Performance Implications of Transient Loop-Carried Data Dependences in Automatically Parallelized Loops

Niall Murphy Timothy Jones
Robert Mullins
University of Cambridge, UK
{niall.murphy,timothy.jones,robert.mullins}@cl.cam.ac.uk

Simone Campanoni
Northwestern University, USA
simonec@eecs.northwestern.edu

Abstract

Recent approaches to automatic parallelization have taken advantage of the low-latency on-chip interconnect provided in modern multicore processors, demonstrating significant speedups, even for complex workloads. Although these techniques can already extract significant thread-level parallelism from application loops, we are interested in quantifying and exploiting any additional performance that remains on the table.

This paper confirms the existence of significant extra thread-level parallelism within loops parallelized by the HELIX compiler. However, improving static data dependence analysis is unable to reach the additional performance offered because the existing loop-carried dependences are true only on a small subset of loop iterations. We therefore develop three approaches to take advantage of the transient nature of these data dependences through speculation, via transactional memory support. Results show that coupling the state-of-the-art data dependence analysis with fine-grained speculation achieves most of the speedups and may help close the gap towards the limit of HELIX-style thread-level parallelism.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Thread-level Speculation, Transactional Memory

1. Introduction

The low-latency on-chip communication offered by multicore processors supports the exploitation of thread-level parallelism (TLP) even when frequent inter-thread communication is necessary. Research has already demonstrated that significant TLP may be extracted automatically even from complex benchmarks such as SPEC CPU [4, 30]. Given the enormous benefits of fully automatic parallelization of complex general-purpose programs, we carefully evaluate the most profitable next steps for boosting performance in order to signpost future research.

The existence of additional TLP is confirmed through a limit study where only dynamic data dependences restrict our ability to exploit loop-level parallelism across loop iterations. At this point it is reasonable to suspect that the shortcomings of current techniques are due to the limits of static data dependence analysis. We show, perhaps counter-intuitively, that this is not the case and that no further gains can be made by improving static analysis.

Achieving performance closer to the TLP limit within an automatic parallelization approach requires the need to exploit the transient nature of dependences. In other words, taking advantage of the observation that dependences are realized with different frequencies, ranging from always to rarely seen. Exploiting this behavior suggests the use of thread-level speculation and we assess the potential of three different code transformations to achieve this. Under the hood, we rely on transactional memory support to provide facilities for recording and validating speculative state.

We find that small transactions lead to better performance gains than large transactions, for both hardware and software transactional memory systems. This is especially pronounced for software transactional memory, where a coarse-grained approach to speculation rarely achieves speedups beyond sequential execution. Further, the vast majority of the speedups are achieved by a fine-grained speculation scheme. A more complex technique that uses profile data at compile time to choose whether to speculate or synchronize each data dependence gains only negligible additional performance. Overall our results suggest that state-of-the-art static data dependence analysis, to identify the code sections to speculate, coupled with a fine-grained speculation system is the most promising direction for future loop-level parallelism extraction.

We start by describing HELIX, a state-of-the-art automatic parallelizing compiler, before assessing the limits of thread-level parallelism it can extract and developing schemes for speculation, to take advantage of the transient nature of data dependences.

2. Background and Motivation

2.1 The HELIX Parallelizing Compiler

HELIX is a compiler able to parallelize loops in sequentially-designed programs by taking advantage of low latency communications between adjacent cores of a single CPU. HELIX distributes subsequent loop iterations on adjacent cores; to preserve dependences between loop iterations (i.e., loop-carried dependences), HELIX creates *sequential segments* which are portions of the loop that execute in loop-iteration order between cores. HELIX has previously demonstrated significant speedups for a number of irregular, sequentially-designed programs, traditionally considered hard to be automatically parallelized [4].

HELIX uses state-of-the-art inter-procedural dependence analyses [5, 10] to detect all dependences in a loop at compile time. Loop-carried dependences are identified and satisfied by creating sequential segments. However, since each iteration of the loop runs in a separate thread, each will have its own private local stack frame and set of registers. Therefore, HELIX does not consider write-

CC'16, March 17–18, 2016, Barcelona, Spain

http://dx.doi.org/10.1145/2892208.2892214

```
1  for (count = 0;
2       /* Start sequential segment 0 */
3       count < weight;
4       /* End sequential segment 0 */
5       count++){
6
7    /* Start sequential segment 1 */
8    /* Global scalar, glob */
9    glob++;
10   /* End sequential segment 1 */
11
12   /* Start sequential segment 2 */
13   for(i = 0; i < factor; i++){
14     /* Global array, A */
15     int tmp = A[factor*(count%16) + i];
16     tmp += count*5;
17     if(tmp%2 == 0){
18       A[factor*(count%16) + i] = tmp;
19     }
20   }
21   /* End sequential segment 2 */
22 }
```

Listing 1: Loop with sequential segments.

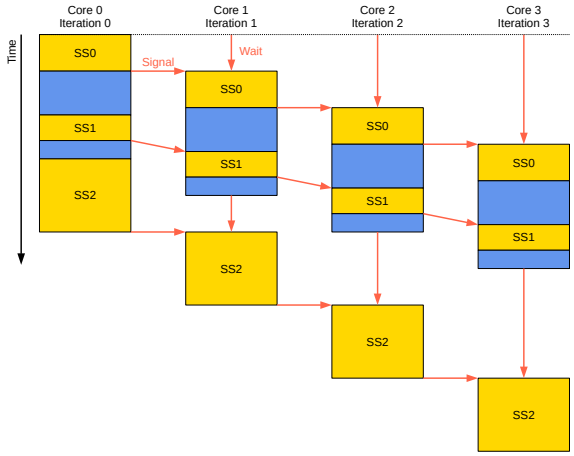


Figure 1: HELIX execution for loop in listing 1. Sequential segments, that could contain loop-carried dependences, are synchronized and execute in loop-iteration order.

after-write and write-after-read loop-carried dependences which occur through registers or the stack. On the other hand, read-after-write loop-carried dependences that occur through the stack or registers are considered and registers are mapped into memory.

To illustrate the nature of HELIX parallelization, consider the code in listing 1. This shows an example loop where each iteration updates a global scalar, `glob`, then an inner `for` loop reads and conditionally updates an index from a global array, `A`. HELIX creates three sequential segments for this loop.

Sequential segment 0 (SS0) contains the loop prologue that HELIX must sequentialize to ensure correct termination of the loop. Here the prologue consists only of a check on the iteration variable, `count`. Since `count` is an induction variable, each thread holds a private copy of it, so there are no dependences within SS0.

Sequential segment 1 (SS1) preserves data dependences on the global variable, `glob`, which is updated on each iteration. Assuming the data dependence analysis that HELIX relied on cannot be certain that loads and stores to this global variable do not alias with other loads and stores, then HELIX does not privatize it in each thread. Therefore, updates to this global variable are executed in loop-iteration order by SS1.

Sequential segment 2 (SS2) surrounds the inner loop because it contains reads and writes to the global array, `A`. Each inner loop iteration computes and reads from an index into the array, conditionally writing to it later. Two iterations computing the same index leads to a data dependence, although this may not often occur, but HELIX is conservative and must respect the uncommon dependence by placing the whole inner loop into a sequential segment.

HELIX mitigates the performance impact of loop-carried dependences by generating separate sequential segments for unrelated loop-carried dependences. For example, consider execution of the code in listing 1, shown in figure 1. The sequential segments are indicated as yellow-shaded blocks, code outside sequential segments is blue. Dynamic instances of each sequential segment execute in loop-iteration order between cores. This means, for example, that only core 0 can execute initially; the others have to wait for their predecessors to finish running sequential segment 0. They are informed of this through a signal from the older thread to the younger.

