control flow and caller-callee relations. For the example program given in Figure 2(a), there are four different *dynamic loop nests* given in Figure 2(b), which correspond to four different combinations of the evaluation results of the two if statements. Different loops exhibit different degrees of parallelism, which may depend on program inputs and vary during different program executions or different stages of the same program execution. It will be difficult to apply ACES-based loop selection and DOS-based scheduling to find a static parallelization scheme that is effective in all the situations.

For loop nests formed dynamically at run time, a good parallelization scheme should exploit parallelism from all loop levels simultaneously. This allows both parallelism and coverage to be maximized. However, we cannot create infinitely many threads. As a result, the limited number of cores available in a platform should be utilized effectively to maximize performance.

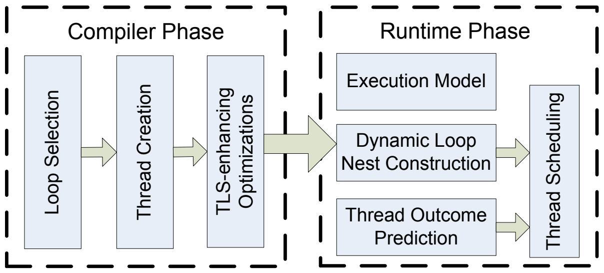


Fig. 3. An overview of SEED.

Figure 3 outlines our SEED methodology designed to be used by ordinary programmers with comparable or better performance than existing compiler techniques. This is made possible due to the key improvements of SEED over the existing AECS- and DOS-based techniques represented by [2], [7], [8], [11], [16], [17], [29] as summarized in Table 1. All the components of SEED, together with the rationales for making these improvements or changes, are described in Section 3. The feasibility of SEED is evaluated in Section 4. Essentially, we rely on the compiler to identify and optimize greedily the loop candidates in $\mathcal{LC}$ and adopt an adaptive runtime scheduling scheme to selectively run in parallel the loop iterations that are likely parallelizable from some multiple loops in $\mathcal{LC}$.

3 An Implementation of SEED

In this section, we present a concrete implementation of SEED to validate our proposed approach. In this implementation, we leverage and integrate several existing TLS-based techniques. We have also added a few new features, for example, the techniques of constructing dynamic loop nests, predicting thread outcomes and scheduling threads with a cost-benefit analysis. However, it is the use of the SEED approach as the underpinning of the implementation and integration that is most novel in this work.

In SEED, the dynamic loop nest being executed at any time in a program is tracked. In order to determine the potential benefit of running threads to execute iterations from either an outer or inner loop, thread boundaries are partitioned differently from other existing techniques. In particular, a thread that executes an inner-loop iteration will execute only

Approach	Loop Selection	Thread				Loop Nesting Construction & Outcome Prediction
		Creation	Spawning	Scheduling	Squashing	
ACES- & DOS-based	Largely researchers-assisted to improve ACES	Non-essential due to being DOS-based	Inorder or out-of-order for both main & speculative threads	Prioiritized by DOS	DOS-based (offending thread & all more speculative ones)	NO
SEED	Greedy, easily automated by compiler (§ 3.1.1)	Iteration boundaries marked more precisely (§ 3.1.2, 3.1.3 & 3.2.1.3)	Out-of-order for main thread and one spawnee for a speculative thread (§ 3.2.1.1)	Prioritized by benefits (§ 3.2.4)	Offending thread & all more speculative ones at the same nesting level (§ 3.2.1.2)	YES (§ 3.2.2 & 3.2.3)

TABLE 1
The key differences between SEED and the ACES- and DOS-based compiler approach.

the continuation code that belongs to the same outer-loop iteration. These differences are reflected in thread spawning and the execution model, which are otherwise similar to the schemes used in the literature.

3.1 Compiler Phase

As shown in Figure 3, three modules are used in this phase. Given a program, $\mathcal{LC}$ is first selected by the "Loop Selection" module. Then thread-spawning instructions are inserted by the "Thread Creation" module to the selected loops in $\mathcal{LC}$. Finally, in the "TLS-Enhancing Optimizations" module, optimizations are applied to those selected loops to reduce the misspeculation cost that would be incurred otherwise. Note that such compiler optimizations cannot be applied as such by pure hardware approaches [12], [1], [4].

3.1.1 Loop Selection

Due to the adaptive scheduling used in SEED, loops can be selected greedily so that a loop that exhibits parallelism in some but not all program executions can be included in $\mathcal{LC}$. Furthermore, any existing loop selection techniques can be leveraged here.

To demonstrate the feasibility of SEED, $\mathcal{LC}$ is selected greedily in our current implementation. All loops in a program are included in $\mathcal{LC}$ unless they are irreducible, have small trip counts, or have small workload in the loop bodies. Our measurement shows that loops with trip counts smaller than three exhibit little loop-level parallelism. They are thus ignored. In addition, a loop must have a body large enough to offset the speculation overhead incurred. There are three cases. In Case 1, a loop that contains calls is assumed to be large. In Case 2, we deal with inner loops that contain no calls. Such an inner loop is large if its estimated execution time is larger than the speculation overhead. In Case 3, we deal with outer loops that contain no calls. For such an outer loop, if all its nested inner loops have small trip counts, it is handled as in Case 2. Otherwise, the outer loop is treated as large and included.

It should be emphasized that in our current implementation, loop selection is done by a simple compiler-time analysis. No memory dependence profiling information is used (although it can be exploited if available). Some execution profiles are needed only to decide whether a loop has a small trip count during most or all its program executions. Such information may be obtained through inter-procedural constant propagation or user annotations.

3.1.2 Thread Creation

For each loop in $\mathcal{LC}$, its iterations may be executed in different threads. The compiler will divide its loop body into a *pre-spawn region* and a *post-spawn region* by inserting a spawn instruction, SPAWN(S), where S is the starting address of the loop. In addition, S is also the starting address of the speculative thread that is spawned. A special instruction CHECK_EXIT(S) is inserted at every *exit* of the loop, and a special instruction CHECK_VALIDATE(S) at the back edge of the loop. We refer to these two types of special instructions as CHECK instructions when there is no need to distinguish them. During the execution of a loop, every loop iteration is terminated by a CHECK instruction. CHECK_VALIDATE plays a similar role in existing TLS execution models. CHECK_EXIT is unique in SEED to make sure that the thread that executes the last iteration of an inner loop may continue to execute the continuation code that belongs to the same outer loop iteration (for scheduling purposes). More details on the semantics of both CHECK instructions are explained in Section 3.2.1.3.

In practice, the starting address of a thread is augmented with the stack pointer so that different dynamically created loops due to recursion are distinguished. For convenience, a program is assumed to be enclosed by a trivial loop, denoted *root*. Thus, every thread starts from a loop's starting address.

3.1.3 TLS-Enhancing Optimizations

In general-purpose applications, loop-level parallelism is often obfuscated by a small number of data and control dependences [30]. TLS-enhancing optimizations are critical to uncover and eliminate such hindrances to loop-level parallelism. In our current implementation, two optimizations, pre-computation [17] and software value prediction [29], are applied.

