Technical Report

UCAM-CL-TR-882
ISSN 1476-2986

Number 882

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Discovering and exploiting parallelism in DOACROSS loops

Niall Murphy

March 2016

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2015 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Darwin College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Although multicore processors have been the norm for a decade, programmers still struggle to write parallel general-purpose applications, resulting in underutilised on-chip resources. Automatic parallelisation is a promising approach to improving the performance of such applications without burdening the programmer. I explore various techniques for automatically extracting parallelism which span the spectrum from *conservative parallelisation*, where transformations must be proven correct by the compiler, to *optimistic parallelisation*, where speculative execution allows the compiler to generate potentially unsafe code, with runtime supports to ensure correct execution.

Firstly I present a limit study of conservative parallelisation. I use a novel runtime profiling technique to find all data dependences which occur during execution and build an oracle data dependence graph. This oracle is used as input to the HELIX compiler which automatically generates parallelised code. In this way, the compiler is no longer limited by the inadequacies of compile-time dependence analysis and the performance of the generated code represents the upper limit of what can be achieved by HELIX-style parallelisation. I show that, despite shortcomings in the compile-time analysis, the oracle-parallelised code is no better than that ordinarily produced by HELIX.

Secondly I present a limit study of optimistic parallelisation. I implement a dataflow timing model that allows each instruction to execute as early as possible in an idealistic, zero-overhead machine, thus giving a theoretical limit to the parallelism which can be exploited. The study shows that considerable extra parallelism is available which, due to its dynamic nature, cannot be exploited by HELIX, even with the oracle dependence analysis.

Finally I demonstrate the design of a practical parallelisation system which combines the best aspects of both the conservative and optimistic parallelisation styles. I use runtime profiling to detect which HELIX-identified dependences cause frequent conflicts and synchronise these while allowing other code to run speculatively. This "judicious speculation" model achieves superior performance to HELIX and approaches the theoretical limit in many cases.

Acknowledgements

I would like to thank:

- Robert Mullins, my supervisor, who offered invaluable insight which guided my work, was a constant source of encouragement and was always ready and willing to extend advice and support.
- Timothy Jones who provided many ideas for interesting directions, worked tirelessly to reduce the technical challenges I faced and could always be relied on to find simple solutions to difficult problems.
- Simone Campanoni for providing technical support for the HELIX compiler.
- Kevin Zhou for assisting in the implementation of the speculation timing model and for providing a parameterisation of TLS-STM.
- Kate Oliver for meticulously proof-reading this dissertation and offering much-needed literary advice.
- EPSRC, Cambridge Computer Laboratory and Cambridge Philosophical Society for funding my research.
- Everyone in "lunch club" for supplying a welcome daily distraction.
- Mam and Dad for always being so supportive and giving me the encouragement I needed to keep focussed.

Contents

Chapter 1

Introduction

The "golden age" of the uniprocessor has come to an end. While Moore's Law continues to hold, the era of exponential growth of sequential performance petered out ten years ago. In those ten years we have witnessed the rise of the multiprocessor and with it the emergence of computer architecture's most urgent question: how do we extract the continued performance enhancement we have become accustomed to in the multicore era? This is a multifaceted question. On the one hand we might focus on the programming model, aiming to give the programmer more support to reason about and express the parallelism inherent in the algorithm. We might propose architectural extensions or entirely new microarchitectures which are designed first and foremost with parallel execution in mind. Or we may turn our attention to the compiler and runtime system, hoping to find automated ways to extract parallelism without burdening the programmer.

Developing new parallel programming models is an elegant approach and it is the best way to accurately capture the intent of the programmer. Existing sequential programming models are inherently restrictive since they force the programmer to specify a single correct path through the program and a single correct output, even though for many algorithms multiple outputs would be acceptable. So far, however, parallel programming models have not gained much traction. Parallel programs are difficult to write and difficult to debug and parallel programming is considered a laborious and highly skilled task. Although in many ways we are fighting an established legacy of sequential programming tools, perhaps the human mind is simply less capable of reasoning about parallel computation than sequential.

Architectural extensions which support parallel programming are gradually appearing in commercial microprocessors but the rate of change in this sphere is low. Transactional memory support has only appeared recently in Intel processors despite enjoying plentiful attention in academia for over 20 years. Chip designers are hesitant to make any changes which could be detrimental to sequential performance, for instance, by trading off single core resources in return for greater on-chip core count. Uptake of completely novel parallel architectures is slow because of the difficulty of providing adequate infrastructural support.

So we may decide to focus our attention on extracting parallel performance with the help of the compiler and runtime system: automatic parallelisation. This is a topic which has been considered for many years by researchers and has demonstrated remarkable potential for regular scientific codes where the flow of data can be easily analysed and understood. The feasibility of automatically parallelising general purpose code is still a matter of debate however. Numerous advances, including improvements to dependence

analysis and thread-level speculation, have failed to conclusively crack this problem.

1.1 Conservative versus optimistic parallelism

Broadly speaking there are two approaches to automatic parallelisation. There is the conservative approach which aims to prove the existence of parallelism at compile-time and generate code to exploit it. This method is naturally restrictive because of the intractability of precise memory disambiguation and the lack of runtime information. Even if the static dependence analysis is perfect, it will still indicate a dependence between two static instructions although at runtime that dependence may only exist for a single dynamic instance of the instructions. Then there is the optimistic approach which allows potentially incorrect code to be generated by the compiler which is then executed speculatively. Runtime support is added to ensure that dependence violations are corrected. This takes advantage of the fact that parallelism is easier to *observe* at runtime than it is to *prove* at compile-time. In addition, it has been demonstrated that for some benchmarks parallelism may depend on the inputs to the program and cannot be exploited by the compiler. The downside of optimistic parallelism is that there is some runtime overhead associated with ensuring the correctness of speculative execution.

In this dissertation I will explore the trade off between conservative and optimistic parallelism for a DOACROSS-style [1] automatic parallelising compiler. First I will look at the primary challenge of conservative parallelisation: accurate dependence analysis. I simulate a perfectly accurate "oracle" dependence analysis by constructing the data dependence graph from a profiling run of the program. Using this oracle as input to a parallelising compiler I demonstrate that improved dependence analysis results in no additional speedup above what is possible with the real dependence analysis. I will then answer the question of whether any additional parallelism exists by simulating an ideal dataflow-based machine which extracts the maximum possible parallelism. I show that there is indeed additional parallelism but that it can only be exploited by taking advantage of the dynamic behaviour of the program. Finally I will discuss the decision of when it makes sense to be conservative and when it makes sense to be optimistic. A method will be demonstrated for balancing conservative and optimistic parallelisation within a single iteration of a parallelised loop.

1.2 Contributions

The original contributions of this dissertation are as follows:

- A method to determine the upper bound of conservative parallelism for a program by creating an "oracle" data dependence analysis and a demonstration of the surprising result that improving dependence analysis results in no additional speedup.
- The design of a simulation model to detect the maximum possible parallelism in a program and a study of benchmarks showing that additional parallelism is available which cannot be exploited by the conservative approach.
- A study of transactional memory overheads and a parameterisation of the overheads to allow quantifiable comparisons between implementations.

- A technique for parallelising loops which combines both conservative and optimistic parallelisation.

1.3 Dissertation overview

Chapters 2 and 3 contain primarily prior work. Chapters 4, 5 and 6 are my own original contributions except where explicitly stated otherwise.

- **Chapter 2**: A survey of existing literature on automatic parallelisation. This chapter also suggests a taxonomy of existing techniques and places my contributions in context.
- **Chapter 3**: A description of the HELIX parallelising compiler which is used as a starting point for my work. HELIX was designed and implemented by Simone Campanoni, Timothy Jones and various other collaborators [2, 3, 4].
- **Chapter 4**: Details of the oracle dependence analysis including a novel runtime profiling technique. A quantification of the accuracy of the existing HELIX dependence analysis is shown along with a study of how the accuracy of the analysis affects performance.
- **Chapter 5**: A study of dynamic parallelism and a demonstration that such parallelism exists and requires optimistic techniques to be exploited.
- **Chapter 6**: A discussion of practical techniques to support optimistic parallelisation including a study and comparison of transactional memory overheads in existing implementations. This chapter also presents a method for mixing both conservative and optimistic parallelism within a single loop iteration and a study of how this affects performance.
- **Chapter 7**: Conclusions and suggestions for future work.

Chapter 2

Topics in automatic parallelisation

In this chapter I will look at some of the key issues in automatic parallelisation and other topics which feature in this thesis. Previous work in these areas will be presented and compared in order to demonstrate the key trends which have emerged in this field. Firstly I will provide an introduction to automatic parallelisation and give a brief overview of the long history of this topic, focussing on important related work (section 2.1). My work has primarily investigated speculative techniques to achieve automatic parallelisation and, as such, section 2.2 looks specifically at speculative techniques for increasing parallelism and the transactional memory style systems which are used to achieve this. Section 2.3 contains related work in the areas of dependence analysis and dependence profiling which are used to determine at compile-time and runtime which memory references can conflict. My contributions include two original limit studies which are related to various previous attempts to discover the limits of parallelism and these will be considered in section 2.4. Finally I present a taxonomy of automatic parallelisation approaches in section 2.5 and show how my contributions relate to prior work.

2.1 Automatic parallelisation

Since the advent of multiprocessors, programmers have been faced with the significant challenge of how to take full advantage of the processing power available. While some algorithms are readily amenable to being written in a coarse-grain multithreaded style, many general purpose programs have parallelism which is difficult to express with current tools. Sometimes parallelism is available but it exists in a form which is too complex for the programmer to reason about, for instance, in graph application where the existence of parallelism depends on the structure of the data [5]. In addition, there exists a large body of legacy sequential code which has for years enjoyed incremental performance improvements afforded by the steady advancement of single core execution. These codes do not reap any benefits from the current trend towards an increasing number of on-chip cores.

For a number of years, automatic parallelisation has been seen as the ideal solution to these challenges. Automatic parallelisation removes the burden on the programmer of understanding and expressing the parallelism which exists in the algorithm. Although this topic has been studied for quite some time, a generally applicable solution is still elusive. In this section I will discuss historical approaches to the problem and classify the current techniques to better understand the exploration space. Speculative parallelisation is discussed separately in section 2.2 since it forms an important basis for the work in this

dissertation.

I am exclusively studying loop-level parallelisation techniques here since these are the focus of the vast bulk of the literature and account for a large proportion of execution time for most programs. Approaches to automatic parallelisation are generally classified into three categories which will be discussed in turn:

1. Independent multithreading (IMT)
2. Cyclic multithreading (CMT)
3. Pipelined multithreading (PMT)

2.1.1 Independent multithreading

In this technique, also known as DOALL parallelism [6], every iteration of the loop is executed in parallel completely independently with no communication between threads. The iterations are assigned to threads in round-robin order so, for example, if we have four cores then core 0 will execute iterations 0, 4, 8, 12 etc. (see Figure 2.1). This form of parallelisation is only possible when the loop contains no loop-carried dependences or can be transformed such that no conflicts occur between concurrently executing iterations. Loops which can be parallelised in this manner are likely to experience large speedups since there is no overhead of inter-thread communication. However, the lack of communication also limits the applicability of this technique as many loops will not be amenable to this form of parallelisation. Zhong et al. [7] show that the fraction of execution time covered by loops which can be proven as DOALL at compile-time is only 8% across a range of common benchmarks.

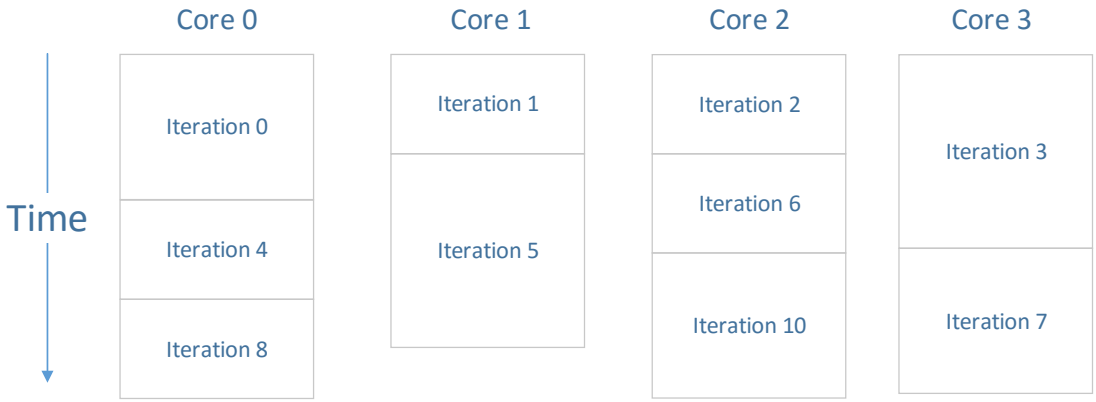

Figure 2.1: Independent multithreading parallelisation.

Unfortunately, loops which are not written with parallel execution in mind can contain dependences which are readily avoidable. In some cases, it is still possible to parallelise loops which have loop-carried dependences in their original form by transforming the loop into some new form in which the dependences are removed or arranged to occur at a sufficient distance that concurrent iterations do not conflict. These optimisations will also be of interest for improving the performance of other forms of parallelisation. Some of these techniques are described here.

Listing 2.1: A loop amenable to DOALL with appropriate transformations.

```
int y = 5;
int sum = 0;
for(int x = 2; x < N; x++){
    int a = A[x];
    a *= y;
    y += 2;
    sum += a;
    A[x] = a;
}
```

2.1.1.1 Privatisation

In some loops we may have loop-carried write-after-read and write-after-write dependences caused by variables which are always written first within a loop iteration. For example, consider the loop shown in Listing 2.1. A naive implementation of a DOALL paralleliser would fail to extract parallelism due to the dependence from reading variable `a` at the end of each iteration to writing it at the start of each iteration. However, it is clear that `a` is a temporary variable and does not cause any true loop-carried dependence.

This can be solved with privatisation of temporary variables [8]. If a variable must always be written first in a loop iteration we may create a private copy of the variable in each thread. This may be achieved by allocating an array to replace the temporary variable. The array is indexed by the thread ID, thus giving each thread a private version of the variable.

Previous work has also shown that arrays can be privatised by adding a dimension to the array [9]. The extra dimension is similarly indexed with the thread ID. This is useful for many programs which process a large amount of data by allocating a block of memory and reading the data block by block, reusing the same allocated memory on each iteration.

2.1.1.2 Induction variable elimination

From a cursory inspection of some simple loops it might be concluded that almost every loop contains at least one loop-carried dependence: the loop induction variable. In listing 2.1, variable `x` will be incremented in every iteration resulting in a read-after-write loop-carried dependence. Without a means to eliminate this class of dependences, there would be very few loops to parallelise.

Fortunately, there is an analysis known as *induction variable elimination* which can successfully remove many dependences of this form [10]. First, imagine introducing a new variable, i, which is equal to the number of iterations previously completed, i.e. an iteration counter. It is then possible to privatise i to each parallel thread by initialising it to the thread ID and incrementing it by the number of threads on each iteration. Then the loop-carried dependence caused by `x` can be removed by privatising it and calculating its value as a function of i in each iteration: $x = i + 2$.