Although each sequential segment of the same type must execute sequentially, distinct sequential segments can overlap with each other. For example, SS2 on core 0 executes concurrently with SS1 on core 1 and SS0 on core 3. The parallel portions of code can execute concurrently with any code running on the other cores.

2.2 Beyond Static Data Dependence Analysis

Performance obtained by HELIX-generated code is mainly limited by loop-carried data dependences. Improvements to the HELIX compiler, therefore, need to target such dependences either by improving the data dependence analysis to minimize the apparent dependences or by evaluating at run time when such dependences should be satisfied to reduce unnecessary synchronization. Because understanding the potential of each of these directions is essential to focus subsequent research efforts, we evaluate the following aspects of the HELIX-generated code:

Apparent dependences. The imprecise nature of compile-time dependence analysis means that sometimes HELIX identifies dependences which are *never* realized at run-time. If dependence analysis can be improved so that these are not included in the compile-time dependence graph, it may be possible to create fewer or smaller sequential segments, resulting in increased parallelism.

Transient dependences. Some dependences are correctly identified by HELIX and are realized at run-time, but only on a small subset of loop iterations. Since HELIX cannot take advantage of dynamic run-time behavior it must conservatively synchronize these dependences on every iteration. If we can enable HELIX to speculate on these dependences we can extract further parallelism.

Comparing the performance implications of these motivates developments in the direction that holds the most promise. High performance gains by removing apparent dependences would motivate further improvements of the compile-time dependence analysis. On the other, high performance gains by exploiting transient dependences motivates the addition of run-time support in the HELIX-generated code. The next section presents studies which demonstrate that the limits of compile-time dependence analysis have already been reached for HELIX-style parallelization and that we must look to transient dependences for further improvement.

3. Limits of Dependence Analysis

The automatic parallelization community has reached a fork in the road: should we focus our efforts on improving compile-time dependence analysis and purely static parallelization; or should we turn our attention to the dynamic behavior of the program through improved run-time speculation support? The community has spent a significant amount of effort on improving compile-time

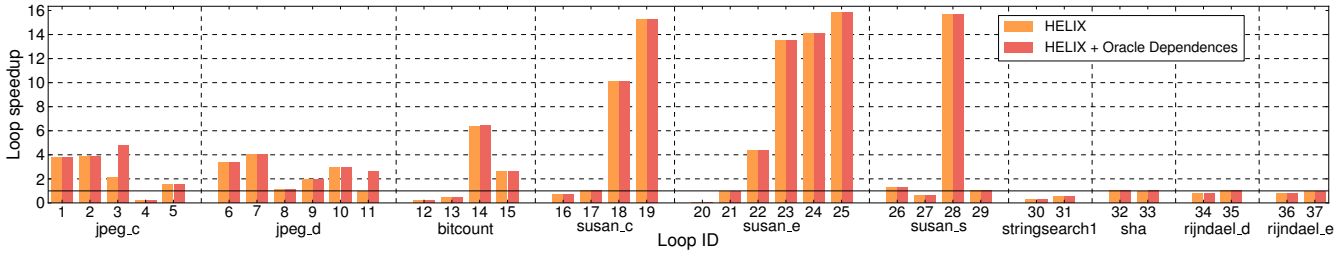


Figure 2: Parallelization with an oracle data dependence analysis shows no improvement. Loop IDs refer to table 2.

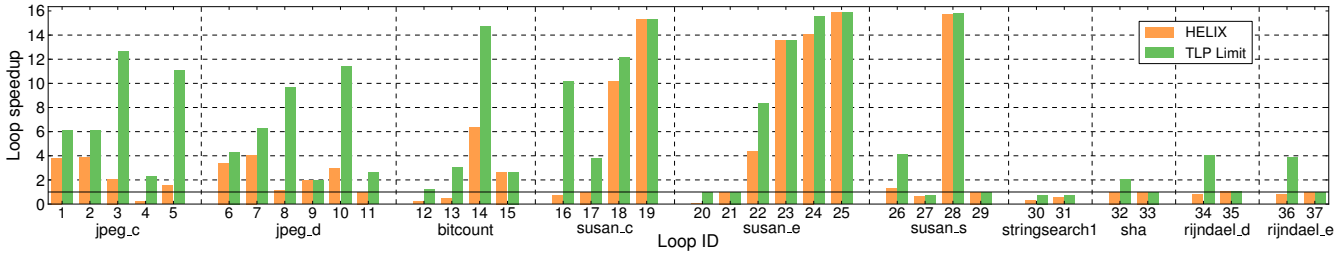


Figure 3: An analysis on the limit of TLP shows that for many loops there is significant additional parallelism to be exploited beyond HELIX.

dependence analysis [10, 15, 18, 19, 23]. Previous work has shown empirically that an improved dependence analysis can enhance the performance of automatic parallelization [27]. While these efforts were justified in the past, we show that this is no longer a limitation for today's compilers because removing all unrealized dependences brings no performance improvement. Further, we show that the transient nature of dependences leaves significant performance on the table, but we can only benefit through run-time strategies.

3.1 Static Analysis

HELIX's dependence analysis is conservative in that it contains data dependences that are unrealized at run-time (i.e., apparent dependences). This is a natural facet of all static data dependence analyses. In general, especially in a language like C, which allows raw access to pointers, it is a challenge for the compiler to determine exactly which memory locations can be touched by a particular instruction. In addition, the compiler is not aware of the dynamic behavior of data dependences, which can vary within different phases of application, or when it is executed with different inputs. It is a requirement that the compiler must, as a minimum, produce correct code so in these circumstances it must be conservative and assume a dependence exist unless proven otherwise. This has been assumed to be a major source of inefficiency in automatically parallelized code, and indeed, Ottoni et al. [27] showed empirically that removing spurious dependences results in improved performance.

We are interested in the extent to which removing these apparent dependences will affect the performance of HELIX. To find the limits of achievable speedups, we simulate a perfect dependence analysis which knows exactly which compiler-identified static dependences will actually exist at run-time. To achieve this we carry out a profiling run of the program, recording every dependence pair seen against the set generated by the compiler. Any dependences unaccounted for at the end of profiling are discarded. What remains are the dependences that a hypothetical improved compile-time dependence analysis must identify. This is the minimum subset of all static dependence pairs that a compiler should detect for a specific program input; in effect, an oracle data dependence graph. Note that even if a dependence exists only between two iterations of the loop, it will still be included in the oracle.

We evaluate the effects of the oracle through parallelization with HELIX. We parallelize all significant loops from a set of cBench applications[1] using HELIX, but replacing the compiler-generated data dependence graph with the oracle. Running the loops both before and afterwards allows us to determine an upper bound on potential speedups for purely static parallelization.

Results of this limit study are shown in figure 2. Despite parallelizing with the oracle, the majority of the loops witness no additional speedups. There are only two exceptions. Loop 3 realizes a performance increase from 2x to 5x. This loop updates an array of RGB values using three pointers which point into different parts of the array. Although the pointers never alias with each other in practice, the compile-time analysis is unable to prove that they would always be independent and so generates a sequential segment to synchronize accesses to the array. The oracle analysis removes this sequential segment enabling greater speedups. Loop 11 achieves a speedup of 3x from 1x due to a similar effect. However, these are outliers and we conclude that for this style of parallelization, compile-time dependence analysis reached its limits to expose TLP.

3.2 Dynamic Behavior

Given the lack of performance gained through a perfectly accurate dependence analysis, we may be tempted to conclude that no additional parallelism exists in these benchmarks. However, the oracle includes *all* dependences, even those that only occur once throughout the execution of the program. Therefore it does not give an accurate account of dynamic behavior.