A pre-computation slice computes the correct input values for a speculative thread, in particular, the values of the loop induction variables, to reduce some frequent data dependence violations caused. So is the computation for the loop terminating condition to reduce control dependence violations.

The (size) ratio of the pre-spawn region over the post-spawn region is crucial. A tradeoff between thread-level parallelism and misspeculation overhead is usually made there. If we push all loop instructions into the pre-spawn region, no misspeculation will occur but the loop will also be totally sequential. In our current implementation, the ratio threshold is preset to be 10%. No more instructions are moved into the pre-spawn region if the threshold is reached. Our results show that this threshold is large enough to accommodate most pre-computation slices.

Software value prediction is a technique that inserts value prediction code as well as mis-prediction check and recovery code into a program at compile time to predict critical values of certain variables. In SEED, this is applied to the variables that are only computed in the post-spawn region and used in the later iterations (i.e., those that cannot be pre-computed in the pre-spawn region). They include loop induction variables that cannot be pre-computed, variables in loops with a similar behavior to induction variables (e.g., `if (cond) i++`, where `cond` is mostly true or false), and return values of some function calls.

3.2 Runtime Phase

This phase aims to execute mostly parallelizable iterations from multiple loops in $\mathcal{LC}$ to maximize *parallelism* and *coverage*, by prioritizing threads based on their *potential benefits* rather than their *degree of speculation (DOS)*. There are four modules as shown in Figure 3. The "Execution Model" module dictates how threads are spawned, squashed and validated. The "Thread Outcome Prediction" module predicts whether a speculative thread will be squashed due to dependence violations or stalled due to load imbalance or speculative buffer overflow. This ensures that some loops in $\mathcal{LC}$ that are greedily selected by "Loop Selection" are not executed in parallel if they have little parallelism in a particular execution unless spare cores are available. The "Dynamic Loop Nest Construction" module traces precisely and efficiently the dynamic loop-nesting structure for all loops at a particular point in time during program execution. Based on the dynamic loop-nesting structure and the predicted thread outcome, a simple yet effective cost model is used to estimate the relative benefit of running different speculative threads in parallel. Finally, the "Thread Scheduling" relies on this cost model to schedule a speculative thread (by de-scheduling another thread when no free cores are available).

3.2.1 Execution Model

As summarized in Table 1, the execution model for the ACES- and DOS-based approach is simple. There is no need to distinguish between outer and inner loop iterations. In particular, a speculative thread that executes the last iteration of an inner loop is allowed to continue executing instructions anywhere in its continuation code. In contrast, SEED aims to exploit the parallelism available in different loops by prioritizing threads in terms of their potential benefits rather than their degree of speculation. For the loops that are nested at the same level and enclosed in another loop, there are generally data dependences of producer-consumer types and/or control dependences between the two adjacent loops. As a result, in SEED, squashing a misspeculated thread will only cause all threads at the same nesting level that are more speculative than itself to be squashed. Data forwarding is now confined to the same nesting level (Section 3.2.1.2). Note that such dependence violations (across the same or different levels) can still be detected as before. As a result, a speculative thread that executes the last iteration I of a loop is allowed to continue executing the first iteration I' of the next loop at the same nesting level. (We could force I and I' to run in two separate threads by inserting a spawn instruction in between. However, doing so will not be efficient since I often contains only a few instructions.)

As is customary, the *main thread* is the only non-speculative thread that can commit and update the program state. Thus, all threads are committed in their original sequential program order. A speculative thread could stall due to a speculative buffer overflow, resource shortage or load imbalance. In addition, a speculative thread will be squashed due to data or control dependence violations.

3.2.1.1 Thread Spawning: On encountering a SPAWN instruction, a thread may spawn a speculative thread if there is a free core. A SPAWN(S) is associated with a loop identified by its unique starting address S. The *nesting level* of a SPAWN is the nesting level of its associated loop in the dynamic loop nest observed at that point in time.

The main thread can spawn out-of-order speculative threads at all SPAWN instructions it encounters during its execution. These SPAWN instructions are associated with different nested loops in a dynamic loop nest. The earliest spawned thread is the most speculative because it is associated with the outermost loop in the dynamic loop nest.

A speculative thread T executing an iteration of a loop L can spawn at most one speculative thread. The first encountered SPAWN instruction will be at the same nesting level as T. All other SPAWN instructions encountered during the execution of T are ignored (i.e., treated as no-op instructions). The intuition behind this *one-spawnee* restriction is to fully parallelize the inner loops of a loop nest first in order to maximize parallelism while keeping outer loop threads relatively large to improve coverage and maintain load balance. Iterations of the outer loops are often more speculative since they are farther apart from each other in the original sequential execution order. In speculative parallelization, allowing a speculative thread to create many smaller spawnees out of order may cause them to stall and thus create unintended load imbalance. Our experimental results presented in Section 4 confirm the effectiveness of this simple *one-spawnee* approach. Thus, "one spawnee" is a performance enabler rather than a limiter. This policy was not generally endorsed in the prior work that dealt with speculative execution of multiple nested loops [2], [7], [8], [11], [16], [17], [29] as shown in Table 1.

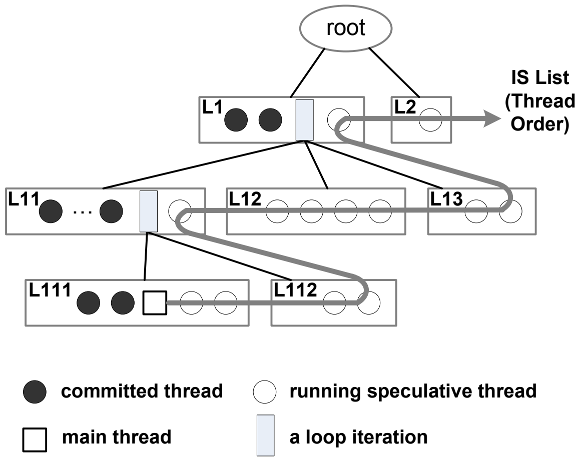


Fig. 4. A snapshot of *S-nest* in SEED (Part I).

SEED maintains at run time the dynamic loop nest structure, *S-nest*, being executed in the program. Figure 4 gives a snapshot of *S-nest* for an imaginary example. *S-nest* is represented as a tree in which each tree node represents a loop. Each labeled horizontally-stretched rectangle in the figure represents a tree node, i.e., a loop, with many iterations in the node. So, *root*, *L1, L11, L111, L112, L12, L13* and *L2*, form the current *S-nest*, where *root* encloses the entire program. A thread is represented by either a circle or a square. A solid circle is a committed thread. The main thread, which conceptually includes all committed threads, starts from `root`. Threads working on the inner loops at the same level are executing the same loop iteration of their immediately enclosing loop.

Due to the one-spawnee policy imposed on speculative threads, the following properties are immediate:

Property 1: In a post-order traversal of S-nest with all committed threads being ignored, all threads are ordered from the least to the most speculative with the first being always the main thread.

Property 2: All children of a loop node L in *S-nest* must be executing the same iteration of L.