Any variable, j, which can be expressed as a linear function of i can be eliminated in this way, i.e. $j = c_0 i + c_1$ [10]. For instance, the variable `y` in listing 2.1 can by expressed as $y = 2i + 5$. This variable can therefore be privatised in a similar manner to `x` and loop-carried dependences on this variable are eliminated.

Listing 2.2: Example loop in listing 2.1 following transformations.

```
int sum = 0;
int i[num_threads], private_a[num_threads], private_x[num_threads],
    private_y[num_threads], private_sum[num_threads];

/* FORK THREADS */
int tid; /* Initialised to thread ID */
i[tid] = tid; /* Local to each thread */
private_sum[tid] = 0;
while(i+2 < N){
    private_x[tid] = i[tid] + 2;
    private_y[tid] = i[tid]*2 + 5;
    private_a[tid] = A[private_x[tid]];
    private_a[tid] *= private_y[tid];
    private_sum[tid] += private_a[tid];
    A[private_x[tid]] = private_a[tid];
    i[tid] += num_threads;
}
/* JOIN THREADS */

for(int t = 0; t < num_threads; t++){
    sum += private_sum[t];
}
```

2.1.1.3 Reduction

Reductions are operations which reduce an array to a single value. Common example of reductions include summing the contents of an array, finding the maximum value in an array and counting the number of elements of a particular type in an array. All of these operations are implemented with a single variable to hold the reduction and a loop which iterates over the array, updating the variable. This variable will therefore cause a loop-carried read-after-write dependence to occur. The variable `sum` in listing 2.1 is an example of a reduction.

Any computation of this sort in which the reducing operation is both commutative and associative can be converted into a number of thread-local reductions and a final combining reduction once the loop completes [11]. In the example, we can privatise `sum` by creating an array of accumulators with one for each thread. Each iteration's reduction operation then accumulates into its thread's private accumulator. An additional loop is added after the main loop to sum all the private accumulators back into the original `sum` variable.

Having applied privatisation, induction variable elimination and reduction optimisation we can now show the example loop in a form suitable for IMT parallelisation. Listing 2.2 shows the transformed version of the loop. Arrays have been allocated for each privatised variable to eliminate conflicts and induction variables (`private_x` and `private_y`) are now calculated privately as a function of the loop counter. The comments indicate the point at which parallel threads are forked and joined and each parallel thread runs the while loop. To implement the reduction to variable `sum` an additional loop has been added subsequent to the joining of the parallel threads.

2.1.1.4 Historical IMT compilers

Much of the work in this area focusses specifically on regular scientific codes written in Fortran such as SPECFP and NAS. These codes have simpler dependence patterns

than many general purpose programs. Stanford's SUIF compiler [12] is an influential example of a system which made significant advances in analysis for parallelisation of such codes. This work introduces a comprehensive suite of interprocedural analyses to enable detection of parallelisable code. SUIF has considerable success at parallelising loops in SPECFP and NAS, specifically as a result of applying interprocedural analysis. This allows privatisation and reduction analyses to be applied across procedure boundaries which increases coverage of the technique to very large loops with coarse-grain parallelism. Interprocedural analysis is generally indispensable if we wish to target the largest loops in a program [13].

Polaris [14] is another influential parallelising compiler contemporary to SUIF. This compiler features many of the same parallelising techniques and optimisations but is focussed specifically on parallelising FORTRAN programs. Polaris also implements a less comprehensive suite of interprocedural analyses, instead relying on method inlining and interprocedural value propagation (IPVP) [15]. IPVP identifies when a method parameter has a specific value depending on its call site and creates a copy of the method for each value of that parameter. This allows more effective analysis of the method since one of the parameters has effectively been removed.

2.1.2 Cyclic multithreading

While the transformations described previously can be effective for increasing the coverage of IMT parallelisation, the requirement of having absolutely no dependences between loop iterations restricts the general applicability of this technique. Non-removable loop-carried dependences must be executed in loop-iteration order, which means that instructions involved in these dependences cannot be executed in parallel. However, a loop which contains such dependences may still exhibit significant parallelism in the rest of the code. To exploit this parallelism, a more sophisticated parallelisation technique is needed which enforces sequential execution of dependence-related code but permits other code to run in parallel. This technique is known as cyclic multithreading (CMT).

A CMT paralleliser, like IMT, assigns iterations to threads in a round-robin fashion. The optimisations described to increase parallelism in IMT loops are also available in CMT. Since CMT parallelisation may add significant communication overhead to synchronise dependences, it is important to take full advantage of these optimisations to minimise the amount of communication required.

An early and influential example of CMT is DOACROSS [1]. In this technique, dependences are identified by the compiler and the start of each loop iteration is delayed until all dependences from previous iterations have been satisfied. In this manner, the parallel portion of one iteration is overlapped with the sequential portion of the subsequent iteration resulting in parallel execution. For example, in figure 2.2 the statement `x = x->next;` causes a loop-carried dependence since it cannot be evaluated until the statement has completed in the previous iteration. Synchronisation is inserted to ensure sequential execution of this statement across cores while the rest of the iteration can run in parallel. Once all cores have started their first iteration, this technique can still approach linear speedup if the parallel portion of the loop is sufficiently large to allow full utilisation of the cores.

A more recent advance in this space is HELIX [2]. HELIX generalises the above approach to DOACROSS loops by creating multiple synchronisation points per loop.

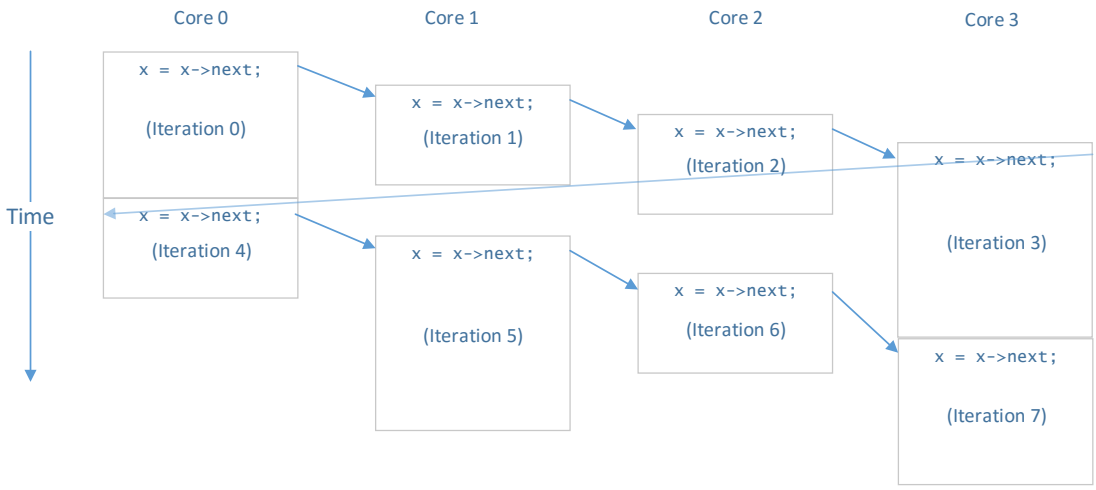

Figure 2.2: Cyclic multithreading parallelisation.

These synchronisation points are called sequential segments and ensure that the code involved in a particular dependence cycle executes in loop-iteration order across cores. HELIX exploits more parallelism than earlier DOACROSS techniques by allowing different sequential segments to execute in parallel. Furthermore, HELIX applies a more general set of optimisations and transformations than SUIF which enables it to achieve substantial speedups on irregular codes. HELIX is further discussed in chapter 3 and forms the baseline for the contributions in this dissertation.

2.1.3 Pipelined multithreading

Pipelined multithreading (PMT) is an alternative method for parallelising loops with cross-iteration dependences. In this approach, the loop body is divided into a number of pipeline stages with each pipeline stage being assigned to a different core. Each iteration of the loop is then distributed across the cores with each stage of the loop being executed by the core which was assigned that pipeline stage. Each individual core only executes the code associated with the stage which was allocated to it. For instance in figure 2.3 the loop body is divided into 4 stages: A, B, C and D. Each iteration is distributed across all 4 cores but each stage is only executed by a single core.

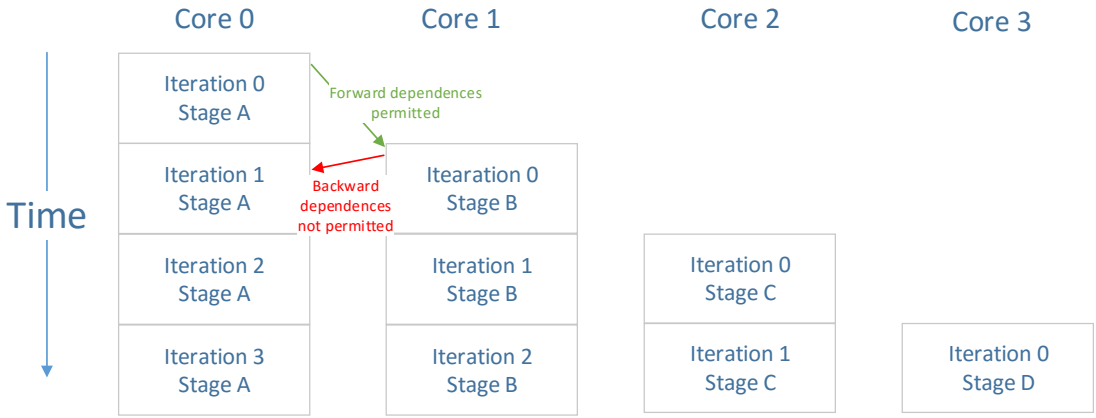

Figure 2.3: Pipelined multithreading parallelisation.

DOPIPE is one entry in the design space of PMT [16]. This approach is mainly targeted at scientific applications with regular dependence cycles [8]. Since the different pipeline stages often contain largely independent code, DOPIPE can be more effective than DOACROSS in reducing the amount of inter-core communication. Parallelising in this manner may be more restrictive than DOACROSS however, since the dependences between threads must be unidirectional, i.e. a pipeline stage can only communicate data to a later pipeline stage.

The state-of-the-art in PMT for general purpose programs is decoupled software pipelining (DSWP) [17]. This technique is similar in style of execution to DOPIPE but is more general in that it can handle irregular dependence patterns and control flow [18]. DSWP automatically generates pipeline stages by finding strongly connected components in the program dependence graph (PDG), a graph combining both data and control dependences. Strongly connected components can be combined into a single pipeline stage to offer better load balancing but cannot be split into multiple pipeline stages since this would result in cyclic dependences between the threads.

DSWP suffers from a scalability challenge not directly faced by CMT techniques like HELIX since the number of threads is limited by the number of pipeline stages that can be created for the loop. HELIX is only restricted by the number of loop iterations, which is usually much larger. Further work in DSWP has attempted to alleviate this by applying IMT or CMT style parallelisation to the individual pipeline stages [18]. This allows a pipeline stage to be replicated across multiple cores so that the stage can be executed for multiple iterations concurrently.

While DSWP targets general purpose programs, pipeline parallelism is also commonly used to accelerate streaming media applications [19, 20, 21]. Pipelining is suitable for such applications since they often consist of a number of stages operating on the input data in sequence with the individual stages being independent from each other. Tournavitis et al. [20] propose a framework for taking advantage of such parallelism by profiling the program and presenting the suspected parallel regions interactively to the programmer. Profiling enables the extraction of coarser parallelism opportunities than would be afforded by the purely static analysis of DSWP. A further advance in DSWP is the addition of speculation [22] which goes some way to addressing this shortcoming.

2.1.4 Programmer-guided techniques

While extracting thread-level parallelism from sequential programs and completely freeing the programmer from the concern of parallel programming is an attractive prospect, some researchers have opined that the sequential model is simply too restrictive to permit the extraction of coarse-grain parallelism. At the same time, the adoption of parallel programming tools such as OpenMP has been tentative due to the difficulty of using such tools and the resultant increase in development time and costs. Therefore there has been some interest in finding a middle ground: providing the programmer with some simple primitives to indicate parallelism opportunities while applying a fully-featured automatic thread-extraction framework which takes care of the intricacies. Often this takes the form of permitting a degree of indeterminacy in the program such that multiple possible outcomes are allowed, a scenario not describable in the sequential model.

Bridges et al. [23] propose the addition of two annotations to the sequential programming model: Y-branches and commutative functions. Y-branches are conditional state-

ments where executing down the *incorrect* path of the branch does not result in an invalid outcome. Surprisingly, it has been shown that 40% of dynamic branches in SPECINT 2000 satisfy this property [24]. Whereas in the sequential model the unpredictable control flow may limit the compiler's ability to extract parallelism, by indicating to the compiler that one path of the branch is always permissible, it is possible to rearrange the execution in a manner which is more amenable to automatic parallelisation.

Commutative functions are those for which calls can be rearranged in any order without affecting the outcome. The classic example of such a function is `malloc`. `malloc` has persistent state which is updated by each call, but calls can still be reordered without affecting program correctness. Automatic parallelisers often struggle with such functions because the update of persistent state means that dependences will be detected and the compiler will endeavour to maintain the sequential ordering of calls to the function. However, if the programmer annotates the source to indicate that a function is commutative, the compiler will be able to relax its constraints relating to that function and extract more parallelism.

A somewhat more invasive approach to programmer-guided parallelisation is the Galois system [25]. Galois is targeted at graph-based irregular programs which exhibit *amorphous data parallelism* [5]: a general form of data-level parallelism where there is no specific order in which the data must be processed and the parallelism is heavily dependent on the input set. To take advantage of this parallelism, the programmer may use *optimistic iterators*: a Galois construct which abstracts away the scheduling of iterations over a graph. The system then invokes worker threads to execute the iterations speculatively in parallel, scheduling them in a manner it sees fit to achieving optimum load balancing and parallelism. Galois offers the programmer a means to express complex data-dependent parallelism in the program while taking care of the details of spawning threads and resolving data dependence conflicts. However, it requires substantial re-writing of the program and thus does not embody many of advantages of pure automatic parallelisation.

2.2 Speculative execution

The automatic techniques described above all suffer from one major limitation: inability to predict and exploit dynamic runtime behaviour. Certain benchmarks, in particular those which operate on graph data structures, have input dependent control and dependence behaviour which cannot be exploited by parallelisation methods which create a static schedule of threads and dependence synchronisations. Consider the example in listing 2.3. We will assume that the initial contents of array `A` are read from a file as input. It is not possible to determine anything about the index variable, `t`, at compile-time because its value depends on the contents of the array. The compiler must assume that a dependence exists as otherwise it might generate incorrect code. However, with knowledge of the possible input data values, we may be able to say that a dependence never exists, or rarely exists. To take advantage of the parallelism caused by this input-dependent, dynamic behaviour, we must use speculative execution. In addition, static dependence analysis is difficult to implement, even when dependences can theoretically be determined at compile-time. Speculation can reduce the burden on the compiler of having to perform precise dependence analysis.

To execute this loop with speculation, we would need to track the addresses touched during the execution of each iteration. If two iterations touch the same address, one

Listing 2.3: Loop which cannot be parallelised without speculation.

```
for(int i = 0; i < N; i++){
    int t = A[i];
    A[t]++;
}
```

or both of the iterations must be re-executed to prevent incorrect output. It may be noted that this characteristic of speculation is similar to the operation of transactional memory (TM). Indeed, many speculation proposals include an implementation of what is essentially a TM system. Therefore I will first discuss prior work on TM to understand how this technique is used to support speculation in section 2.2.1. Then I will look at a number of specific entries in the speculation design space in section 2.2.2.