To quantify the limit of the performance that could be obtained, we created a model that respect dependences only when they actually arise at run-time. This assess the limits of TLP. Each loop iteration is executed on a different core (organized in a ring, as in HELIX). As the loop runs, the model records the memory accesses that occur on each iteration. Stores are assumed to complete immediately, whereas loads must wait for any prior store to the same address to finish execution. The TLP limit model respects all RAW dependences through memory across iterations, but ignores the false WAW and WAR dependences to assess the limits of respecting only the true loop-carried dependences. This limit

[1] More details on our experimental setup can be found in section 5.

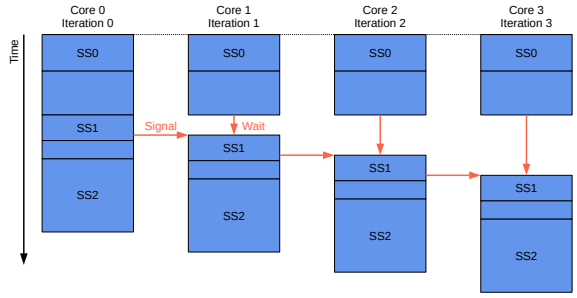


Figure 4: TLP limit execution schedule for loop in listing 1. Only realized dependences are synchronized.

study provides an upper bound on the performance obtainable using HELIX-style parallelization.

Figure 3 shows the results of this TLP limit model, where loop IDs refer to those in table 2. There are broadly three classes of loop. First, those where there is significant parallelism available and HELIX realizes all of the speedup, such as loop 19. These loops are DOALL loops, or DOACROSS loops with few dependences, that HELIX has correctly optimized to gain near-linear speedups with the number of cores. Second, those loops where there is limited performance available and HELIX already achieves the majority of it, for example, loop 9. For these loops, the cross-iteration data dependences that exist are realized on the bulk of the loop iterations, meaning that there is no room for a more aggressive approach to optimize their behaviors. Third, those loops where there is significant performance available beyond the speedups realized by HELIX. Examples of these loops are 14 and 16. Here, the conservative static data dependence analysis forces HELIX to synchronize its sequential segments on every iteration, even when the dependence does not actually exist. Instead, the TLP limit model takes advantage of the transient nature of the dependences, eliding unnecessary synchronizations.

Figure 4 shows an execution of the code in listing 1 using the TLP limit model. All code is executed in parallel, where possible, so it is all shaded blue, but we have picked out the original HELIX sequential segments. The only dependence that actually occurs during execution of this loop is that from sequential segment 1. During execution, all instructions execute as soon as they can, it is only the load in sequential segment 1 that has to wait for the store in the previous iteration to finish, to preserve the RAW dependence through memory. This enables all code from the other two sequential segments to execute in parallel, significantly reducing the amount of waiting carried out by each thread, and therefore the execution time of the loop.

In summary, transient data dependences contribute to significant parallelism not currently exploited by HELIX. In the next section we describe alternatives to take advantage of it.

4. Exploiting Transient Dependences

Figure 3 shows that HELIX could gain significantly more performance by taking advantage of the transient behavior of dependences. However, this analysis only shows the upper *limit* of these speedups: the model assumes zero overhead for the run-time required to support these characteristics. This section, in contrast, discusses practical techniques for extracting the available parallelism. We consider three approaches to run parallel loops speculatively, all using a conflict-resolution scheme based on transactional memory. We explore the trade-offs involved in speculative execution and discuss the required characteristics of an implementation to realize the available performance.

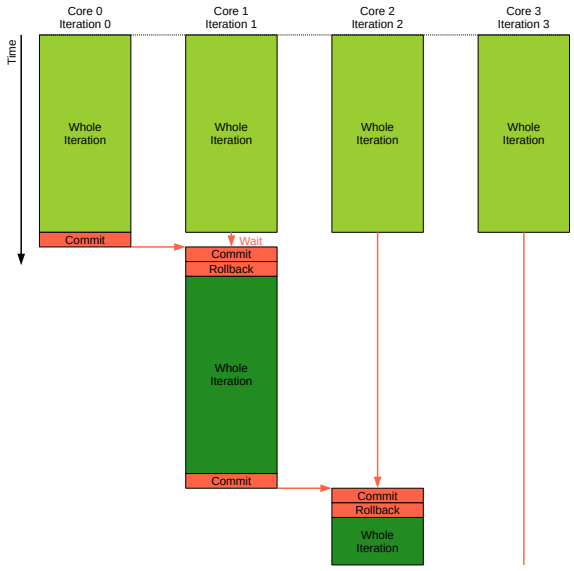


Figure 5: Coarse-grained speculation execution schedule for loop in listing 1. The data dependence in sequential segment 1 causes the whole iteration to be re-executed every time.

4.1 Dynamic Dependence Behavior

To help describe our speculation models we consider an example loop containing different types of loop-carried data dependence. HELIX creates three sequential segments for the code shown in listing 1, as described in section 2.1. To summarize, sequential segment 0 never contains a dependence, sequential segment 1 always contains a dependence, and sequential segment 2 contains a dependence that is only realized between certain iterations of the outer loop (i.e., transient dependence).

When HELIX executes this code, it synchronizes every sequential segment on every iteration. Although this is necessary for sequential segment 1, it means that opportunities for exploiting parallelism are lost when running sequential segment 2 if no loop-carried dependence actually occurs. Closer inspection of the code reveals that the calculation of the index into the global array, `factor*(count%16) + i`, is guaranteed to access a different range of indices in each iteration of the outer loop, provided that the loop is run with 16 cores or fewer. The compiler is unable to prove this and, indeed, does not know the number of cores until run-time, so must be conservative in serializing this part of the code. The transient nature of sequential segment 2 reduces the parallelism HELIX can extract from this loop. In other words, threads wait unnecessarily for their predecessors before entering sequential segment 2, limiting the performance achievable.

4.2 Coarse-Grained Speculation

The simplest speculation model for HELIX places the entirety of each loop iteration into a single transaction. This is a natural extension to the HELIX parallelization model, since each loop iteration is already a unit of work for the parallel threads. In this model, each iteration is run independently of all others and all synchronization is elided. Instead, the run-time records every access to shared memory. At the end of the iteration, the thread waits until it is running the oldest transaction (to preserve the original ordering of loop iterations), then performs conflict detection to commit the speculative state. At this point, if any reads in the current transaction occurred before stores from earlier transactions then a memory RAW dependence has been violated. To address this, the speculative state is discarded and the transaction re-executes the iteration.

Example. With coarse-grained speculation, there are no sequential segments in listing 1 and iterations run concurrently with loads and stores recorded. Its execution is shown in figure 5, with the whole loop iteration included in a single transaction, indicated with a light green box. Since `glob` is read and written on each iteration, when any transaction commits (apart from the first) they identify a loop-carried data dependence violation, so roll back and re-execute the iteration. Figure 5 shows this through the red boxes for commit and rollback, and a dark green box for transaction re-execution. This has the effect of serializing the whole loop, with performance worse than HELIX due to the overheads of the speculation run-time support.

Pros and Cons. The advantage of this approach is that there is no synchronization between transactions (until commit), so truly independent iterations run fully in parallel. In addition, it is simple to implement, since the whole loop body can be placed in a transaction. However, the downside is that even a single dependence between transactions causes the younger to abort and re-execute, effectively serializing the iterations and preventing any speedups from occurring.

4.3 Fine-Grained Speculation

Fine-grained speculation takes advantage of the data dependence analysis HELIX performs by limiting each transaction to the size of a sequential segment. It differs from coarse-grained speculation, which naïvely treats the whole iteration as a single transaction. Due to this, coarse-grained speculation suffers large overheads from recording all reads and writes to memory, even those that cannot be involved in a loop-carried dependence, as well as from rollbacks of the entire iteration when only a single dependence exists.