Property 3: S-nest is left-biased such that only the leftmost child of a node can itself have child nodes.

In out-of-order thread spawning, additional hardware support is necessary for efficient thread ordering. In [20], all running threads will link themselves dynamically in hardware in a *immediate successor (IS) list* according to their sequential order. SEED uses the same *IS* list as in [20] to track the thread order, which happens to be a post-order traversal mentioned in Property 1. In Figure 4, the zigzag arrow in gray represents such a thread ordering sequence, i.e., *IS*. In particular, the main thread is the head of *IS*. The *nesting level* of a speculative thread is defined to be the nesting level of the loop from which this thread is spawned. The *nesting level* of the main thread is defined to be the nesting level of the innermost loop that it is executing in *S-nest*.

3.2.1.2 Thread Squashing: The speculative versioning cache (SVC) [25] is used to detect dependence violations. In [25], when a thread T issues a load request that causes a cache miss, say, in memory location X, a version control logic (VCL) is used to fetch the most recent version of X from a less speculative thread T' based on the *IS* list and the status of the L1 cache line for X. We have modified SVC slightly to incorporate the nesting levels of threads so that T is sent a kill signal if T' is at a different level from that of T. So memory forwarding will take place only within a particular loop level in *S-nest*. Note that the main thread is conceptually working through all nesting levels. This modification aims to reduce the cross-level interference in determining the potential benefits of parallelizing different loop levels.

Everything else works exactly the same as in SVC. If T has obtained the most recent version of X from, say, T'' but there is a subsequent write into X by T' that is more speculative than T'' but less speculative than T, then T will receive a kill signal.

Any thread that receives a kill signal will squash itself and propagate the signal to its successor in *IS* if the successor is at the same nesting level. Hence, the offending thread and all of the more speculative threads at the same level in *IS* are squashed. Unlike DOS-based scheduling, SEED does not indiscriminately squash all more speculative threads in order to exploit more dynamic parallelism in other nested loops in the loop nest.

3.2.1.3 Thread Validation: Like a SPAWN instruction, a CHECK is associated with a loop. A successful execution of CHECK in a thread T indicates the completion of the iteration assigned to the thread. Then, thread validation will take place. There are two cases depending on whether T is a speculative thread or the main thread.

Suppose T is a speculative thread executing an iteration of a loop L. All speculative threads spawned by T are at the same nest level of T. Thus, thread validation takes place only for a CHECK(S) instruction associated with a loop at the same nest level of L. All CHECK(S) instructions that T encounters are ignored if they are associated with the loops nested inside L (since no speculative threads were ever spawned there). If T encounters a CHECK(S) associated with the enclosing loop of L, the task assigned to T is completed and T stalls. Eventually, there are two possibilities: T becomes the main thread. Its execution resumes from this CHECK instruction. Or, T is squashed.

For a CHECK(S) instruction that T encounters at the same nesting level of L, two cases are distinguished:

- CHECK(S) **is** CHECK_VALIDATE(S)**:**
  If the successor thread T' of T in *IS* has a different starting address, then T' must be executing an outer loop iteration. In this case, T will continue executing the next iteration of L. Otherwise, T first checks all predicted values for T'. If all predicted values are validated, then T stalls. Eventually, T will become either the main thread or get squashed. On the other hand, if T detects a data dependence violation in T', then T' will receive a kill signal, causing T' and all other more speculative threads at the same level in *S-nest* to be squashed. T will then continue executing the next iteration of L.
- CHECK(S) **is** CHECK_EXIT(S)**:**
  The current loop L is finished. Let T' be the speculative running thread following T in the thread order *IS*. If the starting address of T' is S, then L has a control misspeculation. In this case, T' and all other more speculative running threads at the same level in *S-nest* are squashed. Either way, T will continue its execution.

If T is the main thread, T needs to validate its successor thread in *IS* every time T encounters a CHECK instruction because the main thread can spawn speculative threads at any nesting level in *S-nest*. All our earlier discussions for the case when T is a speculative thread are still valid except in the case of CHECK_VALIDATE(S). If the successor thread T' of T in *IS* has the same starting address S and all predicted values for T' are validated by T, then T commits and T' becomes the main thread.

3.2.2 *Dynamic Loop Nest Construction*

SEED maintains *S-nest* by augmenting the *IS* list used for maintaining the thread order. The extra cost incurred is minimal. By Property 3, *S-nest* is left-biased. Hence, there is no need to build a tree to represent *S-nest* at run time. We augment the thread description structure of each thread T with two fields, *level* and *leftmost*, to capture *S-nest*. For every SPAWN instruction, the nesting level of the loop at which the SPAWN instruction is executed is known. So T.*level* is set as such. T.*leftmost* represents whether the iteration executed by T is from the leftmost, i.e., the lexically earliest loop at T.*level*. There are two cases to initialize this field when T is spawned. If T is spawned by the main thread, then T.*leftmost*=true. If T is spawned by a speculative thread T', we examine T' instead. If T and T' have different starting addresses, then they must be associated with different loops at the same level, we set T.*leftmost*=false. Otherwise, we set T.*leftmost*=T'.*leftmost*.

The loop where the main thread is executing is always the leftmost node at the lowest level of *S-nest*. Hence, whenever a speculative thread T becomes the main thread, we set the *leftmost* field to true for both T and all other more speculative ones with the same starting address at the same nesting level.

Let T and T' be two threads executing some iterations of loops L and L', respectively. Loops L and L', which may be identical, are at the same nesting level in *S-nest* when T.*level* = T'.*level*. If T.*level* > T'.*level* and T.*leftmost* = true, then L' is nested inside L. Otherwise, L' is not nested inside L.

3.2.3 *Thread Outcome Prediction*

Unlike most previous prediction mechanisms [21], [14], [3] that were introduced to predict values and to reduce inter-thread dependence violations for a pre-selected parallelizable loop, our thread outcome prediction serves to predict whether a future thread is likely to be squashed/stalled or not. The prediction results are used to adaptively select loops to parallelize at run time. Due to the difference in the role of predictors, a much simpler prediction mechanism could be adopted for our purposes. In our current implementation, we use a popular two-level bi-mod centralized prediction mechanism [10] to predict the outcome of speculative threads.

Our predictor has a table of entries indexed with loop starting addresses. Each entry is associated with a loop and has a *stall counter* and a *squash counter*. A stall (squash) counter, which starts with the initial value *not-stall* (*not-squash*), predicts whether future threads spawned will stall (be squashed) or not. A thread may stall due to load imbalance (caused by varying granularities of different loop iterations) or speculative buffer overflow or may be squashed due to dependence violations. A stall counter is updated when a speculative thread stalls or retires successfully without incurring any stall cycles. A squash counter is updated when a thread is squashed or retires successfully. Any update to a stall (squash) counter may cause it to transit between *not-stall* and *stall* (*not-squash* and *squash*) according to the underlying state machine introduced in [10].