2.2.1 Transactional memory

TM was first proposed by Herlihy and Moss as a hardware extension to support lock-free data structures [26]. The concept is based on the widely used transaction model of databases. In this model, the database is only accessed by transactions which conform to the ACID properties:

- Atomicity: Transactions must appear to either run entirely or not run at all. Transactions cannot half complete.
- Consistency: The transaction must leave the data in a consistent state with respect to the constraints of the application.
- Isolation: Transactions must not observe the intermediate state of another transaction.
- Durability: Once a transaction has committed, its results must be stored permanently on disc.

Database transactions support high levels of concurrency for access to a shared memory structure, while shielding the programmer from the complications of parallelism. It is therefore an attractive model to reproduce in the multiprocessor programming world where these are the specific challenges being faced. The properties of transactions for multiprocessor programming are usually referred to as ACI since durability is not relevant to the volatile main memory of a processor.

2.2.1.1 Hardware transactional memory

Early proposals for TM were generally based on hardware extensions to the architecture to support conflict detection, rollback and re-execution [26]. Hardware transactional memory (HTM) techniques generally take advantage of an existing mechanism in the architecture for detecting conflicting memory references: the cache coherence protocol. Herlihy and Moss [26] propose a *transactional cache*, separate from the main data cache, which stores data read and written transactionally. When a read or write occurs within a transaction, the cache coherence bus is used for early detection of conflicts with other

transactions. More recent entries in this space, including Intel's implementation of transactional synchronisations on the Haswell architecture [27], use a single L1 cache for both transactional and non-transactional memory accesses [28]. Cache lines are extended with extra bits to indicate the transactional state of the line.

Hammond et al. [29] go so far as proposing a system where TM is the *only* memory model to satisfy coherence and consistency. This proposal has many advantages in simplifying the cache coherence protocol and in providing a clear and accessible memory consistency model to the programmer. However, apart from IBM's Blue Gene/Q [30] and POWER8 [31] processors, HTM has not yet received widespread adoption by the industry. This may change in the near future with Intel having committed to providing transactional support. In the meantime, researchers have looked at implementing TM in software to provide the same functionality on currently available hardware.

2.2.1.2 Software transactional memory

Software transactional memory (STM) has developed in parallel with HTM, with researchers hoping to experiment with and increase adoption of transactional memory on real systems. The term was coined by Shavit et al. shortly following Herlihy and Moss's influential HTM proposal [32]. STM is generally more flexible than HTM and is less limited by the size of the hardware structures used in HTM, but suffers from a significant performance overhead due to performing conflict checking in software. In general, the decision when designing an STM as to whether to use deferred update or direct update leads to two major directions in implementation [33].

Deferred update A deferred update system uses a buffer to store writes made during the transaction such that they are not visible to other transactions. Committing the results to main memory is deferred until the transaction completes. Shavit and Touitou's STM [32] is deferred update and bears the significant constraint that the programmer must specify all memory locations are accessed by the transaction. A later proposal by Herlihy et al. [34] introduces a more flexible implementation which removes this constraint. This STM is written for C++ and Java and provides transactional semantics at an object granularity. When a transactional object is used in a transaction, a local copy is made and modifications are made to the local copy. The transactional object itself records whether the object is currently open in a transaction and the locations of the old and new versions. When the transaction commits, assuming no conflict occurs, the transactional object is simply marked as committed and the local copy becomes the new definitive version of the object.

Herlihy et al. also employ a commonly used model of contention management where a variety of contention managers can be plugged into the STM through a generic interface. Contention management is concerned with the decision of how to resolve the situation where two transactions conflict. Considerable research has been done specifically into the design of the contention manager, and policies usually take into account some combination of transaction timestamp, amount of work done and which transaction first accessed the conflicting resource [35].

Direct update Deferred update systems commonly experience significant overhead related to making local copies of variables and committing these local copies once the

transaction completes. An alternative approach has been explored more recently where objects are updated in place by the transaction [36, 37]. This generally requires a mechanism for locking memory locations which have been written so that other transactions cannot modify them and a logging mechanism for recording the old value of a location so that it can be restored in the case of rollback. Direct update systems reduce the cost of committing a transaction since all memory has already been updated in place, but increase the cost of aborting a transaction since memory must be restored to its original state.

2.2.2 Thread-level speculation

The difficulty of memory reference disambiguation has long been considered a significant challenge to enhancing performance with automatic parallelisation. Thread level speculation (TLS) is an attractive solution to this issue since it moves the burden of disambiguation from the compiler to the runtime where considerably more information is available and analysis is conceptually more straightforward. As a result, TLS has a long history of discussion in the literature and a wide variety of approaches have been proposed. Speculative threads were first proposed by Knight [38] as a means to parallelise the execution of functional programs containing difficult to analyse side-effects. This work introduced a number of core concepts including the use of a an extra cache to buffer speculative writes (referred to by Knight as the *confirm cache*) and the abortion and re-execution of speculative blocks.

Other early work stems from pushing the limits of instruction-level parallelism (ILP) in superscalar processors. The first implementation of an architecture explicitly designed to support speculative execution at the task-level[1] is the Multiscalar processor [39, 40]. The insight leading to this work is that, as the transistor density and architectural complexity of chips grow, it will become increasingly challenging to scale the performance of centralised resources, in particular the instruction window used to exploit ILP [41]. In addition, Sohi et al. [42] observe experimentally that as the instruction window is increased, the parallelism which is being exploited is increasingly coming from points that are far apart in the instruction stream. Ultimately this leads to the proposal to split the instruction window so that several processing sites can execute instructions from different segments of the instruction stream (*tasks*) concurrently. Register dependences between these tasks are maintained via register reservations from older tasks to newer tasks [43]. Speculative writes to memory are buffered in an additional hardware feature called the *address resolution buffer* [44] where they are tagged with the identification of the processing unit which created them so that later tasks can detect if a dependence has been violated. Selecting tasks is itself an important problem and Vijaykumar et al. [45] describe a number of heuristics used by the compiler including task size, existence of control flow and existence of data dependences.

These initial proposals on speculation at the task-level have spawned a wide variety of related work in the area which will be discussed in this section. While many of these

[1]Here I use the phrase *task-level* to emphasise the distinction between this work, which speculatively executes code segments larger than a basic block, and the more fine-grain exploitation of ILP in conventional superscalar processors. However, the approach used in the Multiscalar work is more similar to research on exploiting ILP than to the research previously discussed on automatic parallelisation. The phrase *thread-level* is used to refer to parallelisation techniques focussing on high-level program structures such as loops, although in some literature these threads are referred to as tasks.

projects have followed in same vein as Knight and Sohi et al. by advocating additional architectural resources, others have attempted to make TLS available through software support. I divide the research into broad categories depending on whether it is purely software-based (section 2.2.2.1) or if additional hardware support is suggested (section 2.2.2.2).

2.2.2.1 Software-only speculation

Contemporary with the Multiscalar proposal of Sohi et al., Rauchwerger and Padua proposed the LRPD test [46]. This work attempts to increase the coverage of DOALL parallelisation techniques for Fortran programs by speculatively applying the compiler optimisations described in section 2.1.1 such as reduction and privatisation. Speculative state is buffered in software and the entire loop is executed in DOALL style (i.e. one iteration per core with no inter-iteration communication). Upon completion of the loop, a test is run to verify the validity of the parallel execution based on the *loop privatising DOALL* (LPD) test [47]. This test maintains shadow arrays for each array used in the loop which record if any element is read before it is written in an iteration, thus indicating that the array cannot be privatised. The test is extended to also verify reduction operations at runtime (the LRPD test).

Rauchwerger and Padua's approach is only suitable for regular scientific applications where the speculative optimisations take advantage of only a small range of specific patterns where array indices cannot be disambiguated statically. Furthermore, this work suffers a very large performance penalty for misspeculation since the entire loop must be re-executed sequentially.

Subsequent work in this style focuses on trying to reduce the cost of misspeculation. Gupta and Nim [48] propose a similar technique which applies parallelising optimisations speculatively but also inserts tests on each iteration which detect, to some degree of accuracy, whether speculation will be successful. This early detection of conflicts greatly reduces the performance hit in the case of misspeculation due to less wasted computation.

A generalisation of the LRPD test technique was presented by Dang et al. [49] which allows parallel execution of a loop even when it contains loop-carried dependences. This scheme uses a sliding window of in-flight iterations with the LRPD test being applied once all the iterations in a window have completed. If loop-carried dependences are detected, the window is simply moved forward to the point of the earliest iteration which executed incorrectly, a new window of iterations is executed (including the misspeculated iterations) and the LRPD test is applied again. This is referred to as the *sliding window recursive LRPD test* (SW-R-LRPD). Cintra et al. [50] further develop this approach by sliding the window each time the oldest thread commits rather than waiting for all threads in the window to complete. This results in better load balancing, since a long-running iteration in the middle of the window does not prevent new iterations from starting.

Subsequent work expands the purview of software-only speculation to include irregular programs. Ding et al. [51] suggest a process-based speculation system known as behaviour-oriented parallelism (BOP). In this technique there is one process which executes the program non-speculatively and other processes which execute possibly-parallel regions speculatively. The speculative and non-speculative processes execute in competition, so that in the worst case, performance should be approximately equal to the original non-parallelised version. A separate competition exists for each possibly-parallel region so that the execution benefits from regions where speculation is profitable but does not

suffer when it is not. BOP uses the virtual memory system to detect conflicts at the granularity of a page. Page fault handlers are installed to detect which pages have been read and written by a process. This has the advantage that once a particular page has been recorded as written, there is no further cost incurred by writing to the page again. Ultimately this makes BOP most suitable for highly coarse-grain parallelism.

A more recent entry in software-only speculation is the Copy or Discard model [52]. This approach has some similarities to those discussed above as it allocates memory space for each speculative thread, allowing them to perform computation without polluting main memory. However the manner in which dependence conflicts are detected is significantly different. Copy or Discard maintains version numbers for all memory locations which are accessed speculatively for each parallel thread. On commit, if the version number in speculative memory does not match that in main memory, the speculative state is discarded and the offending parallel thread is re-executed. This is more general than early software-only schemes which were restricted to detecting conflicts based on specific patterns of array accesses. The generality of the Copy or Discard model allows it to work on general purpose codes including those with dynamic data structures. Copy or Discard also permits parallelisation of loops with loop-carried dependences by executing any code involved in such dependences sequentially at the start of each iteration.

A further refinement of this scheme introduces incremental recovery: the ability to rollback only a portion of a speculative execution and resume [53]. This is implemented by creating multiple subspaces to store speculative state from multiple different regions of the iteration. Then one particular region can be rolled back by simply discarding the state associated with the region. Cao et al. [54] suggest a more general approach which can parallelise recursive algorithms by allowing speculative threads to spawn further threads recursively.

While most recent speculation proposals have been based on maintaining a log of memory accesses to detect conflicts, Privateer [55] reinvents the older LRPD style of speculation by focussing on applying parallelising transformations speculatively. This work performs privatisation and reduction optimisations speculatively on heap-based data structures and verifies at runtime that these are valid. This has the advantage that the overhead of instrumenting speculative loads and stores is reduced since validating that a data structure can be privatised does not require communication, i.e. you just need to check that the data structure was not read before it was written in a given iteration. In addition, STM style speculation does not support speculative reductions. For example, Privateer can speculatively reduce an accumulator which is allocated dynamically even though pointer analysis may not be able to confirm definitively that the accumulator is safe to reduce. By contrast, an unreduced accumulator will always cause conflicts in an STM. Privateer is more general than LRPD since it can speculatively optimise pointer-based data structures as opposed to just arrays.

2.2.2.2 Hardware-supported speculation

Due to the expense of tracking memory references, buffering speculative state and detecting conflicts in software, many proposals in the TLS design space have suggested adding hardware support to accelerate these tasks. While the Multiscalar processor was a completely novel architecture designed around the speculative execution of tasks, later proposals generally suggest more modest architectural extensions to commodity processors. One of the important considerations which drives this view is the desire to minimise

the impact of speculation support on single-thread performance. This constraint limits the scope for adding complex centralised structures which may increase memory access latencies.

One of the earliest schemes in this style is the speculation support on the Stanford HYDRA chip multiprocessor (CMP) [56]. In this work, a speculative coprocessor is added to each core on the CMP. The job of the coprocessor is to execute software exception handlers which spawn speculative threads on other cores. This is in contrast to Multiscalar where speculative thread control is completely managed by the hardware. The HYDRA scheme has the advantage of allowing flexibility in the control of threads while not significantly burdening the main core with executing extra software.

Modifications are also made to the HYDRA cache hierarchy to support speculative write buffering and conflict detection. A *modified* bit is added to the write-through L1 data cache to indicate when a line has been speculatively written and must be invalidated if the thread is restarted. *Read bits* are added to the L1 to indicate that a line has been speculatively read. If a write with the same tag is broadcast on the write bus from a less speculative thread, a dependence violation has occurred and the cache notifies the processor that it must abort. Write buffers are added between the L1 and L2 to prevent pollution of the L2 cache with speculative data.

Contemporary to the HYDRA speculation proposal is a similar scheme developed as part of the Stanford STAMPede[2] project [57, 58, 59]. This scheme suggests even fewer architectural extensions to support TLS. STAMPede eschews HYDRA's addition of a speculation coprocessor and manages speculative threads by adding software to the start and end of each speculative region. The buffering of speculative writes is also simplified in this architecture due to the use of write-back rather than write-through private L1 caches. The writes performed in speculative regions are stored in the L1 cache and tagged with additional bits which prevent such lines from being propagated until the speculative thread commits. The additional bits are similar in functionality to those added in HYDRA and an *epoch id* is piggybacked onto each cache coherence message to indicate the sequential ordering of speculative memory operations.

HYDRA and STAMPede demonstrate the two common approaches to buffering speculative state in hardware: write buffers and private write-back caches. In transactional memory these schemes would both be referred to as deferred update, since they update main memory only when speculation has been confirmed as successful (see section 2.2.1). As in transactional memory where direct update schemes have also been suggested, other TLS proposals work by allowing speculative threads to update the main memory state and maintain an undo log such that modifications can be reversed if a thread aborts. An example of such a scheme is the *Software Undo System* (SUDS) [60]. This is implemented on the MIT RAW architecture [61], a many-core system with hundreds of very small cores integrated on a single chip. The detection of dependence conflicts is offloaded to dedicated memory dependence nodes which record the original non-speculative values of memory locations.

All hardware TLS schemes must deal with the situation where a speculative buffer fills up or where a speculative cache line gets evicted. HYDRA deals with evictions by stalling the thread until it is the oldest (non-speculative) thread and then continues. It is noted that such evictions can largely be mitigated by the addition of a victim cache [62] which records the address of the evicted line and the speculative read bits. Similarly,

[2]STAMPede speculation is referred to as Thread-Level Data Speculation (TLDS) in some literature.

when the write buffer fills, the thread is stalled until it is the oldest. The writes (now non-speculative) are then flushed and execution continues. STAMPede deals with cache evictions by aborting the thread and restarting.