In the first case, HELIX's inter-procedural data dependence analysis has already proven that memory accesses which occur outside sequential segments cannot cause loop-carried dependences. Therefore these can be executed without instrumentation, without incurring the run-time overhead of recording the access. In the coarse-grained speculation model this is not possible, however, since it is necessary to buffer all writes so they can be rolled back in the case of a conflict. Shrinking each transaction to the size of a sequential segment, and converting each sequential segment into a transaction, as is performed with fine-grained speculation, removes this unnecessary recording overhead outside the sequential segments.

In the second case, HELIX creates sequential segments that are entirely independent of each other. Although no two iterations can execute the same sequential segment in parallel, different sequential segments can be overlapped at will because HELIX has proven them independent. This means that loop-carried dependences are restricted to be within a single sequential segment. However, in coarse-grained speculation, any dependence violation causes rollback and re-execution of the entire iteration, even though it only affects one sequential segment. Since loop-carried dependences are restricted to be within a single sequential segment, it follows that transactions need not be any larger than a sequential segment. This reduces the cost of rollback on a dependence violation, compared to coarse-grained speculation, since the transaction identifies a conflict earlier (at the end of the sequential segment, not the end of the iteration) and only re-executes the current sequential segment, rather than the whole loop body.

Example. The execution of listing 1 using fine-grained synchronization is shown in figure 6. The three sequential segments have been converted into transactions, each committing in loop-iteration order. As before, code outside a transaction is colored blue. Since sequential segment 0 contains no dependence, commit is always successful and there are no rollbacks here. On the other hand, sequential segment 1 always contains a data dependence, so the transactions for this sequential segment always rollback and re-execute. Finally, sequential segment 2 occasionally contains a dependence, so some transactions must rollback, while others commit successfully first time. However, in our example, there are no conflicts for sequential segment 2 because we only show 4 cores, whereas we require over 16 to cause violations.

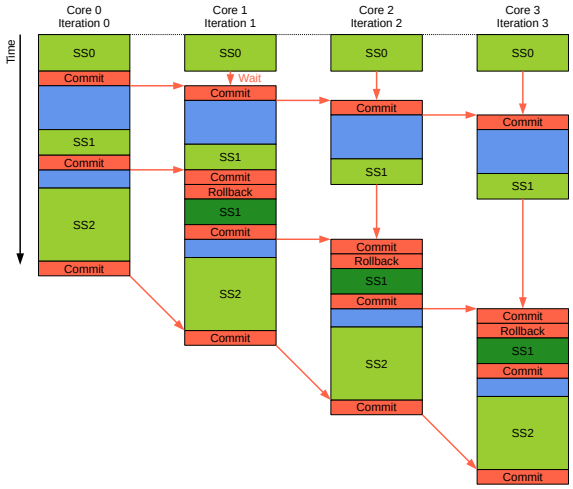


Figure 6: Fine-grained speculation execution schedule for loop in listing 1. The data dependence in sequential segment 1 causes this transaction to be re-executed every iteration, whereas others commit successfully first time.

Pros and Cons. The advantage of fine-grained speculation is twofold. First, it reduces the overhead of bookkeeping, because memory accesses outside sequential segments need not be recorded. Second, it reduces the overhead of rollback, because only a single sequential segment needs re-executing when a dependence violation occurs, rather than the whole loop body. The disadvantages are that loop-carried dependences that are frequently realized cause recurrent rollbacks, meaning that execution can be slower than simply using HELIX's synchronization. In addition, every time a transaction commits, the thread must stall until all threads running older iterations have successfully committed their transactions corresponding to the same sequential segment. This is because the code after the sequential segment is not part of a transaction, so cannot be rolled back if there is a data dependence violation within the prior transaction. Therefore threads must commit each transaction before continuing on to later, non-transactional code. This reduces the extent to which code in different threads can be overlapped in parallel and may lead to multiple stall points in loops with several sequential segments.

4.4 Judicious Speculation

Judicious speculation carefully decides which loop-carried dependences to speculate and which should be satisfied via thread synchronization. This is different to fine-grained speculation, which blindly decides to speculate all loop-carried dependences. Because of this, fine-grained speculation continuously rolls back the sequential segments that often have loop-carried dependences. This leads to wasted work and additional overheads, compared to the original HELIX execution.

To address this, judicious speculation uses profile data to decide whether to speculate or synchronize on a sequential segment. This leads to a judicious application of speculative execution, such that the system never attempts to speculate on code which will regularly fail, but only on when there is reason to believe that speculation will be profitable. Similarly to fine-grained speculation, the sequential

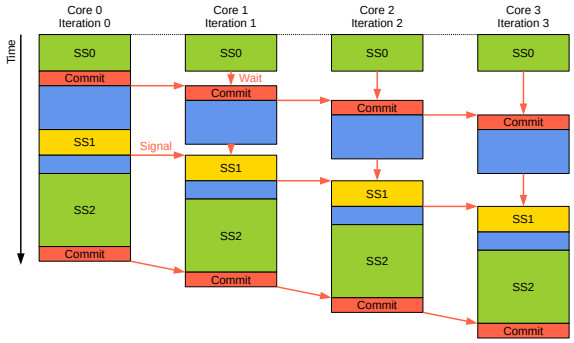


Figure 7: Judicious speculation execution schedule for loop in listing 1. Profiling indicates that SS0 and SS2 rarely contain loop-carried data dependences, so these are speculated; SS1 remains synchronized due to frequent conflicts.

segments previously identified by HELIX provide a logical unit for choosing whether to speculate or not.

Example. Figure 7 shows the execution of judicious speculation for the code in listing 1. Profiling the loop has indicated that loop-carried data dependences rarely exist in sequential segments 0 and 2, whereas sequential segment 1 contains genuine dependences. Therefore sequential segments 0 and 2 are executed speculatively and their write sets are committed to main memory before continuing. Sequential segment 1 is executed using the standard HELIX synchronization primitives. In this case, no transactions need to be rolled back and, despite synchronizing some dependences, the code runs faster than with either full synchronization or speculation, because we gain the best of both worlds.

Pros and Cons. Judicious speculation benefits from all the advantages gained by fine-grained speculation. In addition, and in contrast to fine-grained speculation, performance is robust on always-true dependences since there is no rollback and re-execution. However, the disadvantage is that it is complex to implement, relying on a profiling stage to identify the sequential segments to synchronize and those to speculate. Further, as with fine-grained speculation, it also requires transactions to commit in loop-iteration order before executing non-transactional code, potentially reducing the amount of parallelism extracted.

4.5 Comparing Techniques

Previous sections described three speculative models to augment the baseline HELIX algorithm. Each of them corresponds to a different trade-off between implementation complexity and performance overhead in the presence of always-true dependences. We now compare how these schemes perform on the code in listing 1. Recall that this contains three sequential segments with three different behaviors. Sequential segment 0 contains no loop-carried data dependences, sequential segment 1 contains a dependence that is always true, and sequential segment 2 contains a transient dependence that only occurs when more than 16 threads are used to execute the loop.

We show results from execution of this loop with each of the speculation models in figure 8, using the experimental setup described in section 5 and a hardware transactional memory model. We show performance for varying core counts, up to 16. The HELIX model shows negligible speedup. We would expect a minor performance improvement because the code contains three separate sequential segments, allowing the threads to overlap different parts of the iteration concurrently. However, the proportion of time spent executing sequential segment 2 dominates the overall execution time of the loop, and without being able to parallelize that portion, only a tiny speedup is possible.