A thread has two status bits, *stallable* and *squashable*, with their meanings as indicated. Its status is initialized when spawned and remains unchanged

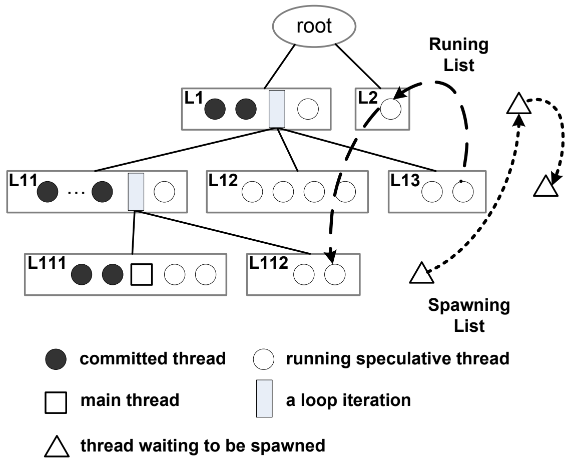


Fig. 5. A snapshot of *S-nest* in SEED with two lists (Part II).

during its execution. The status bits are either inherited from its spawner or obtained from its thread predictor as follows. Let B be spawned by A. Let X be any of the two status bits. We set $B.X = A.X$ if $A.X =$ *true* and as predicted for the loop from which B is spawned otherwise. We make B inherit its status bits from A in the first case by exploiting some future information that cannot be accurately predicted from history information. If A stalls, B will generally stall since all threads commit in sequential order. If A is squashed, so will all other more speculative threads executing the same loop in our execution model.

3.2.4 Thread Scheduling

In our current implementation, speculative threads are scheduled and de-scheduled based on their potential benefits estimated by a simple cost model. The cost model is built based on *S-nest* and the predicted thread outcomes.

The relative benefit of any two speculative threads T and T' are ranked as follows. If $T.level = T'.level$, then the more speculative one is less beneficial. Otherwise, $T.level \neq T'.level$. Then, their predicted thread outcome and nesting information are checked. T and T' are equally beneficial if both are squashable threads. If one is squashable but the other is not, then the squashable thread is less beneficial. Finally, T and T' are two non-squashable threads that must be spawned from distinct loops since $T.level \neq T'.level$. T and T' are equally beneficial if one is *not* nested within the other (due to the lack of information to distinguish them in our current implementation). When both are executing nested loops, we assume without loss of generality that the loop L' executed by T' is nested inside the loop L executed by T. Then there are three cases. In Case 1, if one is stallable but the other is not, then the non-stallable thread is more beneficial than the stallable one. In Case 2, when both are stallable, then T' is more beneficial. In Case 3, when both are non-stallable, then T is more beneficial. The motivation for Case 1 is self-explanatory. Case 2 happens since the outer loop thread T is more speculative and will continue to stall until it either becomes the main thread or gets squashed. Therefore, we prefer to exploit parallelism over coverage by giving preference to the thread T' that runs at the inner loop and squashing T if no free core is available. In Case 3, preference is given to coverage over parallelism instead.

To facilitate thread scheduling, two lists, in addition to *IS*, are maintained as shown in Figure 5. All threads waiting to be spawned are kept in a *spawning list* in non-increasing order of their potential benefits. Such a list has to be maintained in any TLS system that supports out-of-order thread spawning. Due to our one-spawnee restriction, there can be at most one thread waiting to be spawned at each nesting level. So the spawning list in SEED is much smaller than that in other out-of-order spawning schemes. Another, called the *running list*, is used to find the least beneficial one among all running speculative threads in *IS*. As can be observed in Figure 4, the last thread at each nesting level is the most speculative and thus the least beneficial. So the running list simply links the most speculative threads at all nesting levels in the *IS* list in a non-increasing order of their benefits.

The sizes of both the spawning list and the running list are bounded by the number of levels of *S-nest*, which is small in practice. Both lists are significantly smaller than the *IS* list, hence, both lists can be efficiently maintained. In addition, the operations on the lists can overlap with program execution.

The scheduling/descheduling process is simple. When there are free cores available, the head thread in the spawning list is spawned immediately. Otherwise, the head thread is spawned and removed from the spawning list when it is more beneficial than the tail thread in the running list. In that case, the tail thread is squashed and removed from the running list. The running list is updated as follows. First, the immediate predecessor of the squashed tail thread that is also executing the same loop (if any) is made available in the running list. Then, the head thread is linked in the running list to substitute for its predecessor that is executing the previous iteration of the same loop (if any). A committed thread needs to be removed from the running list only if it is executing the last iteration of a loop (otherwise, it would not appear in the list).

4 Experimental Results

We focus on the practicality of SEED by analyzing its relative performance to the ACES- and DOS-based multi-level loop speculation (Table 1), denoted by DOS. In our experiments, the process of manually selecting the "best" loops in $\mathcal{LC}$ for DOS is the most painful undertaking, and a major shortcoming eliminated in SEED. To choose the best $\mathcal{LC}$ set for DOS, we spent about four weeks to perform time-consuming memory profiling with *true inputs* and extensive manual code analysis, as discussed in Section 4.1.

In addition, we also compare SEED against DOS-SEED_$\mathcal{LC}$, a version of DOS applied to the same $\mathcal{LC}$ sets except that DOS-based scheduling is used. The goal is to show that existing techniques cannot effectively work on large-loop candidate sets due to their lack of adaptive scheduling at runtime. In DOS and DOS-SEED_$\mathcal{LC}$, out-of-order task spawning is allowed in both the main thread and the speculative threads.

In some other experiments, we compared SEED against a modified version in which the one-spawnee restriction is relaxed so that out-of-order task spawning is allowed for speculative threads. SEED is superior on almost all benchmarks (due to the reasons given in Section 3.2.1.1).

The compiler phase of SEED described in Section 3 is implemented in GCC-4.1.1, while the runtime phase is realized in a simulator explained in Section 4.2. The same optimizations introduced in Section 3.1.3 are applied to SEED and DOS-related techniques.

A total of 13 benchmarks are selected from SPECint2000 and Olden, and compiled under "-O2". The Olden benchmarks are irregular applications with dynamic data structures. The SPECint2000 benchmarks are general-purpose applications for code compression, games, optimization, FPGA circuit router and languages. The MinneSPEC's large input sets are used for SPECint2000 during simulation. All benchmarks are simulated to completion.

4.1 Loop Candidates

Benchmark	#Loops	$\mathcal{LC}$ (Loop Candidate Sets)	
		SEED & DOS-SEED_$\mathcal{LC}$	DOS
164.gzip	177	85 (95%)	4 (94%)
175.vpr	374	249 (98%)	2 (93%)
181.mcf	48	14 (98%)	6 (85%)
186.crafty	353	245 (59%)	10 (31%)
253.perlbmk	566	308 (99%)	1 (99%)
254.gap	1640	673 (32%)	3 (23%)
256.bzip2	141	68 (98%)	7 (41%)
bh	7	5 (100%)	3 (95%)
bisort	4	2 (88%)	1 (41%)
em3d	17	11 (99%)	2 (98%)
health	11	8 (100%)	4 (60%)
mst	10	6 (100%)	3 (98%)
tsp	8	6 (95%)	3 (95%)

TABLE 2
Loop candidates and their execution-time coverage.