With the emergence of hardware TM in recent commercial processor architectures there has been considerable interest in seeing hardware-supported TLS on real chips. Odaira et al. [63] implement TLS using Intel TSX [27] and evaluate its performance. A major barrier to implementation is the lack of support for ordered transactions. Intel TSX does not provide any mechanism for communicating between transactions without causing rollbacks so it is not possible to stall a transaction while it waits for previous transactions to commit. Odaira achieves a functionally correct implementation by creating a global transaction counter in software. When a transaction attempts to commit it checks the global counter. If the counter does not match its own transaction number then it must abort. This is likely to result in a high rate of rollback but is currently the only way to implement TLS on Intel TSX. Odaira shows that other advanced hardware supports, such as data forwarding and word-based conflict detection, are needed for effective TLS and finds that the maximum achievable speedup on Intel TSX is 11%. Other processors such as IBM Blue Gene/Q [30] and POWER8 [31] support ordered transactions but do not offer these more advanced features.

2.3 Dependence analysis

An important element in a system for effectively extracting parallelism is dependence analysis. Broadly speaking, dependence analysis is considered to include any techniques which attempt to predict or detect the aliasing of memory accesses during the execution of a program. The techniques I study can be divided into two categories. Firstly, compile-time dependence analysis schemes attempt to predict during compilation the instructions which may alias at runtime. Knowledge of the input set and dynamic runtime behaviour is not available, so these techniques usually overestimate the extent of dependences. Much of the difficulty in analysing programs written in C-like languages is the challenge of determining pointer aliasing and, as a result, much of the recent work on improving compile-time analyses has been on pointer analysis [64]. The analysis used as part of the HELIX parallelising compiler is flow-sensitive and context-sensitive [65] and is applied to the whole program. As was previously observed in section 2.1.1.4, interprocedural analysis is imperative for parallelising large loops.

Secondly, there are dependence profiling techniques which track memory references during an actual execution of the program. From recording the real runtime behaviour of the program it is possible to determine exactly which instructions alias. This technique provides more accurate results than the first but is limited in its usefulness since the results only apply to a particular program input. Moreover, the amount of data which is collected is typically extremely large for non-trivial programs, so processing the data efficiently can be a significant challenge. This approach to dependence analysis has been of particular interest to the designers of parallelising compilers and speculation systems. The work in this dissertation is primarily based on runtime profiling of dependences and therefore the discussion will be confined primarily to such techniques.

One of the first attempts to analyse available loop parallelism by tracking runtime dependences is the *parallelism analyser* [66]. This system takes a trace of a sequential program as input and attempts to determine the possible speedup of a parallelised ver-

sion. To detect dependences the analyser records every address written by an iteration and then for each read in subsequent iterations it searches for matches in the record. This technique has a severe scalability problem as the computation overhead increases with the square of the iteration's memory footprint. A similar approach is used by Tournavitis et al. [67] to construct a control and data flow graph (CDFG) which is used as input to an automatic paralleliser. Alchemist [68] is a profiler implemented on top of Valgrind, giving it the advantage of being able to analyse existing binaries. This system can distinguish between intra-iteration, inter-iteration and inter-invocation dependences in loops and outputs information to help the programmer to manually parallelise a loop.

A common challenge for profiling infrastructures such as these is that the algorithms do not scale well for large programs due to memory and computation overheads. A recent proposal for a scalable dependence profiler is SD3 [69]. Rather than recording each address as is done in all the previous proposals, SD3 takes advantage of the regular patterns commonly exhibited by the addresses touched by a particular instruction. This insight is used to record a compressed trace of each instruction's addresses. The dependence profiling described in chapter 4 is based on SD3 and a comprehensive overview of the technique as I have used it is given in that chapter.

2.4 Discovering the limits of parallelism

A common theme amongst researchers in computer architecture who have the goal of maximising performance is to discover the theoretical limits of a particular style of optimisation. This is a worthwhile endeavour because it allows us to understand the fundamental power of a particular idea and to gain insight into the practical limitations it faces. The work presented in chapters 4 and 5 are both directed at discovering the ultimate potential of different models of execution and is inspired by previous work in this vein.

Wall [70] examines the extent of instruction-level parallelism (ILP) that can be extracted by a superscalar processor by collecting a trace of the benchmark and scheduling instructions as early as possible. In the spirit of finding the limits of parallelism, independent of the constraints of circuitry, infinite functional units and register file ports are assumed. Wall finds that, even under these assumptions, the median ILP is only around 5. Austin and Sohi [71] use dynamic dependence graphs to show that much more parallelism can be extracted than indicated by Wall. The dynamic dependence graphs are used to detect output (storage) dependences through memory and the work assumes perfect memory renaming to remove them. In addition, whereas Wall reports average ILP by calculating the total number of dynamic instructions and dividing by the length of the critical path, Austin and Sohi show that ILP is bursty and that programs have phases of very high and very low parallelism. One shortcoming of this study is a failure to account for control dependences: instructions following a branch cannot be scheduled concurrently with instructions before the branch without speculation. Mak and Mycroft [72] show control dependences significantly reduce available ILP, although branch prediction and speculation may alleviate some of this loss.

While these studies look at the limits of ILP, Larus [66] describes an execution model to find the limits of loop-level parallelism when exploited in the style of DOACROSS. This work traces a sequential execution of a program and models parallel execution by simulating a machine where all iterations start at the same time. The model tracks inter-iteration dependences and delays the execution of instructions to resolve such dependences. Larus

also discusses the effect of DOACROSS-style optimisations to reduce the number of dependences and quantifies the parallelism which depends on such optimisations.

Recent work in this style has evaluated the limits of modern execution paradigms such as thread-level speculation. Ioannou et al. [73] find the limits of such techniques in the presence of various architectural features which support speculation. These features include out-of-order thread spawning, multiversioned caching, dependence synchronisation, partial rollback and value prediction. The authors find that, while out-of-order thread spawning did not influence the results, dependence synchronisation and value prediction improved results significantly, suggesting that these features will be important in realising the potential of speculation. Elder von Koch et al. [74] do a similar study of the limits of dynamic binary parallelisation with speculation and evaluate results for various overhead costs. They find that speedups are possible on an idealistic, zero-overhead machine, but that these benefits largely disappear when realistic overhead costs are simulated.

2.5 Putting this work in context

This dissertation presents three related, but distinct, pieces of work: a method for studying the limits of non-speculative parallelisation in chapter 4, a method for studying the limits of speculative parallelisation in chapter 5 and practical implementations for approaching these limits in chapter 6. In this section I will compare these contributions to the literature already presented in this chapter to give some perspective on the broader context where my work belongs.

My approach to finding the oracle DDG in chapter 4 is related to Alchemist [68] which also profiles the program to find dependence pairs and can distinguish between inter-iteration and inter-invocation dependences in loops. However, the output of this is used as feedback to the programmer whereas my technique feeds the output directly into the parallelising compiler. Previous work studying the accuracy of pointer analysis has also used the notion of generated code performance as a metric to quantify the accuracy of the static analysis [75, 76, 77]. To my knowledge, however, my work is the first to apply a profile-enhanced data dependence graph to a purely static parallelisation scheme and use parallel performance as a metric to measure the quality of the static analysis. While Tournavitis et al. [67] also use profiling to enhance the static dependence graph, they rely on the user to confirm the validity of the profile-detected parallelism whereas I use a completely automatic process to find the upper limit of parallelism which the compiler could, in theory, exploit.

The ideal dataflow model presented in chapter 5 is quite similar to the execution model used by Larus [66]. Larus instruments the program at the machine code level which leads to extra work being done to remove spurious dependences such as those caused by stack reuse. I instrument the program at IR level which gives greater control over which accesses are tracked. In addition, Larus's results are not based on the output of a parallelising compiler, whereas by using the code generated by HELIX I was able to obtain more accurate results which take advantage of realistic parallelising optimisations.

The analysis of the upper bounds for practical speculation techniques in chapter 6 has a lot in common with the work of Ioannou et al. [73]. These authors have also explored the limits of speculation for a range of possible architectural configurations. While they have looked at how various novel architectural supports affect performance, I have focussed on how the transactional memory implementation affects performance since this is the most

likely way speculation will be supported in future commercial processors.

The practical methods for extracting speculative loop-level parallelism described in chapter 6 build on a large body of previous work which was summarised in sections 2.1 and 2.2.2. There are many ways to classify the different proposals on this topic, and none are perfect, but a broadly applicable taxonomy can be found by dividing the work into non-speculative/speculative categories and further into IMT/CMT/PMT parallelisation styles. Table 2.1 shows how some of the previously discussed work fits into these categories.

IMT styles are all unable to tolerate any dependences occurring between iterations. Non-speculative entries in this space must prove that iterations are completely independent at compile-time. Speculative entries increase coverage to difficult-to-analyse loops, although any runtime dependences will cause failure. Some authors may apply an additional restriction for classifying a technique as IMT, that the iterations can also be run in any order. This would place *HELIX Pure Speculation* outside the IMT classification since iterations are restricted to commit in order. By contrast, the LRPD test runs the iterations completely independently and performs checks on loop completion to see if speculation was successful. I consider *HELIX Pure Speculation* to fit more comfortably into the IMT category, however, since it has no facility for tolerating loop-carried dependences. Johnson et al. [55] note that Privateer could be applied to other forms of parallelisation with loop-carried dependences, but the cited paper only discusses IMT so it is categorised as such.

CMT styles add some generality to IMT by providing mechanisms to synchronise loop-carried dependences while still exploiting parallelism. HYDRA and STAMPede are placed in the CMT category rather IMT as they support the synchronisation of variables which cause frequent dependence violations even within the context of speculative execution. In STAMPede this support is only possible for scalars which are shown by the compiler to cause dependences and a copy of the variable is allocated to prevent corruption of the actual variable in the case of rollback. Since *HELIX Judicious Speculation* alternates between running speculatively and non-speculatively within a single iteration, synchronised variables can be updated safely in place, so no extra copies or checks are needed. In addition, this supports the synchronisation of all memory references, including those to arrays or heap-allocated data. To my knowledge this is the first implementation of CMT-style speculation which can alternate between purely synchronised and purely speculative execution within a single iteration of a loop.

The PMT category includes techniques where a loop iteration is pipelined across several execution units. DSWP is the state-of-the-art in this approach and SpecDSWP extends the basic technique to support speculative execution. This increases coverage to loops with infrequent dependences and allows better balancing of pipeline stages. PMT-style parallelisation is not considered as part of the main work in this dissertation.

	Non-speculative	**Speculative**
Independent multithreading	• SUIF [12] • Polaris [14]	• LRPD Test [46] • Privateer [55] • *HELIX Pure Speculation (Section 6.3)*
Cyclic multithreading	• DOACROSS [1] • HELIX (Chapter 3) • *HELIX + oracle DDG (Chapter 4)*	• HYDRA [56] • Stampede [59] • Copy or Discard [52] • *HELIX Judicious Speculation (Section 6.4)*
Pipelined multithreading	• DOPIPE [16] • DSWP [17]	• SpecDSWP [22]

Table 2.1: Taxonomy of automatic parallelisation styles. Contributions of this dissertation shown in italics.

Chapter 3

HELIX automatic parallelisation

The analyses throughout this dissertation are based on the HELIX model of automatic parallelisation [3]. HELIX is a fully automatic parallelising compiler which parallelises loops in sequential programs by creating a thread for each iteration of the loop and running these concurrently. HELIX has previously demonstrated significant speedups for a number of general purpose sequential programs, traditionally considered a challenge for automatic parallelisation [2].

A major challenge commonly faced when implementing this style of automatic parallelisation is the synchronisation of loop-carried dependences between threads. The compiler must be able to accurately identify such dependences. HELIX includes a state-of-the-art interprocedural dependence analysis [65] to detect potential conflicts between threads. The compiler produces "sequential segments" to sequentialise code which may cause inter-thread conflicts. In addition, HELIX performs a range of other analyses to reduce the number of synchronisation points in a thread and to maximise the chances of achieving speedups.

This chapter will look at the HELIX model and the compiler analyses performed in detail. Section 3.1 describes *ILDJIT*, the optimising compiler in which HELIX is built. Next I will walk through the HELIX algorithm in section 3.2 to fully explain the nature of the technique. Some previously published advances in reducing the cost of inter-core communication are discussed in section 3.3. Section 3.4 describes the benchmark set used in all my experiments. Finally I will look at the HELIX timing model which is used to estimate the performance achieved by parallelisation in a deterministic manner (section 3.5). This chapter is exclusively prior work with the exception of my validation of the timing model in section 3.5.2.

3.1 ILDJIT

HELIX is implemented in the ILDJIT compiler framework [78]. ILDJIT is a flexible and extensible compiler which easily allows the addition of modules to implement new optimisations and passes. ILDJIT uses a low-level intermediate representation (IR) and provides substantial support for instrumenting code at the IR level. This was convenient for implementing support for the various runtime systems discussed in later chapters.

The compiler has a Common Intermediate Language (CIL) frontend. To compile C programs using ILDJIT, the C code is first converted to CIL using the GCC4CLI [79] backend for GCC. ILDJIT has previously been shown to produce code of comparable performance to GCC's optimised code [78]

Listing 3.1: A sample loop suitable for HELIX paralleisation.

```
while(nodeA != endA && nodeB != endB){
    nodeA = nodeA->next; // A
    process(nodeA);      // B
    nodeB = nodeB->next; // C
    process(nodeB);      // D
}
```

3.2 HELIX

HELIX parallelises a sequential loop by assigning successive iterations of the loop to different cores. At runtime, the iterations are run concurrently to the greatest degree permitted by the dependences within the loop. The cores conceptually form a ring such that iterations are assigned to the cores in a cyclic fashion. In this way the operation of the runtime is simplified by ensuring that communication between successive iterations is predetermined, i.e. core 1 always sends data to core 2, core 2 to core 3 and so on.

The compiler recognises loop-carried dependences and inserts synchronisation code to ensure that such dependences are not violated. HELIX can insert many such instances of synchronisation code so that different dependence cycles can be sequentialised independently. In this manner, HELIX can take advantage of parallelism even in the presence of many dependences by overlapping the code of different dependence cycles in different iterations. HELIX can only execute a single parallelised loop at a time so, for instance, if we have nested loops, it is not possible to run multiple levels of the nest in parallel concurrently. Therefore, the appropriate selection of loops to run in parallel may have a significant impact on the performance of the program since, in a given loop nest, the outer loop may provide less speedup than an inner loop.

To illustrate the nature of HELIX parallelisation, consider the code in listing 3.1. This loop processes two dynamic linear data structures which are iterated across with pointers `nodeA` and `nodeB`. On each iteration of the loop we need to find the next node in each of the data structures and perform some processing on that node. We assume for this example that the processing of each node is independent of each other node.

Following a compile-time data dependence analysis, HELIX produces a data dependence graph (DDG) as shown in figure 3.1 with loop-carried dependences indicated in green. As we can see there are two loop-carried dependences which need to be synchronised in this loop. Therefore HELIX will create two sequential segments to sequentialise the execution of `A` and `C` across the different iterations.