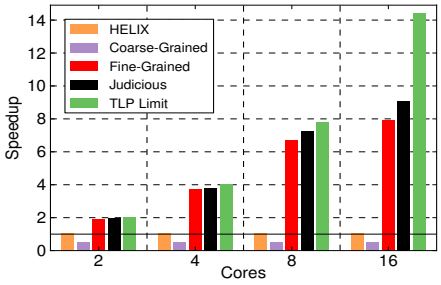


Figure 8: Performance of various models for benchmark in listing 1. Judicious speculation performs better in cases where dynamic behavior cannot be predicted.

Coarse-Grained. The coarse-grained speculation model performs significantly worse than HELIX in this case. No speedup is possible due to the data dependence cycle caused by the update of `glob`. In this case, the model executes the entire iteration speculatively, waits for all previous iterations to complete and then checks for conflicts. On each occasion it detects a memory access violation so the transaction is rolled back and the entire iteration executed again. This means that the entire loop is essentially sequentialized. The performance degradation relative to the baseline is due to the overhead of tracking memory references, conflict checking and rolling back.

Fine-Grained. In contrast, consider fine-grained speculation. In this case, each sequential segment is placed in its own transaction and committed in loop-iteration order. All code outside sequential segments executes as normal, without the overheads of transaction bookkeeping. Sequential segments 0 and 2, which never violate memory dependences in this example, never roll back. Sequential segment 1 always causes conflicts and rolls back, but the quantity of code the needs to be re-executed is limited to the amount in sequential segment 1 only. Fine-grained speculation achieves significant speedups over vanilla HELIX, approaching the TLP limit for low core counts.

Judicious. Finally, the judicious speculation model does not suffer from the fine-grained speculation model's shortcoming of having to speculate on variables which have been proven to cause conflicts. On each iteration, this model synchronizes sequential segment 1 in exactly the same manner as HELIX. This incurs no additional overhead relative to HELIX and no memory references are tracked since threads access this sequential segment in loop-iteration order. Judicious speculation achieves the benefits of speculation, when realized data dependences are rare, and synchronization, when they occur frequently. It achieves slight speedups over fine-grained speculation for this loop; they are minor because sequential segment 1 contains only a small fraction of the loop code.

TLP Limit. Also shown in figure 8 are the results for the TLP limit, as described in section 3. The judicious speculation model does not go all the way to exploiting the speedup that the TLP limit shows is theoretically available. The reason is that the TLP limit model does not suffer the overheads of tracking memory references and performing conflict checking, nor is it restricted by the boundaries of sequential segments. In the speculation models, any conflict within a transaction rolls back and re-executes all instructions, even though the majority of them did not cause a data dependence. Conversely, when synchronizing sequential segments, all instructions in a sequential segment must wait, even those that cannot cause a data dependence. The TLP limit model, on the other hand, synchronizes only the instructions involved in a realized data dependence, allowing all others to execute as soon as they are able.

It is interesting to note from figure 8 that while the judicious speculation model tracks the performance of the TLP limit accu-

rately up to 8 cores, its performance starts to plateau for higher core counts. Unfortunately the model falls foul of Amdahl's Law. While it is possible to speculate on the entire sequential segment, the read set validation and commit phases of the transaction must be performed serially to ensure correctness, limiting the speedups possible.

In this loop, the sequential segments contain a large number of memory accesses relative to the overall amount of code executed. For this example, the validation and commit phases account for around 5% of the entire execution time of an iteration. This gives a maximum possible of speedup of 9x with 16 cores. The density of memory references in a sequential segment is a key determinant of whether or not speculation is likely to be successful.

4.6 Summary

We have presented three models that use speculation to take advantage of transient data dependences and extract parallelism from HELIX-parallelized loops. Our coarse-grained model is the simplest, although a single conflict within the transaction causes the whole loop iteration to be re-executed. Fine-grained parallelism takes advantage of the static data dependence analysis already performed by HELIX, but faces continuous roll-back and re-execution of sequential segments that always incur data dependences. Finally, judicious speculation contains the best of both worlds, synchronizing on frequent dependences and speculating the others, but is the most complex to actually implement. We next describe our experimental infrastructure for evaluating these models before presenting their results.

5. Experimental Infrastructure

We evaluate the opportunities for extracting performance via loop-level parallelization on loops from the cBench benchmark suite and the HELIX timing emulator. cBench was selected since it covers a range of application areas, from security to image manipulation. This section describes the infrastructure we design to measure the performance of the different speculation models previously described.

5.1 HELIX Timing Emulator

The HELIX timing emulator is a profiling tool for exploring the execution of HELIX-parallelized code. It can be used to analyze the interactions between parallel loop iterations and emulate the underlying architecture that the code would be run on. The HELIX timing emulator lowers the barrier for implementing prototypes of architectural features and runtime supports by providing an abstraction of the machine that actually executes the code. It has been validated against the 16-core Atom-based CPU modeled by the cycle-accurate simulator XIOSim [20]. While it would be possible to use real hardware to support speculation, there are currently no commercially available HTM-enabled processors which provide adequate support for the TLS features we are emulating [25].

The emulator is tightly coupled with a pass within the compiler that allows us to insert instrumentation code into an application. HELIX-parallelized compiler IR code is fed into the pass. This IR contains the sequential segments identified by HELIX, as well as all code to execute the parallel loops with multiple threads. The subsequent pass instruments the parallel loops, inserting callbacks at the loop start and end points, at the start of each iteration, at the start and end of each sequential segment, at all memory access instructions, and at the start of each basic block. This enables the emulator to keep track of the parallel loop currently in execution; the in-flight iteration of the current loop; the memory dependences between loop iterations; and the emulated time taken to execute each basic block.

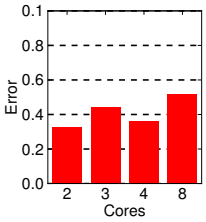


Figure 9: Error.

	TinySTM	TCC HTM
Start	150	0
Load	25 + 3*num_writes	0
Store	32 + 3*num_writes	0
Validate	6*num_reads	0
Commit	66*num_writes	5*num_writes
Abort	0	0

Table 1: Overheads.

After compilation to an actual binary, the application is executed natively. During execution, each parallel loop is run sequentially, but with all the additional code inserted by HELIX and the callbacks for the emulator. The emulator simulates a fixed number of cores, assigning the loop iterations to them in a round-robin fashion, as HELIX would. While an iteration executes, the emulator maintains a time-stamp of the latest operation on that core, as well as a mapping between stored addresses and their time-stamps.

5.2 Modeling Speculation

We created models for each of the techniques in section 4 within the HELIX timing emulator. For speculation, we implemented a simple transactional memory (TM) system with lazy conflict checking, meaning that all reads and writes to memory are recorded and only checked for conflicts with other transactions when the transaction attempts to commit. This model is based on a deferred update TM system, where stores are buffered and only written through to memory on a successful commit. Each transaction contains a hash table to store speculative writes and allow efficient querying for the existence of addresses. This is called the *write set*.

In addition to the write set, the transaction maintains a record of all addresses that were read during execution, along with the times they occurred. This information is stored in another hash table called the *read set*.

When a transaction commits, it must ensure that it has not read any values which were subsequently updated by an older transaction. Once the transaction has completed execution, it searches all older transactions to see if they contain any of the addresses in the current transaction's read set. If the address is found in a prior transaction and this conflicting transaction committed after the address was read in the current transaction, the current transaction must be rolled back and re-executed.

Overheads From our experience of implementing TM systems and from studying existing TM schemes, we have determined that overheads of speculation can be categorized according to six main sources:

Transaction start When the transaction starts there is the overhead of storing the current environment to enable execution to restart from this point, i.e. program counter, stack pointer, live register values. In addition, there might be some other overhead related to setting up required data structures (mainly for software TM).