Table 2 lists the loop candidates and their (execution-time) coverage using the three approaches. SEED and DOS-SEED_$\mathcal{LC}$ share the same $\mathcal{LC}$ sets (Column 3), which are determined automatically as described in Section 3.1.1. They also exhibit a higher

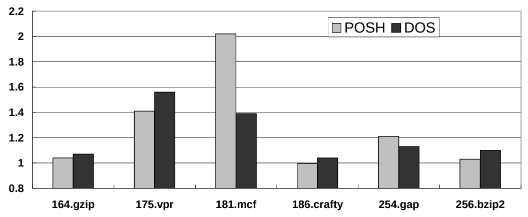


Fig. 6. Speedups over single-threaded code.

coverage than DOS due to the use of adaptive scheduling in SEED. However, DOS uses a smaller subset of the loops used by SEED.

The $\mathcal{LC}$ sets for DOS are manually optimized as follows. For a program, every loop whose coverage is no smaller than 3% is examined. First, DOALL loops and *nearly* DOALL loops (based on dependence analysis and profiling) are selected. Next, for a DOACROSS loop (such as the loop from `fullGtU` in `bzip2` and the loop from `BlueRule` in `mst`), if its cross-iteration dependences are highly predictable, value prediction is applied so that the optimized loop is selected. For a DOACROSS loop with a coverage of over 30%, more optimizations such as inlining and code motion [21], [29], [30] are applied to reduce misspeculations. If the cross-iteration dependences in the optimized DOACROSS loop are highly predictable or likely to be preserved during speculative execution, the optimized DOACROSS loop is selected. In `vpr`, for example, some functions for updating the heap structures, such as `free_heap_data` and `get_heap_head`, that are called from within `route_net`, are inlined so that the heap update operations can be performed earlier in each iteration. Finally, among a large number of possible $\mathcal{LC}$ sets for a program, the one that gives the best execution time is listed in Table 2. We are not aware of any systematic solution unless we try all possibilities exhaustively.

The time-consuming selection of the $\mathcal{LC}$ sets for DOS is effective. In Figure 6, the speedups of DOS over single-threaded code on a 4-core TLS system are compared with those from POSH [11]. POSH is a TLS compiler that relies on a profiler to exploit beneficial threads from program structures including loops. The profiler simulates the execution of sequential code, records the time and dependence information at important program points such as memory access instructions and potential thread start and end points, and estimates the benefit of every potential thread. All common benchmarks are listed in Figure 6. The loop-level parallelism performance results for POSH are taken directly from their paper; similar results were also reported elsewhere [7], [30]. In most benchmarks, DOS performs similarly or slightly better than POSH. As an exception, POSH works very well for `mcf` but the smaller speedup achieved by DOS is close to those reported elsewhere (e.g., about 1.4 in [30] and about 1.25 in [7]). These data show that DOS can accurately represent existing DOS-based compiler techniques. Such a comparison is fair because it is often difficult to replicate an existing TLS technique (due to the lack of details).

4.2 Simulated Architecture

We used a multicore TLS system, in which each core has its own function units, register file, L1 I-cache and L1 D-cache. All the cores share a unified L2 cache. In our simulator (based on SimpleScalar), the SVC [25] coherence protocol is used to buffer speculative states in the L1 data cache, identify remote L1 accesses, and detect misspeculated data dependences. The hit latency for the L1 data cache is modeled as 3 cycles [7]. There is no special hardware used for sharing register values among cores. Each core uses the Alpha ISA with its main parameters listed in Table 3. It relies on a hardware-assisted misspeculation detection mechanism such as [7], [11], [17], [23]. Our implementation uses SVC but it could be any other similar scheme.

Individual Core	Value
Fetch, Issue, Commit	bandwidth 4, out-of-order issue
L1 I-Cache	16KB, 4-way, 1 cycle (hit)
L1 D-Cache	16KB, 4-way, 3 cycles (hit)
L2 Cache (Unified)	1MB, 4-way, 12 cycles (hit)
Memory Latency	250 cycles

TLS Parameter	SEED(cycles)	DOS-SEED_$\mathcal{LC}$ DOS (cycles)
Spawn overhead	20	12
Commit overhead	5	
Squash overhead	20	
Remote L1 accesses	$\geqslant 8$	

TABLE 3
Architecture simulated.

The overhead of thread spawning, committing or squashing a thread and the latency for remote L1 D-cache accesses are given in Table 3. In out-of-order spawning, an ordered list similar to *IS* for threads [11], [20] is maintained and updated when threads are spawned and squashed. A remote L1 D-cache access also involves traversing the list. For SEED, a few additional comparisons are needed to schedule a more beneficial thread by squashing a less beneficial thread. The impact of such hardware changes is small [20]. Thus, the spawning overhead of SEED is slightly larger than that of DOS and DOS-SEED_$\mathcal{LC}$ with these extra operations considered.

We have used a two-level adaptive predictor [28] to predict the outcomes of speculative threads with `l1size=8`, `l2size=2048`, `hist_size=8` and `xor=0`. The first-level table has 8 entries indexed by loop start addresses with a history width of 8 bits. The second-level table is set to 2048. This setting suffices to capture the stall and squash behaviour for the loops used in our benchmarks (Section 3.2.3).

4.3 Performance and Analysis

Figure 7 compares SEED, DOS-SEED_$\mathcal{LC}$ and DOS for their execution times (lower bars are better). By using same $\mathcal{LC}$ sets as SEED, DOS-SEED_$\mathcal{LC}$ is the worst performer among the three, and is 10% (11%) slower than SEED on average with 4 (8) cores. This shows that DOS cannot be effectively applied to many loops without also employing a similar adaptive scheduling mechanism. By automating the entire loop candidate selection process, SEED is still at par with DOS with its average speedups over DOS by 0.19% and 0.22% for 4 and 8 cores, respectively. As analyzed below and also in case studies in Section 4.4, SEED is successful in exploiting more parallelism than DOS in some dynamic loop nests and also keeping to a minimum the overhead incurred for loops that turn out to have little parallelism.

SEED achieves the largest speedup over DOS in `gzip` (6.6% (6.0%) on 4 (8) cores). There is a loop nest whose two loops (residing in `deflate` and `longest_match`) have limited parallelism because only a small number of their cross-iteration dependences are predictable. DOS blindly parallelizes the inner loop while SEED has succeeded in finding a balance between the two loops in its parallelism exploitation. If DOS parallelizes either loop alone, similar results are observed. On the other hand, SEED displays the largest slowdown over DOS in `mst` (4.0% (3.9%) on 4 (8) cores). For a double loop in `AddEdges`, DOS parallelizes only the outer loop while SEED has included both loops in $\mathcal{LC}$. The inner loop has a relatively strong cross-iteration dependence that makes the speculative parallelism difficult to exploit. Thus, SEED suffers some overhead when the dependence does happen.