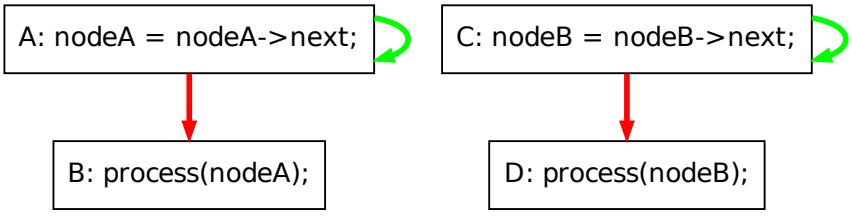

Figure 3.1: Data dependence graph for loop in listing 3.1. Green arrows indicate loop-carried dependences, red arrows indicate intra-iteration dependences.

Figure 3.2 demonstrates the manner in which the HELIX-parallelised loop would execute. The sequential segments are indicated as yellow-shaded blocks. The time taken

to execute `B` and `D` varies depending on the input data. Each sequential segment must execute in sequential order across cores, for example, sequential segment `A` is never overlapping in different cores. Similarly, core 1 must stall while it waits for sequential segment `C` to complete in core 0. However, different sequential segments can overlap with each other, for example, `C` on core 0 executes concurrently with `A` on core 3. The parallel portions of code can execute concurrently with any other code on the other cores.

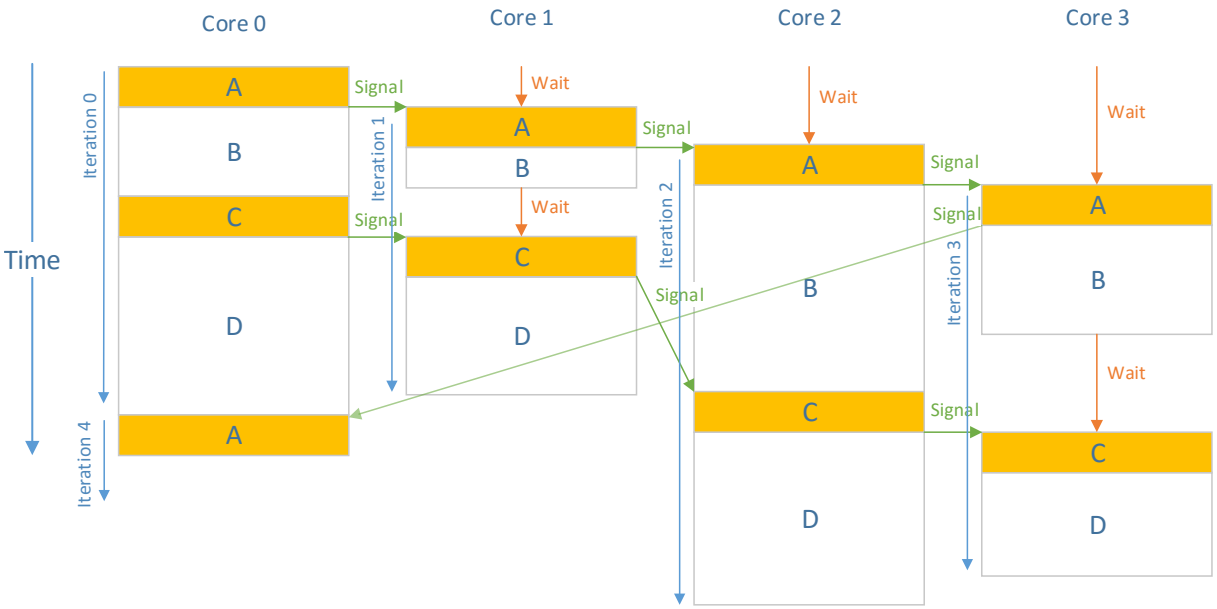

Figure 3.2: Execution schedule for loop in listing 3.1.

HELIX transformations operate on the intermediate representation (IR) of the loop. The remainder of this section looks at the specific steps of the parallelisation pass in more detail.

3.2.1 Loop normalisation

As an initial step towards parallelisation, the loop must be transformed into a particular format which the compiler can work on more easily. All HELIX parallelised loops must consist of a *prologue* and a *body*. The control flow graph for the example in listing 3.1 is shown in figure 3.3. The prologue is the smallest set of instructions which must be executed to determine whether the body of that iteration should be executed. This consists of those instructions which are not post-dominated by the loop's header.

So in this example, the tests `nodeA != endA` and `nodeB != endB` are not post-dominated by the header, so this code constitutes the prologue. The prologues of each iteration must run sequentially since until the prologue completes it is not known whether or not the succeeding prologue should execute. Therefore the prologue can be thought of as a sequential segment similar to other sequential segments throughout the body, and many of the models in this dissertation treat the prologue as such.

Once a loop exit condition is reached in any given prologue, it sets a flag to be read by the successive thread to indicate that it may exit when it attempts to start its next prologue. This thread sets the corresponding flag for the next thread and so on until all threads have exited the loop.

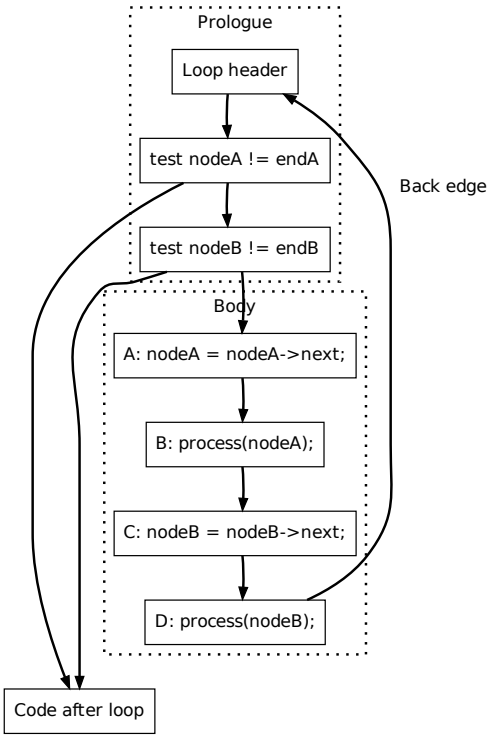

Figure 3.3: Control flow graph for loop in listing 3.1 following normalisation.

The body of the loop simply consists of all instructions which are not in the prologue. Each time the body of the loop executes, this indicates that the prologue of the next iteration should execute. Therefore code is added to the start of the body to set a flag indicating to the next thread that it may proceed with the next prologue.

3.2.2 Dependence analysis

Having normalised the loop, HELIX uses a state-of-the-art interprocedural dependence analysis [65] to detect all dependences in the loop, including those caused by functions called from within the loop. This analysis was chosen because it operates at a low-level and does not require detailed semantic information from the source code, which suited the low-level ILDJIT IR. In addition, interprocedural analysis is generally a necessity to parallelise the biggest loops in a program. The analysis was enhanced beyond what was described by Guo et al. [65] to provide more detailed information to help facilitate parallelisation. In particular, semantic information about C library calls was added which greatly reduced the number of dependences caused by such calls.

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 we do not need to consider write-after-write and write-after-read dependences which occur through registers or the stack. Read-after-write loop-carried dependences which occur through the stack or registers must still be respected and these will be converted into accesses to shared memory as described in section 3.2.4.

In general, a loop is likely to contain many loop-carried dependences as a result of induction variables which are updated on each iteration of the loop. However, in most cases it is possible to compute induction variables locally in each thread from the iteration number. Therefore it is possible to eliminate dependences caused by these variables. In

addition, dependences caused by accumulators (such as in a loop to sum the values in an array) can be removed by creating a local accumulator in each thread and summing together these local accumulators at the end of the loop.

3.2.3 Sequential segments

Now that HELIX has normalised the loop and detected the loop-carried dependences which must be satisfied, it can proceed to create the sequential segments which will ensure that these dependences are not violated. A data dependence consists of two instructions which may access the same memory address. A sequential segment is created for each such data dependence and guards are inserted at the beginning and end of each segment to ensure sequential execution. At the beginning of the sequential segment a `Wait` operation is inserted which stalls the thread until the corresponding sequential segment in the preceding iteration has completed. At the end of the sequential segment a `Signal` operation is inserted which indicates to the `Wait` of the corresponding sequential segment in the succeeding thread that it is now safe to continue execution. For each sequential segment, the `Wait` and `Signal` operations create a chain of sequential execution across the cores which ensures that any code involved in loop-carried dependences executes in loop iteration order. `Wait` and `Signal` may be implemented by simple flags with one flag per sequential segment per thread. `Signal` sets the appropriate flag of the succeeding thread and `Wait` waits for its own flag to be set, resets it and continues.

HELIX also performs optimisations to reduce the amount of code that must be executed sequentially. Instructions which are contained within a sequential segment but which are not in any way involved in loop-carried data dependences may be moved to after the end of the sequential segment. Functions called from within the loop which cause data dependences may also be inlined to allow more fine-grained scheduling of the code to reduce the size of sequential segments.

3.2.4 Communicating between threads

To permit threads to communicate signals to each other, a memory buffer is allocated for each thread. When the loop starts, each thread is initialised with a reference to its own buffer and the buffer of the succeeding thread. The last thread is given a reference to the buffer of the first thread, thus creating a ring. Each thread writes into the buffer of the succeeding thread and reads from its own buffer.

The number of sequential segments is known at compile-time and thus the memory required for the buffers can be allocated before execution of the loop begins. The locations of the flags which guard each sequential segment within the buffers are also statically determined.

In addition to signals for sequential segments, the threads must be able to communicate variables used in the loop which belong to the stack frame of the main sequential thread. These variables are referred to as *live-in* or *live-out* variables depending on whether they were live at the entrance to or the exit from the loop. In addition, the loop may contain live-in variables which are allocated in the stack frame of the main thread and are written in one iteration of the loop and read in a later iteration. Such variables are kept in the stack frame of the main thread, and loads and stores are inserted into the code of the parallel threads to access the originally allocated location. Data dependences caused by these variables are preserved with sequential segments in the usual manner.

3.2.5 Loop selection

Applying HELIX parallelisation is not profitable for all loops. Code has been added to the loop to execute the `Signal` and `Wait` operations and this adds overhead to the execution of the parallelised loop. In addition, transferring the signals from one core to another will usually involve communication through a shared L2 or L3 cache which typically incurs a large latency. For some loops where the parallel sections are small relative to the time spent communicating, the parallelised version of the loop may not speed up or may even become slower than the original sequential version.

In addition, the HELIX model only allows for a single parallelised loop to be executing at a time. If there are nested loops and a parallel version of the outer loop begins executing, the inner loops will be forced to run sequentially, since HELIX is only capable of executing a single parallelised loop at a time. If this outer loop offers less parallel speedup than some combination of the inner loops, optimal performance will not be achieved. For these reasons, it is critical that loops are selected which will be most profitable for parallelisation.

To find an optimal solution we start by building a loop nesting graph which covers all the loops in the entire program. This is a directed graph where each node represents a loop and each edge represents a nesting relationship (i.e. the loop at the tail of the edge is nested within the loop at the head). A nesting relationship can transcend function boundaries and a loop *L2* which is in a function called from within loop *L1* is considered to be nested within *L1*. The graph is not necessarily a tree (or forest) since a function which contains a loop may be called from within multiple other loops, resulting in multiple parents.

To estimate the speedup afforded by parallelising a particular loop, it is useful to do a profiling run of the loop with the HELIX timing model since this gives deterministic results and is machine-independent. Each loop is executed sequentially with callbacks to a performance model which estimates the amount of time the various parallel threads would have spent stalling and communicating data in a parallel execution. The output of the model is a figure for how much time would be saved by executing a parallel version of the loop. The timing model is discussed in more detail in section 3.5.

Two values are assigned to each node in the loop nesting graph: t which is the time saved by running the parallel version of this loop as opposed to the sequential version, and $maxT$ which is the maximum time saved by either parallelising this loop or any selection of its subloops. The algorithm proceeds in two stages. First it works from the leaves (the nodes with no children) upwards. At each node set $maxT$ to either t at the current node or the sum of its children's $maxT$s, whichever is greater. This is repeated at each of the node's parents.

Now each node contains a value for the maximum speedup achievable at each loop or a combination or its descendants. If the maximum time saved at a node is equal to the time saved by that node, then choosing that node is the optimal solution for the node and all its descendants. So the second stage of the algorithm works from the roots (nodes with no parents) downwards. If t is equal to $maxT$ then select this loop for parallelisation and proceed no further into its descendants. If not then repeat this at each of the node's children.

Figure 3.4 shows a sample loop nesting graph. The graph is not a tree since loop5 is nested within both loop1 and loop2 due to function func4 being called by both func1 and func2. In this example, parallelising loop1 would save 100 cycles but parallelising the best selection of its subloops would save 4000 cycles. The loops which are ultimately selected are shaded green.

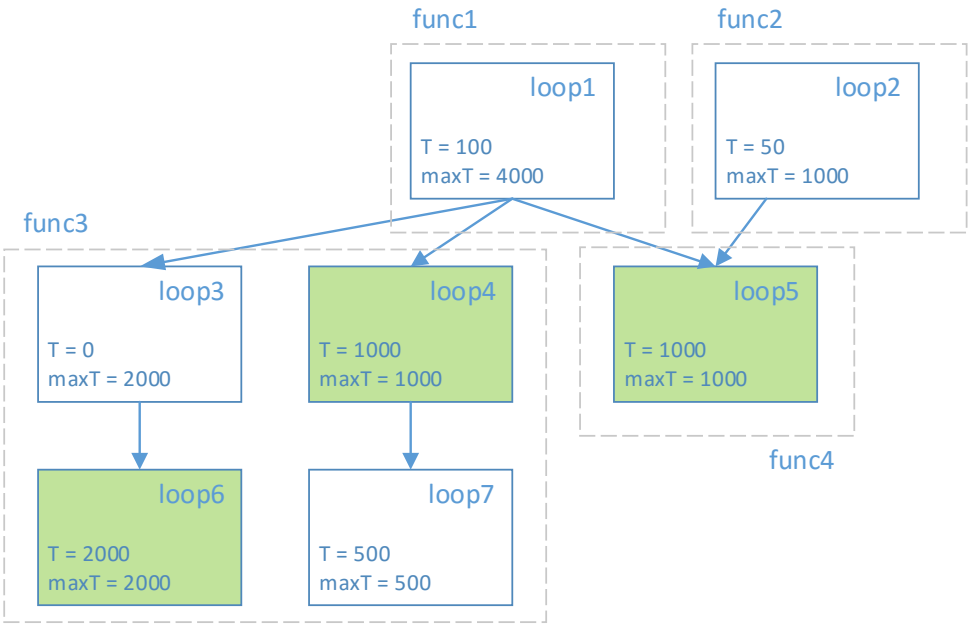

Figure 3.4: Sample loop nesting graph with selected loops shaded green.

The algorithm as described may not choose the best loops for certain graphs where a node has multiple parents, since the time saved by parallelising a subloop should be distributed between its parents according to the proportion of relative frequencies with which the loop is invoked by each path. For example, in figure 3.4 if loop 5 is called 999 times from loop 1 but only once from loop 2 then the the $maxT$ for loop 2 will be incorrect. This can be solved by enhancing the profiling to calculate the time saved by a loop along each possible path in the graph and propagating $maxT$ upwards in appropriate proportions. HELIX performs aggressive inlining so that multiple copies of functions may exist when they are called from different sites. This means that the problem of multiple predecessors in the loop graph occurs fairly rarely. For the benchmarks I have studied the basic algorithm was found to be sufficient to choose the best loops.