Transactional load When a transaction performs a load it incurs overhead from adding the address to a read set, checking and recording the current version of the location, and/or recording the current value of the memory location.

Transactional store When a transaction performs a store, it incurs overhead from adding the address to a write set and/or buffering the new value.

Read validation When a transaction attempts to commit, it verifies the read set to ensure that the version of a variable which was read is consistent. This cost is per entry in the read set.

Table 2: Loops extracted from cBench applications, with the fraction of the execution that they represent.

ID	Benchmark	Function	Time	ID	Benchmark	Function	Time
1	jpeg_c	jpeg_fdct_islow	5%	20	susan_e	susan_thin	15%
2	jpeg_c	jpeg_fdct_islow	5%	21	susan_e	susan_thin	15%
3	jpeg_c	rgb_ycc_convert	10%	22	susan_e	susan_edges	18%
4	jpeg_c	encode_mcu_AC_first	10%	23	susan_e	susan_edges	18%
5	jpeg_c	encode_mcu_AC_refine	17%	24	susan_e	susan_edges	56%
6	jpeg_d	jpeg_idct_islow	14%	25	susan_e	susan_edges	56%
7	jpeg_d	jpeg_idct_islow	15%	26	susan_s	susan_smoothing	96%
8	jpeg_d	h2v2_fancy_upsample	18%	27	susan_s	susan_smoothing	98%
9	jpeg_d	h2v2_fancy_upsample	18%	28	susan_s	susan_smoothing	100%
10	jpeg_d	ycc_rgb_convert	21%	29	susan_s	susan_smoothing	100%
11	jpeg_d	decompress_onepass	45%	30	stringsearch1	strsearch	71%
12	bitcount	bit_count	10%	31	stringsearch1	strsearch	84%
13	bitcount	bit_shifter	35%	32	sha	sha_update	78%
14	bitcount	main1	100%	33	sha	sha_stream	97%
15	bitcount	main1	100%	34	rijndael_d	decfile	7%
16	susan_c	susan_corners	7%	35	rijndael_d	decfile	92%
17	susan_c	susan_corners	7%	36	rijndael_e	encfile	7%
18	susan_c	susan_corners	83%	37	rijndael_e	encfile	96%
19	susan_c	susan_corners	83%				

Write commit Once a transaction has validated its read set, buffered writes can be written out to their original intended locations. This cost is per entry in the write set.

Abort When a conflict is detected, the TM must revert the system to its state before the transaction began. For a deferred update system this simply involves clearing out the data structures used to record the transaction's reads and writes. This cost is per entry in the read and write sets.

Relying on this framework, we model two implementations of a TM. The first, TinySTM, is an open-source software TM implementation which supports various TM designs [32]. We determined the overhead parameters for TinySTM by manually instrumenting a sample application with calls to the software TM and measuring the time taken to execute each call with the x86 time-stamp counter (`RDTSC`). The measured values are shown in table 1. We validated the accuracy of the parameterization manually by parallelizing a loop so that its iterations ran speculatively in parallel and comparing the speedups to those predicted by the model. The results of this experiment are shown in figure 9. Although the model overestimates the speedups by up to 50%, the trend as the number of cores increases matches the real application.

The second implementation is TCC HTM, a hardware TM implementation described by Olukotun et al. [26]. Overhead parameters for TCC HTM have previously been determined by the authors and these are shown in table 1.

Judicious Speculation Modeling judicious speculation requires deciding for each sequential segment whether it should be speculated or synchronized. To achieve this, we first run a profiling stage which detects the sequential segments that actually cause conflicts at run-time and how often this occurs. We then choose a threshold for conflicts such that sequential segments with a conflict percentage above the threshold are synchronized and those below the threshold are executed speculatively. In our experiments, we set this threshold to 10%.

Currently the profiling run uses the same input set as is used when executing the program with speculation. In practice it would be necessary to use different input sets to fully evaluate the technique. However, the current evaluation is still useful to gauge the potential of judicious speculation.

For both judicious and fine-grained speculation we implemented a specialized TM model which allows for multiple *domains* of transaction. In this model, conflict detection only occurs between transactions in the same domain and transactions in the same domain must commit in-order, although they can be out-of-order with respect to transactions in different domains. Each static sequential segment created by HELIX corresponds to a single domain. This allows a whole loop iteration to be split into multiple domains and finish in loop-iteration order.

5.3 Benchmarks

We ran the HELIX timing emulator on benchmarks from cBench [7] using the first provided input set. We ran all applications that could be correctly converted into input for HELIX and its underlying compiler. Within each workload, we profiled the loops and selected all those that correspond to at least 5% of the total execution time of the benchmark. These are shown in table 2. To gather results, each application was run to completion, extracting statistics for one loop at a time, to capture all invocations and iterations of the loop.

6. Evaluation

Many loop-carried data dependences are realized only on a small fraction of the loop iterations, which leads to significant performance gains when using a speculation-based approach. After highlighting this, we show that there are only marginal improvements left to be gained by a profile-based scheme that chooses at compile time which dependences to speculate. Finally, we show that there are significant improvements available from choosing what to and when to speculate at run-time.

6.1 Impact of Speculation

HELIX satisfies loop-carried data dependences by synchronizing the execution of sequential segments between cores, as described in section 2.1. Hence, HELIX implicitly assumes that each of these dependences requires synchronization between each pair of adjacent loop iterations. The opposite alternative is to speculate that these dependences do not exist, and therefore let sequential segments run in parallel. This is the fine-grained speculation model described in section 4.3.

The majority of the dependences are rarely realized (i.e., they rarely require synchronization and data movement). Figure 10 shows the performance of HELIX and the fine-grained speculation model implemented with hardware TM, compared to execution of the original (sequential) code. The latter shows significantly higher performance compared to HELIX. Therefore, speculating that most of the loop-carried data dependences require neither synchronization nor data movement is profitable. Coupling this information with the fact that most dependences identified by HELIX are realized at least once at run time (as demonstrated by figure 2), we can conclude that loop-carried data dependences are only apparent dependences most of the time for the benchmarks considered in this paper.

6.2 Transaction Granularity

Identifying the proper granularity of transaction is one of the most important design choices for realizing the benefits of speculative execution. On the one hand, small transactions (e.g., the size of a sequential segment) are appealing both for low bookkeeping requirements (and therefore overhead in terms of both performance degradation and energy consumption) and for reducing the overhead of a miss-speculation. On the other hand, large transactions are appealing for keeping the communication costs of committing transactions low because they naturally tend to have a lower frequency of commits.

Small transactions lead to better performance gains for both software and hardware TM. The fine-grained speculation model relies on transactions as small as sequential segments (we give more detail on actual sizes at the end of this section). On the other hand, the coarse-grained speculation model relies on transactions as big as a whole loop iteration. Figure 10 shows the performance

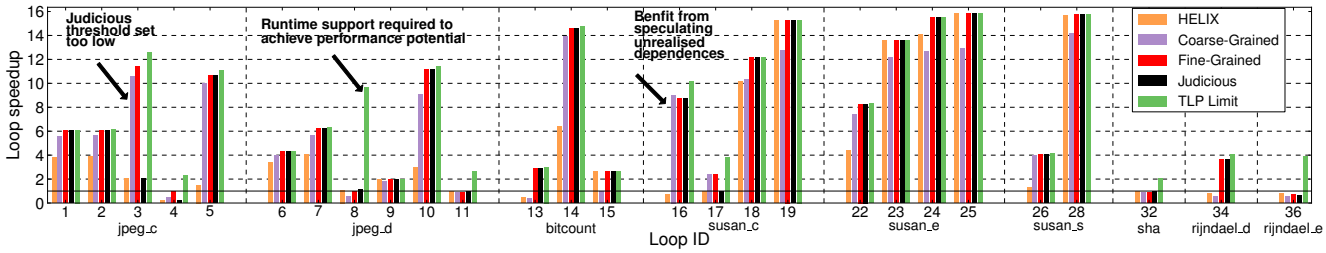


Figure 10: Performance obtained with hardware transactional memory. Fine-grained speculation with hardware support approaches the theoretical maximum speedup. Loops which showed no potential for speedup with the TLP limit model have been removed for clarity.

gains of these two models when hardware TM is used. Instead, figure 11 shows their performance gains when software TM is used. The fine-grained speculation model outperforms coarse-grained in both cases. This result suggests that it is more important to keep transactions small, to keep both the miss-speculation costs and the book-keeping overheads low, rather than focusing on larger transactions, which mainly optimize the communication costs related to commits.