Let us look at `crafty`, `gap`, `bzip2`, `bisort` and `health` for which the $\mathcal{LC}$ sets selected by SEED have much higher coverage ratios than those used by DOS. For `bzip2`, SEED has a higher coverage since its $\mathcal{LC}$ contains a time-consuming loop; but the loop is too large and ends up being filtered out by the stallable predictor. SEED outperforms DOS since SEED has succeeded in exploiting more parallelism in a loop nest (with the outermost loop contained in `sortIt` and the innermost in `fullGtU`). In this loop nest, the innermost loop has more parallelism than the other outer loops but has a smaller coverage. DOS is biased toward parallelism by parallelizing the innermost loop while SEED finds a better balance between parallelism and coverage. For `bisort`, SEED suffers from some overhead by including in $\mathcal{LC}$ a hot loop that turns out to have no speculative parallelism. For `health`, SEED has included in $\mathcal{LC}$ a loop that contains a recursive function with deep recursion. This loop has little speculative parallelism and is mostly filtered out by the stallable predictor. For `crafty` and `gap`, SEED has included significantly more loops in

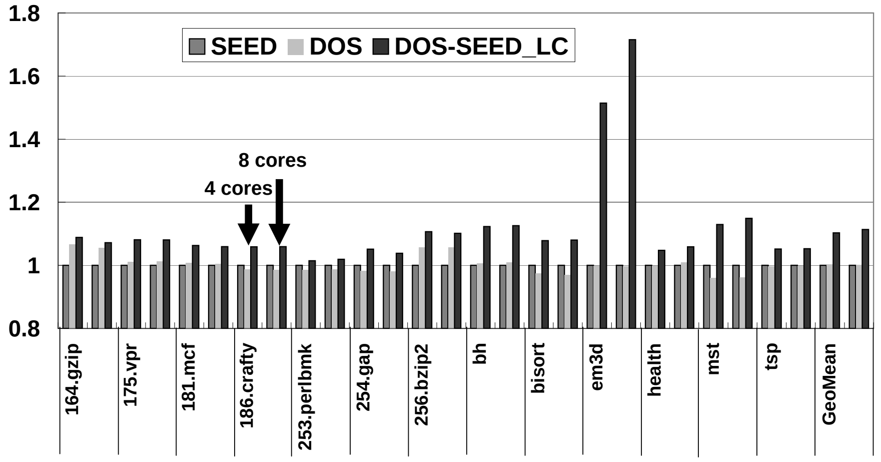


Fig. 7. Execution times normalized with respect to SEED.

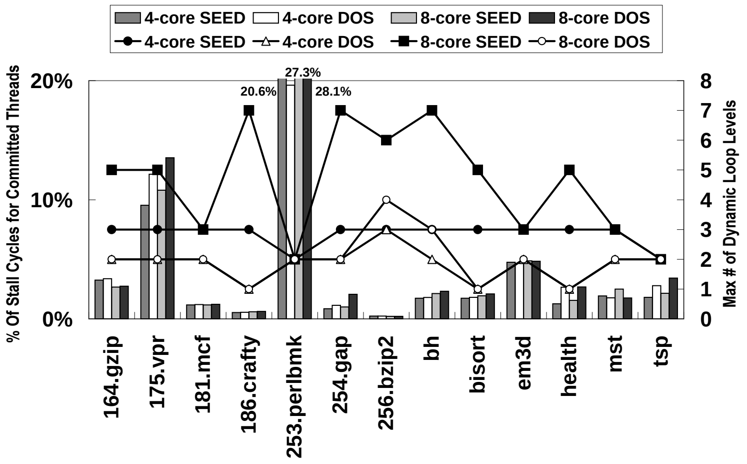


Fig. 8. Dynamic statistics for stall cycles of committed threads (shown by the left y-axis) and the maximum number of loop levels (shown by the right y-axis) executed concurrently.

$\mathcal{LC}$ than DOS. Some of these loops turn out to be beneficial but the majority of them could be filtered out. Hence, SEED underperforms DOS slightly for both benchmarks.

Let us look at the remaining benchmarks. For bh, SEED is superior. In a triply-nested loop with parallelism in all of its loops, DOS favors to parallelize the innermost loop while SEED favors to parallelize the most beneficial one, i.e., an outer loop in most cases. For mcf, SEED outperforms DOS for a similar reason concerning a doubly-nested loop in primal_bea_mpp. For perlbmk and tsp, SEED and DOS have (nearly) the same coverage, implying that the extra loops included in $\mathcal{LC}$ by SEED either are nested inside those selected by DOS or have negligible execution times. However, SEED is slightly worse than DOS due to its adaptive scheduling overhead despite a significantly large number of extra loops being selected.

Figure 8 gives some dynamic statistics. Each bar represents the percentage of the number of stalled cycles incurred by all committed threads over their total number of execution cycles. Only in 6 out of 26 configurations (4 cores for perlbmk, 8 cores for bzip2 and 4 and 8 cores for both em3d and mst) does SEED incur slightly more stall cycles than DOS. SEED may sometimes favor outer loops but only when they are unlikely to stall. Those benchmarks with more stall cycles are expected to benefit if more sophisticated scheduling policies are used. In Figure 8, each line represents the maximum number of dynamic loop levels being parallelized. For both SEED and DOS, more threads are executed in parallel as the number of cores increases. However, given the same number of cores, SEED executes more loops in parallel than DOS.

Due to the greedy heuristics used in loop selection, the $\mathcal{LC}$ sets for SEED are significantly larger than those of DOS (Table 2). However, the number of worse performers in the benchmarks is small due to the adaptive out-of-order scheduling strategies used. This is particularly significant because the $\mathcal{LC}$ sets for both SEED and DOS are selected manually. These preliminary results show that SEED is not only practical because it can be easily deployed but also could achieve comparable or better performance compared to other existing techniques.

4.4 Two Case Studies

We choose vpr to examine the overhead incurred by SEED's adaptive scheduling strategy. It allows us to understand why DOS could outperform SEED sometimes when the $\mathcal{LC}$ sets for DOS are ideally determined by an "oracle" selector. On the other hand, we choose em3d to demonstrate the performance advantage of SEED when the degree of parallelism in loops varies under different problem sizes and inputs. It is therefore possible for SEED to outperform DOS even if an "oracle" loop selector is used in DOS. Thus, as a whole SEED can be as competitive as or even better than DOS when evaluated across a large set of benchmarks.

4.4.1 *vpr*

We examine the cost and benefit of SEED's adaptive scheduling under four different configurations using the two predictors. Bo-Pred is the default used in SEED (with both the squashable and stallable predictors turned on). In SQ-Pred (ST-Pred), only the squashable (stallable) predictor is on. In No-Pred, both predictors are off. Figure 9 compares the performance using vpr under these configurations. Both predictors have positive effect on performance for this benchmark. It runs about 13% slower on 4 cores if both predictors are turned off. Below, we first examine the characteristics of this benchmark, and then analyze the impact of prediction on performance.

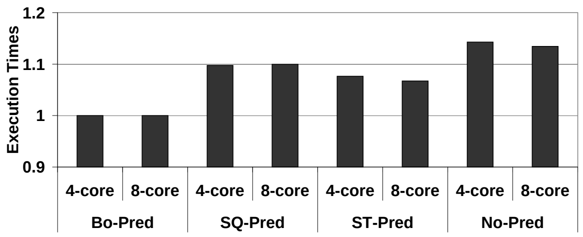


Fig. 9. Execution times of vpr achieved by SEED under four different prediction schemes normalized with respect to Bo-Pred.