3.3 Optimising inter-core communication

A common challenge with this form of parallelisation is the overhead of communication between cores required to implement synchronisation [80]. In HELIX, synchronising sequential segments is on the critical path and minimising the latency of signals is crucial to enabling speedups. In prior work on HELIX, two techniques have been proposed to reduce the expense of these operations: helper threads and the ring cache.

Helper threads

`Wait` and `Signal` operations are implemented by creating a flag in shared memory which is set by the signalling thread and inspected by the waiting thread. Communication of the signal relies on the cache coherence protocol making the signal visible when the waiting thread attempts to read it. Therefore, communication only takes place once the thread begins to wait, so it will always incur the latency of going through the cache coherence protocol, even if the signalling thread "sent" the signal many cycles previously.

To reduce this latency on a processor with simultaneous multithreading (SMT), HELIX may couple each iteration thread with a helper thread [2]. This is a thread which runs on the same core as the iteration thread but whose sole task is to prefetch the signals from the previous core. Conceptually, the signalling thread can now "push" the signal into the private cache of the waiting thread, rather than having the waiting thread "pull" it in when it is required. The latency of waiting for a signal then reduces to almost zero in cases where the signal was sent much earlier since the `Wait` operation will simply consist of loading a value from the core's own private cache.

Ring cache

Helper threads are useful in the cases where signals occur significantly before the corresponding waits, but when a wait occurs before the signal the thread suffers the full latency of communicating through the cache coherence protocol. Unfortunately most commercial processors do not provide any other mechanism for low latency inter-core communication. However, it has been shown that low latency communication on a chip multiprocessor (CMP) can be supported with modest architectural extensions. Campanoni et al. [4] propose the addition of a *ring cache* to an Atom-like CMP: a ring network with one node attached to each core. Each ring node contains a set associative cache, 1KB in size with word-sized cache lines. The instruction set is extended with two instructions to delimit the boundaries of sequential segments: `wait` and `signal`.

When the core is executing a sequential segment (after a `wait` but before a `signal`) all memory accesses are directed first to the core's ring node. The address and value are stored in the node's cache and then propagated to all other nodes in the ring. In this manner, all memory accesses which could be involved in data sharing can forward their data through the ring cache at low latency to whatever core the data is shared with. The network topology takes advantage of the nature of CMT parallelisation: data produced by one iteration will usually be consumed by the subsequent (or nearly subsequent) iteration and the iterations themselves are distributed across cores in a logical ring. Reducing the overhead of communication not only improves the performance of loops which were already parallelisable but also enables the profitable parallelisation of small hot loops which would previously have been overwhelmed by the communication overheads. An evaluation of the ring cache on a 16-core architectural simulator found that it increased the geometric mean speedup of SPECINT with HELIX from 2.2x to 6.85x [4].

3.4 Benchmarks

To test the performance of HELIX and the various modifications discussed throughout this dissertation I have used the *cbench* benchmark suite [81]. This is a set of sequential benchmarks based on the MiBench suite [82] but with the addition of multiple datasets. The suite includes applications from various domains such as security, telecommunications and office software. The benchmarks used and their description are shown in Table 3.1. This table includes all of the benchmarks which I was able to compile with ILDJIT and parallelise with HELIX. There are a number of other benchmarks in the suite which have not been included because I was unable to push them through the framework. Many of these benchmarks could not be compiled from C to CIL (required by the ILDJIT frontend) because of the lack of a complete C-library implementation in the GCC4CLI

Benchmark	Lines	Loops	Description
automotive_bitcount	460	4	Counts number of bits set in a word.
automtive_susan_c	1376	4	Finds corners in an image.
automotive_susan_e	1376	6	Finds edges in an image.
automotive_susan_s	1376	4	Gaussian smoothing on an image.
security_sha	197	2	Calculates SHA hash of a file.
security_rijndael_d	952	2	Rijndael decryption.
security_rijndael_e	952	2	Rijndael encryption.
office_stringsearch1	338	2	Searches text for string matches.
consumer_jpeg_c	14014	20	Compresses an image with JPEG.
consumer_jpeg_d	13501	17	Decodes a JPEG compressed image.

Table 3.1: Benchmark descriptions.

backend. The GCC4CLI project has not been maintained for a number of years and this caused many problems with compiling benchmarks. In addition, some benchmarks are not included because ILDJIT could not compile them or did not generate correct code. Of the benchmarks in cbench which are not listed in 3.1, 14 could not be compiled with GCC4CLI and 6 could not be compiled with ILDJIT or crashed during execution.

The SPECINT benchmark suites are commonly used for the type of work discussed in this dissertation since these benchmarks are large and are generally considered difficult to parallelise. However, the analyses and timing models implemented, in particular the oracle data dependence analysis described in chapter 4, incurred very large memory and computational overhead. The SPECINT benchmarks proved too difficult to analyse, with estimates indicating that some loops would take weeks to complete. The applications in cbench are smaller but still contain interesting, non-trivial kernels and as such this suite was considered a good fit for my work. It may be possible to refine these techniques to allow analysis of SPECINT by using more information from the compile-time dependence analysis to reduce the number of checks the profiler needs to perform.

3.5 HELIX timing model

While HELIX is capable of producing multi-threaded parallel code to be executed on a real machine, for most of the experiments in this dissertation I will be using the HELIX timing model to estimate speedups. The model allows quick exploration of the design space and the ability to experiment with new runtime systems and architectural parameters. To estimate execution time with the model, the HELIX parallelised code is instrumented with callbacks to a performance model which records the amount of time the various operations would have taken in a real execution.

To run the timing model, the HELIX transformations are performed to produce a parallelised version of the loop as normal. However, rather than running the code in parallel, the loop code is instrumented at the IR level with callbacks to an analytical performance model. The following callbacks are inserted:

- A callback is inserted after each basic block to record the number of cycles taken to execute the basic block. The number of cycles is determined statically based on the type and number of instructions contained.

- Callbacks are inserted at the start and end of each sequential segment. This allows the model to determine how much time each thread would spend stalling while waiting for signals.

- Each instruction which accesses shared memory is instrumented. This will allow the system to track runtime data dependences in the models proposed in chapters 5 and 6.

- The entrances and exits to loops and the starting point of each iteration are also instrumented.

When the code is executed, rather than creating multiple threads and assigning successive iterations to successive threads, only a single thread is created and all iterations run on a single core. The timing model simulates a multi-core system with a cycle counter for each core. Each time the loop makes a callback to the model, the cycle counter for one of the cores is incremented to reflect the time spent performing whatever operation was indicated by the callback. Initially the cycle counter for core 0 is incremented and the model moves to the next core each time a new loop iteration begins. Note that because the HELIX paralleliser runs as normal, the code being executed is an accurate representation of the code that would be executed by any individual thread in parallel execution.

The number of cycles attributed to each instruction was determined by tuning the model to match the execution time of an Intel Atom-based architectural simulator [83]. The tuning was performed on the SPECINT2000 benchmark suite. The model was tuned to fit 70% of the benchmarks and then tested on the remaining 30% to prevent overfitting. The average error was found to be within 6%. Although the overall cycle count would not be the same when comparing the timing model to other architectures, the speedups obtained from parallelisation should be stable since these are always compared to a baseline from the same architecture or model. An attempt was made to include a simulation of the memory hierarchy in the timing model but it was found that this increased the execution time of the model to unacceptable levels.

Figure 3.5 shows a sample execution and how the cycles elapsed are assigned to the cycle counters of the virtual cores. When the execution enters a sequential segment, a delay is applied to the appropriate core based on the actual running time of the previous invocation of that sequential segment. The model also applies delays to model the latency of communicating between cores and executing the code for waits and signals.

I have opted to base most of my experiments on the timing model since it gives deterministic results, is more amenable to performance debugging and allows experimentation with a variety of runtime systems where providing full implementations would not be feasible or sensible before their worth has been proven. In addition, it allows tweaking of the architectural parameters, such as the time it takes to communicate values between cores. This ability is convenient for determining what the effects of communication latency are and for simulating different architectures. A validation of the model in section 3.5.2 confirms that the timing model can predict speedups on a real machine for the benchmarks I am studying.

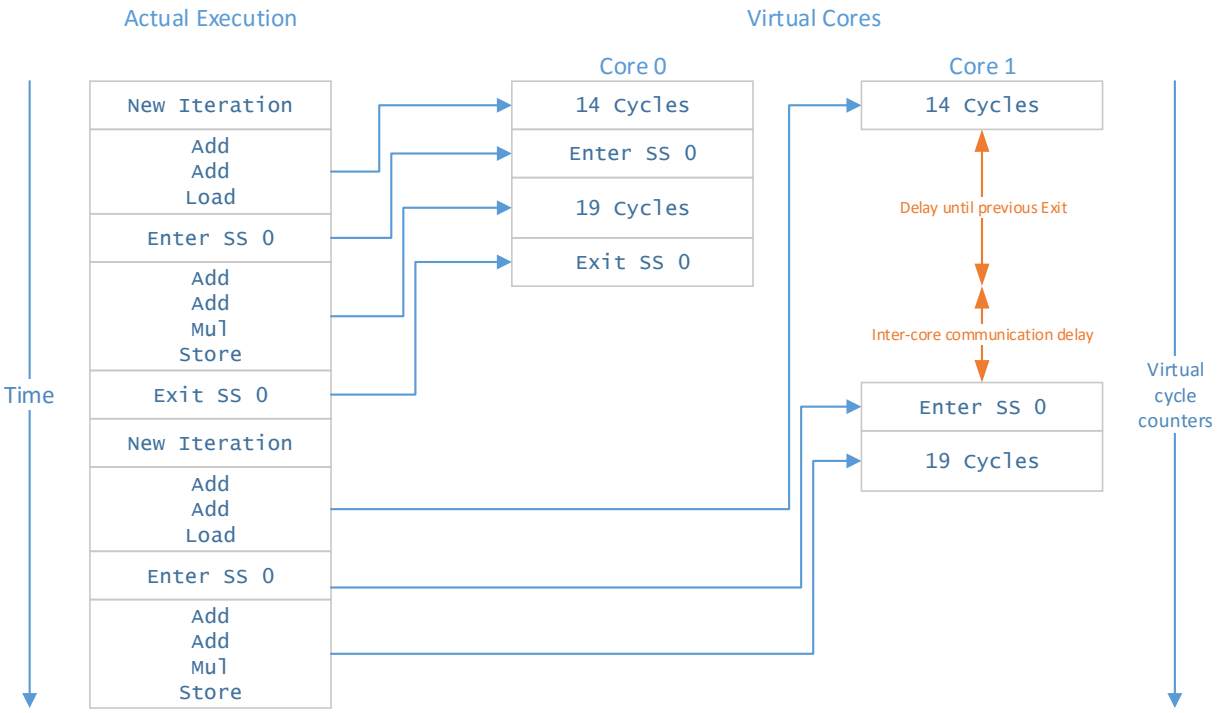

Figure 3.5: Sample execution of the HELIX timing model.

3.5.1 Assumptions

In the implementation of the HELIX timing model the following assumptions are made:

- That the time taken to execute any particular type of instruction is constant and can be determined statically.
- That the working set for the selected benchmarks largely fits within the private caches of the cores such that memory instructions have constant latency.
- That the time to transfer values between private caches (e.g. to communicate signals) is constant.
- False sharing effects between private caches which could reduce performance are not modelled.

In practice, a real machine will violate these assumptions. We are faced with a trade-off where, the more accurate the simulation is, the more time it takes to run, the more difficult it is to implement various execution models and the more tied the results will be to a particular architectural design. I opted to base my results on the HELIX timing model since it allows exploration of a large design space and the results provide a more accurate upper bound of the potential of the execution styles which were studied since they are considered in isolation from specific architectural decisions.

3.5.2 Validation of the timing model

While the timing model is convenient for experimenting with different parameters and for producing deterministic results, it is imperative to ensure that the trends seen in the

	Haswell	Atom
Processor	Xeon(R) E3-1230 v3	Atom(TM) Z3735F
Frequency	3.3GHz	1.33GHz
Cores	4 (hyper-threaded)	4
Cache	32KB + 256KB private 8192KB shared	32KB private 1024KB x2 shared

Table 3.2: Machines used to measure HELIX performance.

timing model results could be accurately reproduced on a real machine. To establish confidence in the timing model I have parallelised a number of loops with HELIX and timed the execution on a real machine. These loops are listed in table 3.3 and are all high-coverage loops in cbench. The results are then compared to those produced by the timing model to ensure they are comparable. The most important metric is loop speedup, therefore this validation will compare the speedup trend rather than the wall time or raw clock cycles. The baseline is the sequential version of the compiled program produced by ILDJIT. To attain speedup results the loops are parallelised by HELIX and executed on various numbers of cores.

I calculated the speedups achievable by HELIX on two machines, an Intel Haswell server and an Intel Atom compute stick. The specifications of these machines are shown in table 3.2. The Haswell machine features hyper-threading technology which allows two threads to run concurrently on the same core, giving a total of 8 virtual processors. It was found that allowing the operating system to use its own heuristics to assign threads to virtual processors gave rise to variable results. This was because the operating system would sometimes assign two threads to the same core resulting in excessive demand on that core's resources. In practice, it was always advantageous to keep each of the HELIX generated threads on separate cores, so I manually assigned each thread to specific cores for all tests. In addition, both of the test machines use frequency scaling to achieve a particular balance of performance and power consumption so it was necessary to fix the processor frequency to its maximum value.

I selected several significant loops from the benchmark suite for which the timing model claimed that speedups were attainable. Figure 3.6 shows a comparison of the speedups claimed by the model and those achieved on real hardware. All loops constitute at least 50% of the total benchmark execution time and speedups are calculated just for the time spent running the loop, i.e. not the whole benchmark. Speedup is calculated relative to the sequential performance of the loop on each machine, e.g. Atom speedup is calculated as sequential execution time on the Atom divided by parallel execution time on the Atom. The highest level of cache on the Atom is the 2048KB L2 which is split into two caches, one shared between cores 0 and 1, the other shared between cores 2 and 3. As a result, communication from core 1 to core 2 must go through memory which is considerably more expensive. This results in poorer performance scaling on this machine when going beyond two cores.

The results shown for Atom and Haswell do not use either of the techniques for reducing the latency of inter-core communication described in section 3.2.4. This is because the helper thread implementation was not operational at the time of writing due a recent

Benchmark	Function	Coverage
automotive_susan_c	susan_corners	88.3%
automotive_susan_e	susan_edges	56.5%
automotive_susan_s	susan_smoothing	99.6%
automotive_bitcount	main	100%

Table 3.3: Loops used to validate the timing model.

change of the compiler backend to LLVM which caused some functionality to break. The ring cache could not be evaluated since it has only been implemented in simulation. This may reduce performance relative to the timing model which assumes that some mechanism is being used to "push" signals into the private cache of the waiting core. In particular, relatively small loops, like automotive_bitcount, may suffer noticeable performance loss since the overhead of communication is large relative to the loop.