6.3 Choosing What to Speculate

How to choose which dependences to speculate is an important design aspect of a system that enables speculative executions. Simpler solutions lead to less complexity in the implementation, independently of whether it is in hardware or in software. However, more elaborate solutions that rely on profilers require greater design effort to actually implement them.

Compile-time selection The additional performance gained by tuning the selection of dependences to speculate at compile time is negligible for the majority of loops. Figure 10 shows the judicious speculation model, which decides the set of sequential segments (i.e., loop-carried data dependences) to speculate based on profile data. To this end, the program is profiled using the same input to compute the miss-speculation rate per sequential segment. Sequential segments with a rate lower than 10% get speculated; the rest get synchronized as in HELIX. Figure 10 shows that this judicious speculation model obtains performance very close to the fine-grained speculation model, which implements a simpler policy: it speculates every sequential segment. Therefore, the additional complexity of such a profiler is not justified for the benchmarks considered in this paper. The same conclusion is obtained for software TM (see figure 11).

Run-time selection The performance gap shown in both figures 10 and 11 between the fine-grained speculation model and the TLP limit suggests that choosing when to speculate a given sequential segment brings more benefits than performing this choice at compile time. This result also suggests that the transient behavior of loop-carried data dependences requires further run-time investigation to enable the design of an optimal system able to fully take advantage of it. We keep this as a future work.

Data dependence analysis We concluded that simply choosing to speculate all sequential segments brings most of the performance for a compile-time solution. However, this does not imply that data dependence analysis is not useful anymore; this result is enabled by state-of-the-art data dependence analyses carried out by HELIX, which allowed the compiler to slice loop iterations into small sequential segments. Small sequential segments significantly reduced the miss-speculation cost, which enabled the performance benefits of the simple speculate-always policy implemented by the fine-grained speculation model.

6.4 Transactional Memory Implementation

Transactional memories, which are used to enable speculative execution, can be implemented either in hardware or in software. Results shown in figure 10 show that the hardware implementation is significantly more appealing than software, which is assumed for figure 11. The comparison between these figures suggests that hardware TM enables automatic parallelizing compilers to gain significantly more performance than otherwise.

A common constraint for hardware TM is having small size transactions. However, as previously stated, figure 10 suggests that having small transactions is not a limitation for speculative-HELIX-like executions. Because of the importance of this aspect for a hardware implementation, we dedicate the rest of this section to measuring the actual length of transactions for the most promising speculation model discussed in this paper: fine-grained speculation.

6.5 Transaction Sizes

Software TM is fairly robust in dealing with large transactions. TinySTM, for example, resizes the read and write sets for particularly large transactions, but this does not constitute a significant portion of the overall overhead. By contrast, hardware TM is constrained by the fixed sizes of the transactional write buffer or L1 cache. In most implementations of hardware TM, if a transaction exceeds the maximum allowable size it must be stalled until it is safe to run non-transactionally. Obviously this results in serialization of transactions and the complete loss of any possible performance gains. Our fine-grained model assumes hardware support for unbounded transactions and, therefore, we now study the actual sizes of its transactions. We show that these transactions are fairly small and should not generate any performance degradation in an actual hardware implementation.

Table 3 shows the limitations imposed by hardware on the size of transactions for various research and commercial systems. Read state is usually recorded in the L1 cache and is limited by its size. We may reasonably assume an L1 of at least 16KB (although in most modern processors 32KB is expected). Capacity for writes may be shared with reads if they are also buffered in the L1, as is the case for Haswell TSX. Alternatively, a smaller, separate write buffer may be used to make commit more efficient. At the lower end of the scale, Hydra implements a 2KB write buffer. This is generally higher in more recent proposals. TCC values [14] were proposed as the minimum required to effectively support TM across a broad range of applications.

Figure 12 shows the average and maximum sizes for the read and write sets in all the loops we have studied in cBench when using the fine-grained speculation model. Transaction sizes for the judicious speculation model would necessarily be the same size or smaller. The figures indicate that the transactions that have been studied are all fairly small, rarely exceeding 1KB on average. In particular, average write set sizes never exceed 1KB which is a rea-

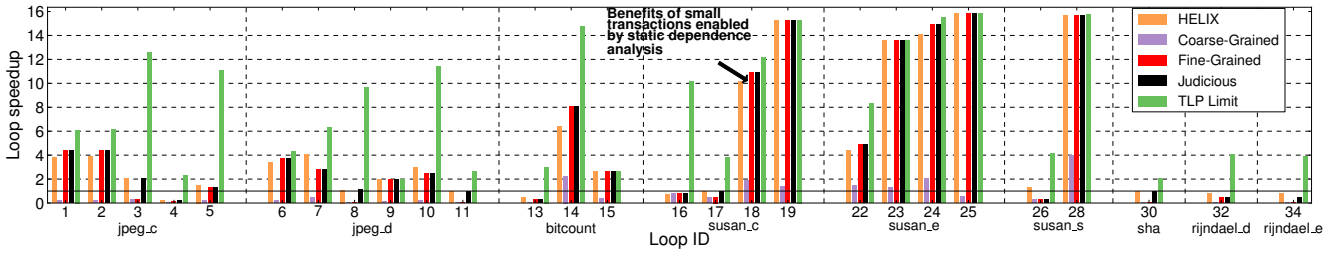
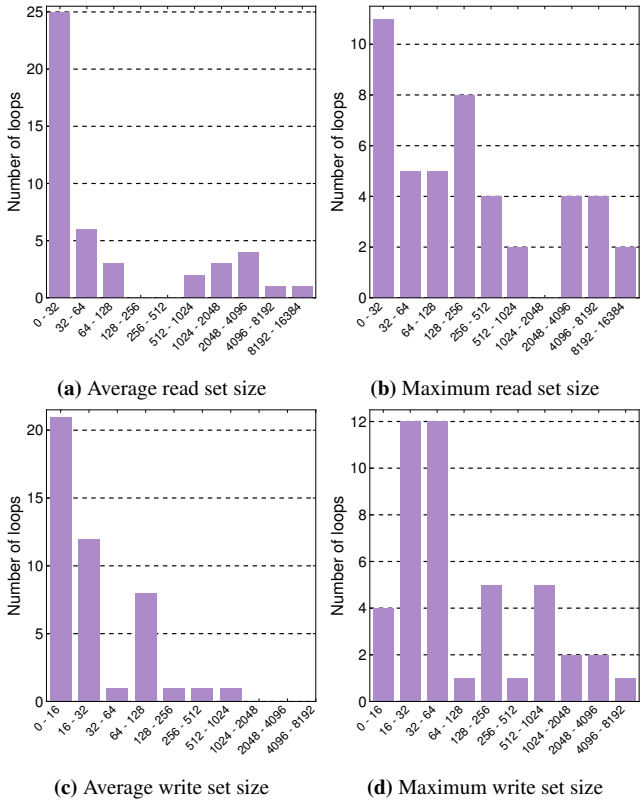


Figure 11: Performance obtained with software transactional memory. While all speculation models have lower performance compared to when hardware transactional memory is used, the coarse grain speculation model performs worst.