As shown in Table 2, the $\mathcal{LC}$ set selected by SEED for vpr contains 249 loops, of which 154 loops are dynamically executed. In particular, the coverage of a four-deep dynamic loop nest is nearly 90%. Let L_1, L_2, L_3 and L_4 be the four loop levels ordered from outermost to innermost, where L_1 appears in try_route, L_2 and L_3 in route_net and L_4 in expand_neighbours. The speculative threads spawned from L_1 and L_2 are so large that they often stall due to speculative buffer overflow. However, it rarely happened to the threads spawned from L_3 and L_4. Due to the branches contained in L_3, about 44% of its iterations are significantly larger than the remaining 56% iterations. There are loop-carried dependences at all four loop levels; but a large number of cross-iteration data-dependent values are predictable. So this nest has good parallelism, especially at L_2 and L_4.

When the squashable predictor is turned on in SQ-Pred, it favors loops with more parallelism. Thus, L_2 and L_4 are selected over L_1 and L_3 for parallel execution. Better performance is thus observed under SQ-Pred than No-Pred. When the stallable predictor is turned on in ST-Pred, the two outermost loops L_1 and L_2 are not selected because they are more likely to stall. In addition, preference is given to L_4 over L_3 because of the load imbalance in L_3. Better performance is thus observed under ST-Pred than No-Pred. When both SQ-Pred and ST-Pred are both turned on in Bo-Pred, even better performance is observed.

The squashable and stallable predictors may take several iterations to warm up in order to function more effectively. Such overheads could offset some of their performance gain.

4.4.2 *em3d*

One of its kernel loop nests is shown in Figure 1. Its outer loop has input-dependent DOACROSS parallelism while its inner loop is a DOALL loop. This kernel covers 98% of the total execution time of em3d.

We use four different inputs to show that the outer DOACROSS loop exhibits different degrees of parallelism at run time. In input1, the outer loop becomes almost like a DOALL loop and the inner DOALL loop has a very small trip count (often only one iteration). In input2, the outer loop behaves again like a DOALL loop, but the inner DOALL loop has a relatively large trip count. In input3, about 70% of the outer loop iterations are independent. In input4, only about 50% iterations of the outer loop are independent. In both input3 and input4, the inner loop behaves the same as in input2.

Figure 10 shows the speedup of SEED over DOS-BOTH (by parallelizing both loops), DOS-INNER (by parallelizing the inner loop only), and DOS-OUTER (by parellelizing the outer loop only), using the four different inputs on 4-core and 8-core systems. SEED is the best performer in nearly all configurations except for input1. With input1, the inner DOALL loop has very small trip counts, resulting in frequent control misspeculations (at its last iteration). In contrast, DOS-OUTER does not suffer from this penalty because the outer loop it parallelizes is almost like a DOALL. DOS-BOTH and DOS-INNER have similar performance because DOS-BOTH is biased toward se-

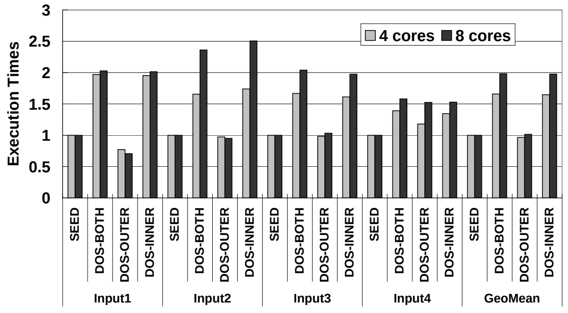


Fig. 10. Execution times of SEED, DOS-BOTH, DOS-OUTER and DOS-INNER for ema3d normalized with respect to SEED.

lecting innermost loop with its DOS-based scheduling strategy.

Now, let us examine the performance using different inputs. DOS-OUTER is the best performer under input1 and input2 because the outer DOACROSS loop it selected is almost like a DOALL loop. DOS-OUTER is also superior to SEED because we did not manually filter out non-beneficial loops for SEED as we did for the other approaches. In this benchmark, there is a DOSEQ loop between the two loops shown in Figure 1. Its outer loop is further nested in some other DOSEQ loops. SEED filtered out these DOSEQ loops dynamically with some small extra overhead. Both DOS-BOTH and DOS-INNER are not effective because they both favor the inner loop that exhibits little parallelism under input1. As the parallelism in the outer loop decreases with input3 and input4, SEED begins to achieve a better performance than the other approaches. In particular, when input4 is used, only half of the iterations in the outer loop are independent. In this case, SEED can achieve a speedup of over 50% than the other three approaches.

Looking at the geomean bar in Figure 10, we could see that SEED outperforms the other three approaches for all four inputs, and shows its adaptability to different inputs. On the other hand, DOS may perform well if the loops could be *correctly* selected (by an oracle).

5 Related Work

A great deal of compiler work has been done on extracting speculative threads from sequential code. In [2], [7], [8], [11], [13], [17], [29], [21], the loop candidates where speculation is likely to be beneficial for a program are identified based on program analysis, profiling information and pre-executions. In [13], [21], [23], [29], [30], loop transformations are applied to remove, isolate, overlap, shift or pre-compute the cross-iteration dependences that may hinder parallelization. In particular, DSWP [23] represents a different way to extract threads from loops. This approach pipelines one single loop iteration across multiple cores so that the latencies of the computations in different slices of the same iteration can be overlapped. However, DSWP is limited to parallelizing one loop level in a loop nest. Parallel-stage DSWP [18] allows multi-level loop parallelization but does not support speculation. In fact, many existing loop-oriented compiler techniques are limited to single-level loop speculation [2], [29], [16]. Some other techniques support multi-level loop speculation [7], [11], [17] and more aggressive speculation at arbitrary program points [7], [8], [11], [17], [26], [27]. In [9], some existing TLS techniques are evaluated.

In [5], a static approach is introduced to parallelize loops also for non-expert programmers. This work can quantitatively estimate the actual speedup/slowdowns of parallelized loops. However, its model supports only in-order thread spawning and allows only one loop in a loop nest to be parallelized.

In TLS research, little attention has been paid to thread scheduling. Initially, thread scheduling attracted little interest due to the use of in-order thread spawn on pre-dominantly ring-based interconnection architectures [4], [22], [24], [26]. Although in-order spawn is simple, out-of-order spawn can often significantly improve performance as demonstrated later in [1], [11], [14], [17], [20]. To facilitate out-of-order spawn, additional hardware support is required to maintain the sequential order of running threads, such as the immediate successor (*IS*) list [20] and the ordering tree [1]. This work demonstrates that the dynamic loop nesting structure of a program can be efficiently and precisely obtained from the *IS* list.