The results show that the timing model is accurate at predicting speedups and that the performance trends are similar between the model and the real execution. The main source of inaccuracy in the model is that it does not model the memory hierarchy and so underestimates the cost of some long latency memory accesses. In addition, the expense of communication in the case of false sharing between the cores' private caches is not accurately modelled; however, there is considerable prior work on compiler optimisations to reduce the rate of false sharing in parallel programs [84] which could be employed to reduce this problem. The benchmarks studied in this dissertation have small working sets which fit inside the cache and so modelling memory accesses to have constant latency is considered reasonably accurate for these test cases.[1] This simplification of the model greatly reduced the complexity of experimenting with different models and made the model considerably faster to run, which was conducive to exploring a variety of novel runtime systems as discussed in chapters 5 and 6.

[1]The analysis of transaction sizes in section 6.5 confirms that these loops have sufficiently small working sets to fit inside the cache.

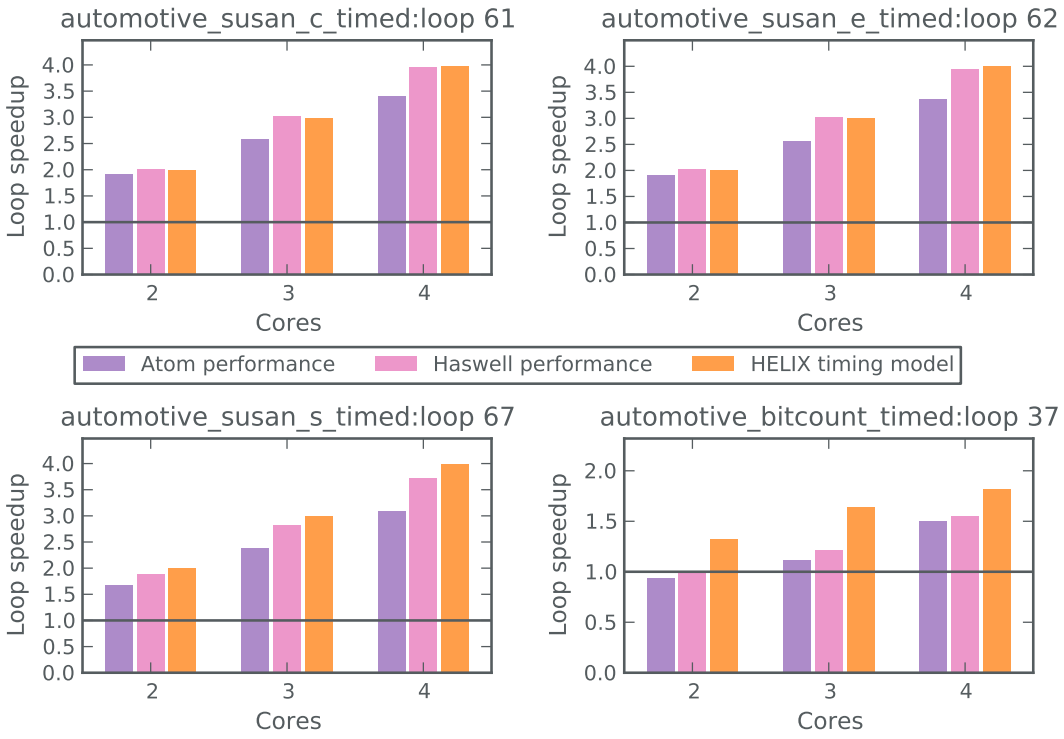

Figure 3.6: Comparison of speedups with the timing model and running on real hardware.

Chapter 4

Limits of static parallelisation

In the previous chapter I described the HELIX parallelising compiler and stated that it uses a state-of-the-art interprocedural dependence analysis. In general it has been assumed in the automatic parallelisation community that dependence analysis is a formidable challenge in achieving speedups with statically parallelised code. Therefore, the community has spent a significant amount of effort on improving this dependence analysis [64, 65, 85]. Previous work has shown empirically that an improved dependence analysis can enhance the performance of automatic parallelisation [86]. While these efforts were justified in the past, there is no evidence that this is still a limitation for today's compilers. What if dependence analysis is already good enough and further enhancements will result in only negligible performance improvements? In this chapter I will study the upper limits of compile-time analysis for HELIX style parallelisation and determine to what extent speedups are being limited by shortcomings in the analysis.

To find the limits of achievable speedups, I simulate a perfect dependence analysis which can know exactly which compiler-identified dependences will actually exist at runtime. To achieve this I carry out a profiling run of the program to detect any may-dependence pairs claimed by the analysis which are never actually realised at runtime. These are the only dependences that a hypothetical improved compile-time dependence analysis could eliminate. These apparent dependences can be removed from the compile-time determined data dependence graph (DDG) to create an oracle DDG. This is the upper limit of how accurately the compiler can identify dependences given a specific program input set. By parallelising the code again, replacing the compile-time DDG with the oracle DDG, it is possible to determine an upper bound on potential speedups for purely static parallelisation in HELIX.

This chapter begins with a description of how I generated an oracle DDG by collecting runtime memory traces and analysing them to find actual runtime data dependences (section 4.1). Section 4.2 presents results for how effective the oracle DDG analysis is at removing dependences from the static DDG and evaluates the performance of HELIX when parallelising with the oracle DDG. Finally in section 4.3, I analyse the results from section 4.2 and explain the outcomes with reference to specific benchmarks.

4.1 Oracle data dependence graph

The HELIX-generated DDG contains data dependences which do not manifest themselves at runtime. 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 extremely difficult for the compiler to determine exactly what memory locations can be touched by a particular instruction. In addition, not having access to the input data on which the program will be run can make it even more challenging for the compiler to predict the flow of data at runtime. It is a requirement that the compiler must, as a minimum, produce correct code so in these circumstances it must be conservative and assume that all possible data dependences do actually exist. It is assumed that this is a major source of inefficiency in automatically parallelised code, and indeed, Ottoni et al. [86] show empirically that removing spurious dependences from the DDG results in improved performance.

In this section I will describe a method for generating an *oracle DDG*, a refined version of the compiler's static DDG which only contains those dependences which actually exist at runtime for a particular program input set. First, traces are generated which precisely record all memory references at runtime. Then the traces are analysed offline and all memory references which did not exist at runtime are removed from the static DDG. I opted for an offline approach for two reasons. Firstly, keeping an entire trace of every memory instruction in the program in main memory at once would be infeasible; by analysing the trace offline I could reduce the memory overhead by dealing with individual instructions at a time. Secondly, by recording the trace once, I could then perform various analyses on the traces without having to expensively rerun the trace collector each time. The approach used here is similar in essence to that described by Tournativis et al. [67], however, I have used compression schemes which have been shown to reduce the memory overhead of the analysis by up to 20 times [69].

Generating the oracle DDG takes place in three stages:

1. **Memory trace**: Collect a trace of all addresses touched by each memory instruction in the program.

2. **Control flow trace**: Collect a trace of the order in which each memory instruction was executed to determine the direction of dependences.

3. **Analyse traces**: Analyse the traces to find all pairs of instructions which conflict.

Since the program is run multiple times it must be possible to identify the equivalent instructions from one run to the next. This is achieved by first compiling the code to the compiler's intermediate representation (IR), assigning each IR instruction a unique ID, and storing the IDs and associated IR on the disk. Then, on each run, it is possible to identify which runtime instructions correspond to each other. Each of these instructions is instrumented with a call back to the trace collector.

4.1.1 Memory trace

The memory access trace provides a complete record of every memory event which occurs during the execution of the loop. The trace is compressed using the SD3 scheme [69]. This takes advantage of the fact that memory instructions in a loop typically access memory in a stride fashion. For example, consider the loop in listing 4.1. If we imagine the array `B` begins at address `0x1000`, the reads from the array during one iteration of the inner loop will form this sequence of memory accesses:

```
0x1000 0x1004 0x1008 0x100C 0x1010 0x1014 0x1018 0x101C
```

Listing 4.1: Sample loop with corresponding IR code.

```
for ( int i = 0; i < 4; i++ )
    for ( int j = 0; j < 8; j++ )
        A[i] += B[j];

i0: r0 = read(B[j])
i1: r1 = read(A[i])
i2: r2 = r0 + r1
i3: write(A[i], r2)
```

This can be represented as a base address (`0x1000`), a stride (4) and a number of repetitions (8), together referred to as a *memory set entry*. A complete trace for one instruction consists of a sequence of memory set entries. If array `A` begins at address `0x2000`, then the complete memory trace for this program would look as follows:

```
i0: (0x1000, 4, 8), (0x1000, 4, 8), (0x1000, 4, 8), (0x1000, 4, 8)
i1: (0x2000, 0, 8), (0x2004, 0, 8), (0x2008, 0, 8), (0x200C, 0, 8)
i3: (0x2000, 0, 8), (0x2004, 0, 8), (0x2008, 0, 8), (0x200C, 0, 8)
```

Some instructions, such as those involved in pointer chasing in dynamic data structures, do not compress well with this scheme. These are compressed by taking advantage of the fact that such instructions tend to be confined to a relatively small portion of the address space. Each memory set entry records the base address as the offset from the previous base address rather than the absolute address.

4.1.2 Control flow trace

With the memory trace there is already enough information to determine which instructions touch the same addresses at runtime. However, there is no record of the ordering of dynamic instructions, and consequentially it is not possible to determine if the conflicting memory references occurred in different iterations of the loop. This is important because it is necessary to identify instruction pairs which cause loop-carried dependences. Therefore a *control flow trace* is recorded which stores the ordering of every dynamic instruction which accesses memory in the program.

This trace could become prohibitively large without an effective compression scheme. Two complimentary methods are used to compress the data to a manageable size: iteration-level compression and loop-level compression.

4.1.2.1 Iteration-level compression

Firstly a novel compression scheme is used which takes advantage of the nested patterns which exist in the instruction trace. For example a loop with a number of levels of inner nested loops might produce a raw instruction trace which looks like the following, where each number is the ID of a static instruction:

```
1 2 3 4 5 4 5 4 5 2 3 4 5 4 5 4 5 2 3 4 5 4 5 4 5 6 7
```

To record the nested patterns a new grammar is used which is expressed here in Extended Backus-Naur Form (EBNF):

```
digit               = "0" | "1" | "2" | "3" | "4"
                    | "5" | "6" | "7" | "8" | "9"
instruction-id      = digit{digit}
num-reps            = digit{digit}
compress-pattern    = "("compress-pattern","num-reps")"
                    | "("instruction-id")"
compressed-output   = {compress-pattern}
```

The example raw trace would be compressed to:

```
(1)((2)(3)((4)(5),3),3)(6)(7)
```

A new algorithm[1] is now presented which performs the required compression. It is named *match and merge* after the two compressing operations it performs:

1. **Match**: Match a sequence of symbols with a previously detected compression pattern.
2. **Merge**: Merge two identical sequences to form a new compression pattern.

An example compression sequence is shown in figure 4.1. The algorithm uses a fixed-size window of symbols. Each time a symbol is added to the right hand side of the window, the algorithm attempts to find any possible matches or merges. In the diagram, the orange boxes represent symbols that have just been added to the window.

Beginning with four symbols in the window, `A B C C`, note that there is a duplicate sequence: a `C` followed by another `C`. These can be *merged* to form a new compression pattern, `(C)`, with 2 repetitions. Operations are performed repeatedly on the window until no further compression is possible. For example, on the fourth line the algorithm first performs a merge to form a new pattern, `(C)` and then performs another merge to form another new pattern `((A)(B)(C,2))`. Finally, now that this pattern has been established, further sequences of the same symbols can be *matched* with the pattern. This removes the matched symbols and increments the pattern's repetition counter.

4.1.2.2 Loop-level compression

Match and merge compression could be used to compress the entire loop, however, it is also important to take into consideration how the data will be processed later on. Since the oracle analysis will focus mainly on loop-carried dependences, it is important that it can efficiently determine from the control flow trace exactly which iteration of the loop a particular dynamic instruction occurred in and whether two dynamic instructions occur in different iterations. The control flow trace as described so far contains no explicit information regarding iterations of the main loop.

Therefore I have opted to use match and merge to compress each individual iteration of the main loop and to use another scheme to compress the overall loop. Each time an iteration of the loop completes, the compression pattern for that iteration is recorded along with the iteration number. For each subsequent iteration which has an identical compression pattern, the new iteration number is simply added to the original record. So an example compression for an entire loop might look like this:

[1]The algorithm is an original contribution of this thesis.

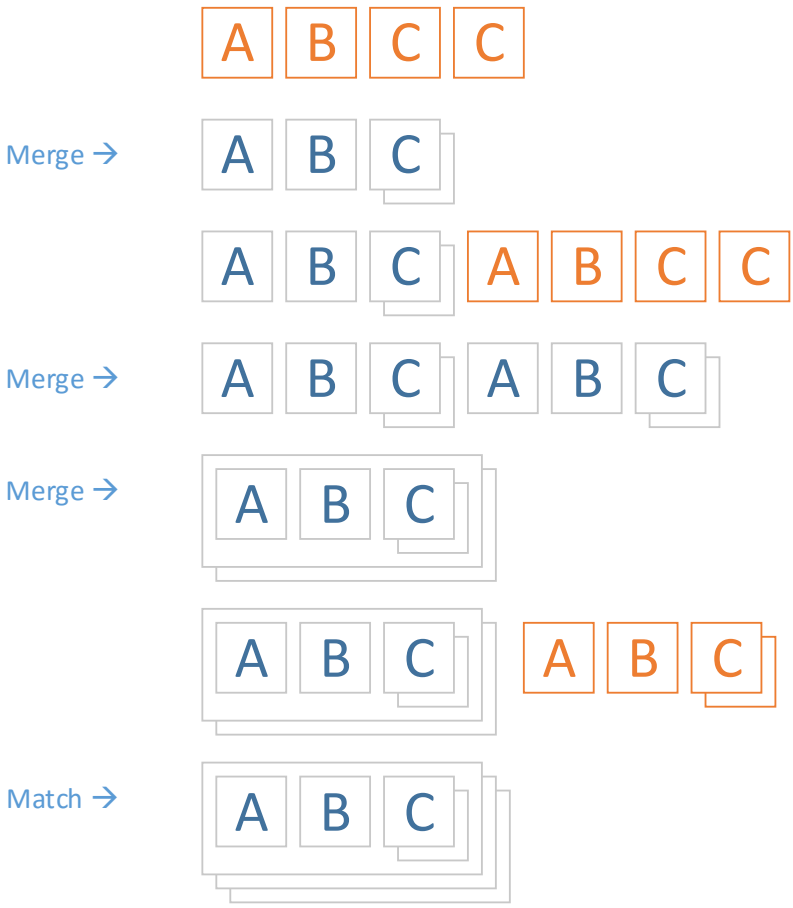

Figure 4.1: Sample sequence being compressed by match and merge. Orange boxes indicate symbols that have just been added to the window.

```
{0,2,4}  (A)(B)(D)(E)
{1,3,5}  (A)(B)(C)(E)
```

This indicates that iterations 0, 2 and 4 had compression pattern `(A)(B)(D)(E)` while iterations 1, 3 and 5 had compression pattern `(A)(B)(C)(E)`.

4.1.3 Dependence analysis

The next task is to analyse the traces to find every pair of instructions which conflict at runtime. The first step is to find pairs of memory set entries which access the same address for a given pair of instructions. Given two memory set entries, a naive approach to finding common addresses would be to iterate through every combination of addresses at $O(n^2)$ complexity. Since the instructions being studied will possibly have millions of dynamic instances, however, this would quickly become prohibitively expensive. Fortunately Kim et al. [69] find a solution whose complexity does not depend on the size of the memory set entries.