Table 3: Hardware resource limitations in current research and commercial hardware transactional memory offerings.

	Per-transaction resources
Stampede [34]	32KB L1 cache (reads and writes)
Hydra [13]	16KB L1 cache + 2KB write buffer
Haswell TSX [39]	32KB L1 cache (reads and writes)
TCC [14]	6-12KB read state + 4-8KB write buffer

sonable write buffer size, according to previous implementations. These results suggest that the performance obtained by the fine-grained speculation model will not be affected by limitations of the hardware and, therefore, that relatively modest architectural support, on the scale of the L1 cache, is enough to enable its significant performance gains.



(a) Average read set size **(b)** Maximum read set size

(c) Average write set size **(d)** Maximum write set size

Figure 12: Histograms of average/maximum read/write set sizes (in bytes) show that a transaction can be accommodated by contemporary hardware supports.

7. Related Work

7.1 Automatic Parallelization

Previous approaches to automatic parallelization of loops without speculation can be split into three categories:

Independent multithreading Also known as DOALL, loop iterations are distributed between threads without any communication between them. This is the most efficient method of parallelization, but is obviously limited to loops which can be transformed to contain no loop-carried dependences [1, 3, 17, 24, 28].

Cyclic multithreading Similar to DOALL but synchronization code is added to the loop body to allow communication between threads. The concept was introduced as DOACROSS by Cytron [8] and subsequent work has been influenced by this technique [6, 16]. HELIX [4] is a generalization of the DOACROSS technique, allowing multiple independent sequential segments per loop iteration. This paper explores the limits of HELIX-style parallelization and evaluates the potential for enhancing HELIX with speculation.

Pipeline multithreading Iterations are broken into stages and these stages are distributed across threads so that data flows through a pipeline between threads [22, 29, 30, 35, 37].

7.2 Thread-Level Speculation

Much prior work exists in the area of thread-level speculation. A number of approaches advocate the addition of dedicated hardware support to reduce the overhead of tracking memory references [13, 33, 34, 40]. Since there are no modern processors which directly support speculation, some authors have implemented speculation purely in software to get some of the performance benefits on currently-available hardware [9, 11, 31, 36]. With the emergence of hardware transactional memory support in some recent processors, there has been interest in using this feature to implement speculation. Odaira et al. [25] implement TLS using Intel TSX [12] and evaluate its performance on SPEC CPU2006. Due to the lack of advanced hardware supports, such as in-order transaction commit and word-based conflict detection, the maximum achievable speedup on Intel TSX is 11%. We model both of these supports and show that greater speedups are possible although it would be necessary to run the same benchmark set to make a direct comparison.

7.3 Limits of Parallelism

A common theme among researchers in computer architecture who have the goal of maximizing performance is to discover the theoretical limits of a particular style of optimization. This is a worthwhile endeavor because it allows us to understand the fundamental power of an idea and to gain insight into the practical limitations it faces. Wall [38] examines the extent of instruction-level parallelism (ILP) available to a superscalar processor and finds that the median ILP is only around 5. Austin and Sohi [2] use dynamic dependence graphs to show that much more parallelism can be extracted than indicated

by Wall. While these studies look at the limits of ILP, Larus [21] describes an execution model to find the limits of loop-level parallelism when exploited in the style of DOACROSS which is similar to our study of the TLP limit.

8. Conclusion

This paper has examined a state-of-the-art automatic parallelization technology, suggesting promising directions for future research. Limit studies showed that improving the compiler's dependence analysis was not sufficient to exploit the additional thread-level parallelism which we know to exist. This was due to the existence of transient data dependences which reduce the parallelism available to a non-speculative parallelizer.

We evaluate three speculation models, simulating both software and hardware transactional memory support. Utilizing the compiler's existing dependence analysis is crucial for reducing the size of speculative transactions. Hardware transactional memory support is necessary for speculation to be profitable since software models add excessive overhead. Our results show that fine-grained, always-on speculation, driven by static compile-time dependence analysis, is most profitable but points towards future research in run-time analysis of the transient nature of the dependences.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) through grant references EP/G033110/1 and EP/K026399/1. Additional data related to this publication is available at https://www.repository.cam.ac.uk/handle/1810/253650.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Pearson Education Singapore, 1986.

[2] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *ISCA*, 1992.

[3] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC*, 2008.

[4] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G. Y. Wei, and D. Brooks. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *CGO*, 2012.

[5] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G. Y. Wei, and D. Brooks. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *ISCA*, 2014.

[6] D.-K. Chen and P.-C. Yew. Redundant synchronization elimination for DOACROSS loops. *IEEE Trans. Parallel Distrib. Syst.*, 10(5), 1999.

[7] cTuning Foundation. cBench: Collective benchmarks. http://www.ctuning.org/cbench, 2015.

[8] R. G. Cytron. Doacross: Beyond vectorization for multiprocessors. In *ICPP*, 1986.

[9] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI*, 2007.

[10] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August. Practical and accurate low-level pointer analysis. In *CGO*, 2005.

[11] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *SC*, 1998.

[12] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, A. J. Martinez, T. Piazza, T. Burton, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, and M. Hunsaker. Haswell: The fourth-generation Intel Core processor. *IEEE Micro*, 34(2), 2014.

[13] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS*, 1998.

[14] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[15] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[16] A. R. Hurson, J. T. Lim, K. M. Kavi, and B. Lee. *Parallelization of DOALL and DOACROSS Loops—a Survey*, volume 45 of *Advances in Computers*, pages 53–103. Elsevier, 1997.

[17] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL*, 1988.

[18] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *SAS*, 2010.

[19] N. P. Johnson, T. Oh, A. Zaks, and D. I. August. Fast condensation of the program dependence graph. In *PLDI*, 2013.

[20] S. Kanev, G.-Y. Wei, and D. Brooks. Xiosim: power performance modeling of mobile x86 cores. In *ISLPED*, 2012.

[21] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parallel Distrib. Syst.*, 4(7), 1993.

[22] I. Lee, C. E. Leiserson, T. B. Schardl, J. Sukha, and Z. Zhang. On-the-fly pipeline parallelism. 2013.

[23] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *CC*, 2006.

[24] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ICS*, 1999.

[25] R. Odaira and T. Nakaike. Thread-level speculation on off-the-shelf hardware transactional memory. In *IISWC*, 2014.

[26] K. Olukotun, L. Hammond, and J. Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

[27] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *MICRO*, 2005.

[28] V. Pankratius, A.-R. Adl-Tabatabai, and W. Tichy. *Fundamentals of Multicore Software Development*. CRC Press, 2011.

[29] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *MICRO*, 2009.

[30] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *PACT*, 2004.

[31] L. Rauchwerger and D. Padua. The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *ICS*, 1994.

[32] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA*, 2007.

[33] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA*, 1995.

[34] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3), 2005.

[35] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO*, 2007.

[36] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO*, 2008.

[37] G. Tournavitis and B. Franke. Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information. In *PACT*, 2010.

[38] D. W. Wall. Limits of Instruction-level Parallelism. In *ASPLOS 91*.

[39] Z. Wang, H. Qian, H. Chen, and J. Li. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *APSys*, 2013.

[40] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative parallelization of partially-parallel loops in dsm multiprocessors. In *HPCA*, 1999.