Even though out-of-order spawn has become a predominant scheme for spawning threads, thread scheduling remains in-order in prior work [8], [11], [20], [17], [12], [14], [1]. Therefore, which threads to squash when a new thread is to be spawned but no free cores are found to be available or when dependence violations are detected is decided based on threads' degree of speculation (DOS), which is exactly the execution order in the absence of mis-speculations. This work represents the first attempt of combining out-of-order thread scheduling with out-of-order thread spawning.

Adaptive thread scheduling entails the use of runtime feedback. Some prior studies have proposed microarchitectures to enhance thread spawning with some runtime feedback. Capsule [15] delegates the parallelization decision to the architecture at run time through frequent hardware resource probing by the program. Capsule has been specifically designed for component-based programs. Thread creation is by means of self-replication. Threads are allowed to commit in any order (not necessary sequential), avoiding thread scheduling altogether. However, Capsule is well suited to only componentized applications.

DMT [1] updates a predictor on thread actions like retirement and squash and then decides whether to spawn a thread by the predictor. Although thread spawning in DMT is adaptive, its thread scheduling is still DOS-based and thus in-order. Furthermore, DMT spawns new threads at subroutine and loop continuations. So the innermost loop in a loop nest is always seqentialized (since it is executed in one thread). Finally, unlike the prediction mechanism used in [1], our prediction mechanism is different in that we predict the outcomes of speculative threads and that a thread predicted to be squashable can still be scheduled and commit if it is not squashed by other threads. VPT [3] uses a predictor for future dependence violations. Their prediction table can be large, particularly for irregular programs, since it records the information for all earlier memory access violations. In addition, their mechanism requires every memory read to consult the prediction table, resulting in additional overhead for memory accesses. Unlike VPT, SEED works at the granularity of loop iteration threads. SEED requires a thread to consult our predictor only once when it is spawned and updates the predictor only when the thread is squashed, stalled or retired. The overhead thus incurred is small, particularly when it can be overlapped with other thread overhead.

In [19], it is shown that the energy cost of TLS can be kept modest by using a lean TLS CMP microarchitecture and by minimizing wasted TLS work.

6 Conclusion

We describe an approach to speculatively parallelize multiple nested loops. The basic idea is to allow the compiler to select and optimize loop candidates greedily and rely on adaptive scheduling at runtime to exploit parallelism using a benefit analysis (instead of degree of speculation as in most existing schemes). Compared to other existing techniques, the set of loops (i.e., the program coverage) that could be parallelized is much larger. The extra overhead incurred could be mitigated by using the adaptive out-of-order scheduling scheme mentioned above as well as by the increased parallelism and the program coverage. Our preliminary results show that this new approach not only makes it easier to deploy in a production compiler, but also achieves a better or comparable performance to existing techniques. With more sophisticated scheduling policies being proposed, its potential performance could be improved even further.

Acknowledgments

This research is supported by an Australian Research Grant DP0987236 and an NSF Grant CNS-0834599.

References

[1] H. Akkary and M. A. Driscoll, “A dynamic multithreading processor,” in *MICRO-31*, 1998, pp. 226–236.

[2] M. K. Chen and K. Olukotun, “The Jrpm system for dynamically parallelizing Java programs,” in *ISCA'03*, 2003, pp. 434–446.

[3] M. Cintra and J. Torrellas, “Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors,” in *HPCA'02*, 2002, p. 43.

[4] L. Codrescu, D. S. Wills, and J. Meindl, “Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications,” *IEEE Transaction on Computers*, vol. 50, no. 1, pp. 67–82, 2001.

[5] J. Dou and M. Cintra, “A compiler cost model for speculative parallelization,” *ACM Trans. Archit. Code Optim.*, vol. 4, no. 2, p. 12, 2007.

[6] L. Hammond, M. Willey, and K. Olukotun, “Data speculation support for a chip multiprocessor,” in *ASPLOS-VIII*, 1998, pp. 58–69.

[7] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, “Min-cut program decomposition for thread-level speculation,” in *PLDI'04*, pp. 59–70.

[8] ——, “Speculative thread decomposition through empirical optimization,” in *PPoPP '07*. ACM, 2007, pp. 205–214.

[9] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos, “Tight analysis of the performance potential of thread speculation using spec cpu 2006,” in *PPoPP'07*, 2007, pp. 215–225.

[10] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, “The bi-mode branch predictor,” in *MICRO-30*, 1997, pp. 4–13.

[11] 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*, 2006, pp. 158–167.

[12] P. Marcuello and A. González, “Thread-spawning schemes for speculative multithreading,” in *HPCA'02*, 2002, p. 55.

[13] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun, “Software and hardware for exploiting speculative parallelism with a multiprocessor,” Tech. Rep. CSL-TR-97-715, 1997.

[14] J. T. Oplinger, D. L. Heine, and M. S. Lam, “In search of speculative thread-level parallelism,” in *PACT'99*, 1999, p. 303.

[15] P. Palatin, Y. Lhuillier, and O. Temam, “Capsule: Hardware-assisted parallel execution of component-based programs,” in *MICRO-39*, 2006, pp. 247–258.

[16] I. Park, B. Falsafi, and T. N. Vijaykumar, “Implicitly-multithreaded processors,” in *ISCA'03*. ACM, 2003, pp. 39–51.

[17] C. G. Quinones, C. Madrile, J. Sanchez, P. Marcuello, A. Gonzalez, and D. M. Tullsen, “Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices,” in *PLDI'05*, pp. 269–279.

[18] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, “Parallel-stage decoupled software pipelining,” in *CGO'08*, 2008, pp. 114–123.

[19] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas, “Thread-level speculation on a cmp can be energy efficient,” in *Proceedings of the 19th annual international conference on Supercomputing*, 2005, pp. 219–228.

[20] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, “Tasking with out-of-order spawn in tls chip multiprocessors: microarchitecture and compilation,” in *ICS'05*, 2005, pp. 179–188.

[21] J. G. Steffan, C.B.Colohan, A.Zhai, and T. C. Mowry, “Improving value communication for thread-level speculation,” in *HPCA'08*, 2002.

[22] J. Y. Tsai and P. C. Yew, “The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation,” in *PACT'99*, 1999, pp. 35–46.

[23] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, “Speculative decoupled software pipelining,” in *PACT'07*, 2007, pp. 49–59.

[24] T. N. Vijaykumar, “Compiling for the Multiscalar architecture,” Ph.D. dissertation, 1998.

[25] T. N. Vijaykumar, S. Gopal, J. E. Smith, and G. Sohi, “Speculative versioning cache,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 12, pp. 1305–1317, 2001.

[26] T. N. Vijaykumar and G. S. Sohi, “Task selection for a multiscalar processor,” in *MICRO-31*, 1998, pp. 81–92.

[27] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T.-f. Ngai, and J. Fang, “Dynamic parallelization of single-threaded binary programs using speculative slicing,” in *ICS'09*.

[28] T.-Y. Yeh and Y. N. Patt, “Two-level adaptive training branch prediction,” in *MICRO-24*, 1991, pp. 51–61.

[29] 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*, 2004, pp. 71–81.

[30] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke, “Uncovering hidden loop level parallelism in sequential applications,” in *HPCA’08*.