To demonstrate, consider the sample memory set entries for instructions `x` and `y` in figure 4.2. The memory set entries have strides of 4 and 3 respectively. Finding the common addresses for these two instructions is equivalent to solving the diophantine equation:

$$100 + 4x = 95 + 3y \quad (0 \leq x \leq 5, 0 \leq y \leq 7)$$

If the greatest common divisor (GCD) of the strides (in this case 1) divides evenly into the difference between the bases (in this case 5) then there may be common addresses. The first solution can be found with the extended Euclidean algorithm [87] and all subsequent solutions can be found by repeatedly adding the lowest common multiple of the strides (in this case 12).

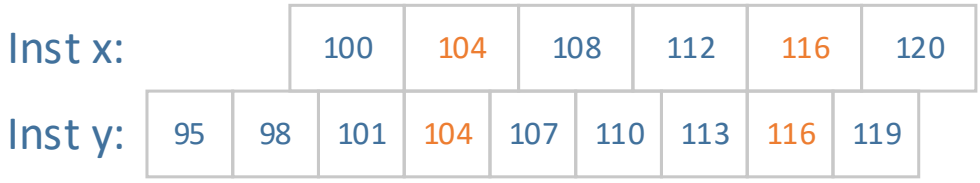

Figure 4.2: Finding common addresses as solutions to a diophantine equation.

From this algorithm it is possible to find all pairs of instructions which ever touch the same address; however, this would potentially include instructions which only touch the same address within a single iteration. To eliminate these pairs it is necessary to consult the control flow trace. Initially I tried storing all solutions to the diophantine equation and then looking up the iteration numbers of the dynamic instructions responsible for each common address. Although it was possible to build data structures which permitted efficient querying of the control flow trace to find iteration numbers, this proved to be too slow, especially in cases where the conflicting instructions were always in the same iteration so that every single solution had to be checked. It was possible to make this more efficient by observing that the number of dynamic instances of an instruction within a single loop iteration is usually constant across iterations (e.g. 1 for an instruction which is not in an inner loop). So if both instructions satisfy this property and the first solution to the diophantine equation involves dynamic instances in the same iteration, then all subsequent solutions will also involve dynamic instances in the same iteration and no inter-iteration dependence exists. Instructions are marked during trace collection to indicate if they satisfy the property. A convenient test is to simply check if the first and last solution to the diophantine equation are intra-iteration. If this is the case then all solutions in between will also be intra-iteration.

4.1.4 Worked example

In this section I will describe a sample loop for which the compiler over-estimates the DDG and show how the oracle DDG improves the parallel performance of such a loop. Consider the outer loop in listing 4.2 which contains 4 memory references:

1. **A**: Read `glob`
2. **B**: Write `glob`
3. **C**: Read `A[factor*count + i]`
4. **D**: Write `A[factor*count + i]`

The HELIX dependence analysis will generate the DDG shown in figure 4.3. Using this DDG, HELIX parallelises the loop and creates two sequential segments to synchronise each of the dependence cycles. The control flow graph of the code generated by HELIX is

Listing 4.2: Sample loop with difficult to analyse dependences.

```
for (count = 0; count < weight; count++){
  glob++;
  for(i = 0; i < factor; i++){
    int tmp = A[factor*count + i];
    tmp += count*5;
    if(tmp%2 == 0){
      A[factor*count + i] = tmp;
    }
  }
}
```

shown in figure 4.4. When the loop is parallelised, code in SS 1 can be run concurrently with code in SS 2 but each sequential segment can only be running on a single thread at any given time.

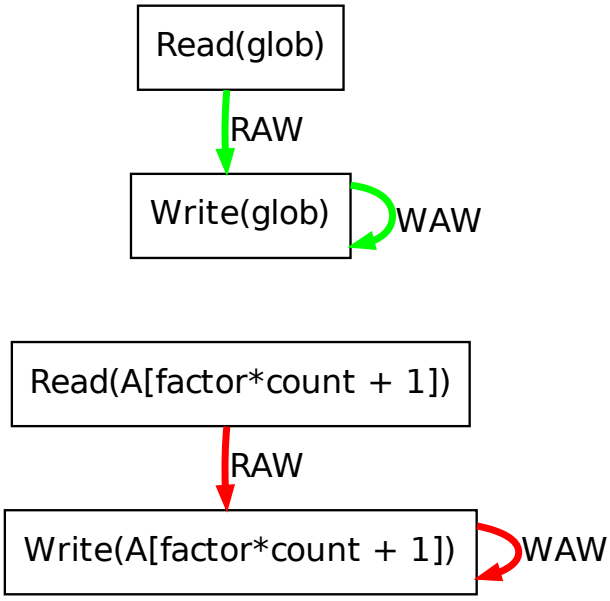

Figure 4.3: Compiler generated DDG for loop in listing 4.2.

Now the runtime traces are collected and dependence analysis is performed as described above. The colour of the edges in figure 4.3 indicates the outcome of the analysis. The green edges are confirmed to be part of the DDG but the red edges are not realised at runtime. The oracle DDG is a subset of the DDG shown in figure 4.3 consisting of only the green edges.

Finally the oracle DDG is fed back into the compiler and the loop is re-parallelised. Having removed the red edges from the DDG, the compiler no longer generates sequential segment 2. This code is now free to execute in parallel with any other code in the loop. The performance impact of the oracle DDG for this benchmark is demonstrated by the results in figure 4.5.

While the loop experiences virtually no speedup with HELIX using the compile-time DDG, the improved accuracy of the oracle DDG results in close to linear speedups. This is because the code in sequential segment 2 generated by the compiler accounted for a large proportion of sequential execution time. Without being able to run this code in parallel, HELIX could not have performed well on this loop. Evidently, over-estimating

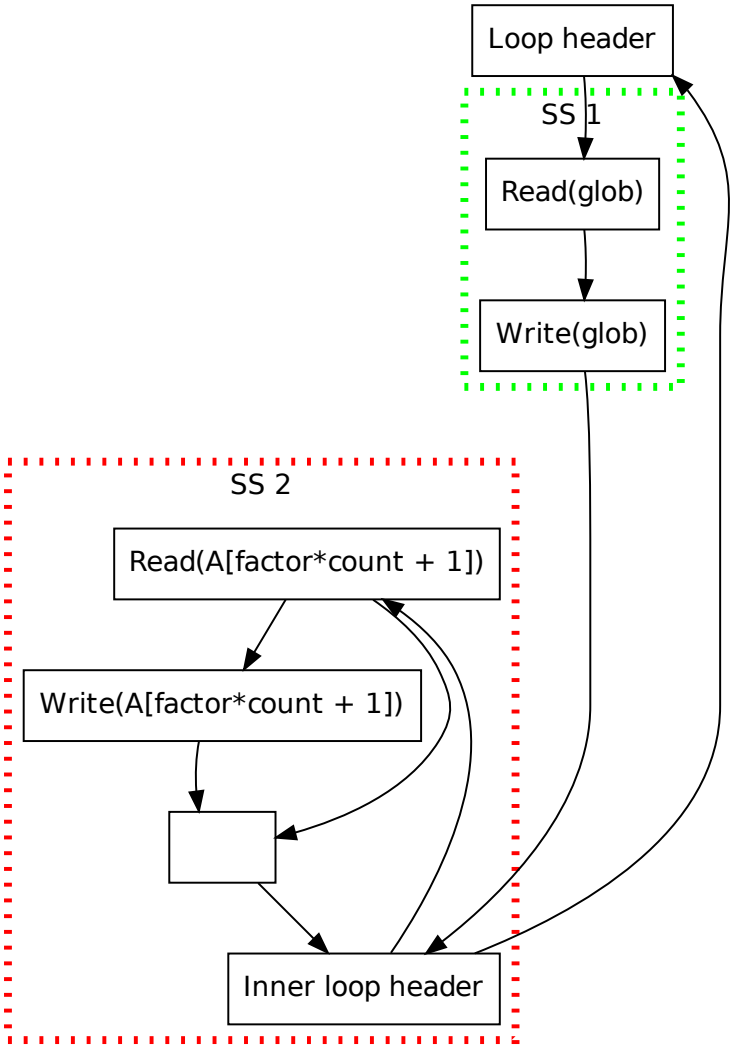

Figure 4.4: Simplified CFG for parallelising loop in listing 4.2.

the DDG can result in significant performance degradation for automatic parallelisation.

4.2 Evaluation

In this section I will generate oracle DDGs for a number of cbench benchmarks and evaluate the extent to which HELIX is over-estimating the DDG and the resultant parallel performance degradation. It is important to note that the results do not include dependences which are caused by local variables. This is explained in section 4.2.2.

4.2.1 Accuracy of HELIX static analysis

When the HELIX static analysis is run, a graph is generated of all the suspected dependences in the loop. One way of quantifying the accuracy of this analysis is to create an oracle DDG as described in section 4.1 and see what percentage of edges in the original graph have been removed. If the analysis is extremely accurate then it is expected that hardly any dependence edges would be removed, whereas if the analysis is rather poor it is expected that a large percentage of edges would be removed.

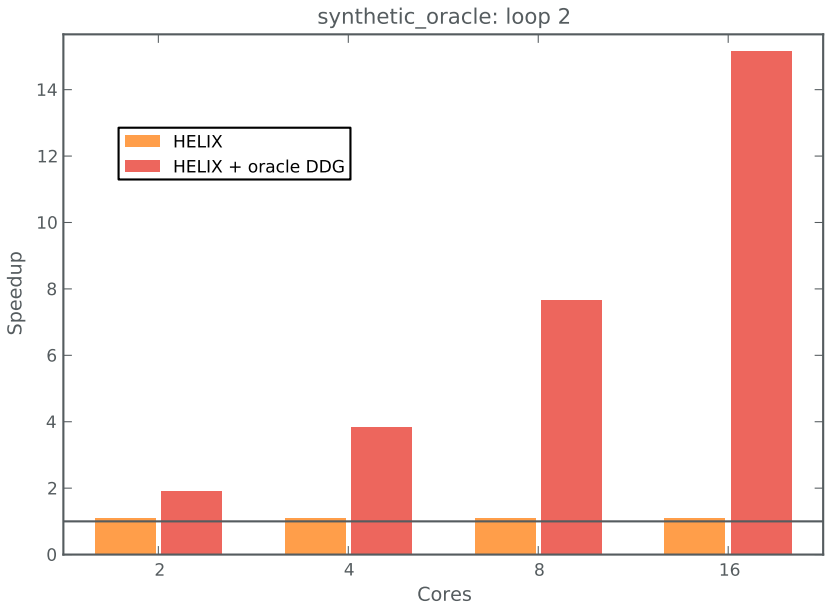

Figure 4.5: Parallel performance of loop in listing 4.2.

Figure 4.6 shows the percentage of edges which were removed from the DDG for various cbench benchmarks. Each benchmark consists of a number of different loops which were each analysed individually. The figures depicted in the graph were obtained by summing the number of edges in each of the compile-time DDGs (one per loop) and calculating what proportion of edges had been removed in total. The graph shows that the accuracy of the static analysis varies greatly for different benchmarks. The analysis was perfect for automotive_bitcount for example, with no dependences removed, whereas the analysis vastly over-estimated the number of dependences in automotive_susan_c with almost 100% of dependences removed.

Overall the graph shows that a significant proportion of dependence edges could be removed from the DDG for most benchmarks. This means that the compile-time analysis identified many pairs of instructions which it determined could potentially have accessed the same memory locations but in reality did not alias even once during execution.

Figure 4.6 is useful for getting an overview of the compile-time DDG accuracy, but to fully appreciate the potential of the oracle analysis it is instructive to look at the individual loops. A breakdown of these results by loop is shown in figure 4.7. Each bar indicates the percentage of dependences which were removed by the oracle analysis for a particular loop in the benchmark. "Coverage" indicates the percentage of sequential execution time during which this loop was executing. Coverage figures may sum to more than 100% due to loop nesting (i.e. time spent executing inner loops also contributes to the coverage of the outer loops).

From these results it is evident that even within a benchmark different loops may have vastly different DDG accuracy. In addition, there are high-coverage loops, such as loops C and D in automotive_susan_c, for which a very large proportion of dependence edges can be removed. These loops may hold promise for gaining parallel performance with an improved compile-time dependence analysis. Conventional wisdom would suggest that when the profile-determined oracle DDG is used to parallelise, it will be possible to exploit more parallelism than previously.

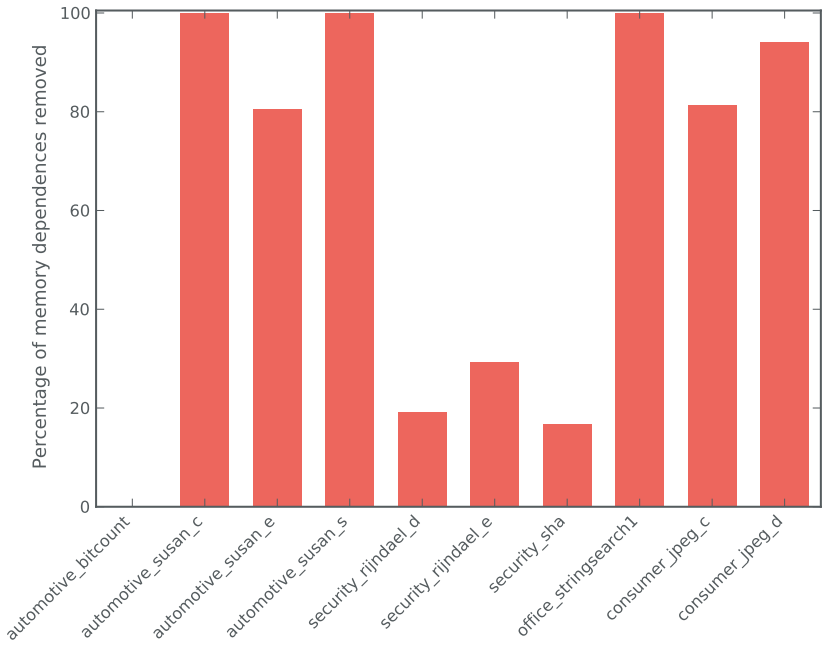

Figure 4.6: Percentage of memory dependences removed by the oracle analysis for cbench benchmarks.

Listing 4.3: An example of local variables causing loop-carried dependences.

```
for(int i = 0; i < 100; i++){
    if (A[i] > 0){
        x++;
        B[x] = i;
    }
}
```

The raw data from which these graphs were generated is included as appendix A for reference.

4.2.2 A note on local variables

When HELIX parallelises a loop it detects loop-carried dependences through memory and through local variables. Dependences can occur through local variables when a variable is written in one iteration of the loop and read in a subsequent iteration. Consider variable x in listing 4.3. For sequential execution x would be allocated on the stack but for parallel execution its value must be communicated between threads. Therefore HELIX allocates shared memory to store the variable which is accessed by all threads.

The oracle analysis only tracks references which refer to heap-allocated data or to stack-allocated arrays and does *not* track local variable references, even when they cause loop-carried dependences. The reason for this is that detecting dependences caused by local variables is easy for the compiler since their addresses never change. The only scenario in which the compiler would detect a dependence between two local variables which did not actually exist would be if one instruction had a control dependence which caused it to never execute. Since tracking local variable references would have hugely

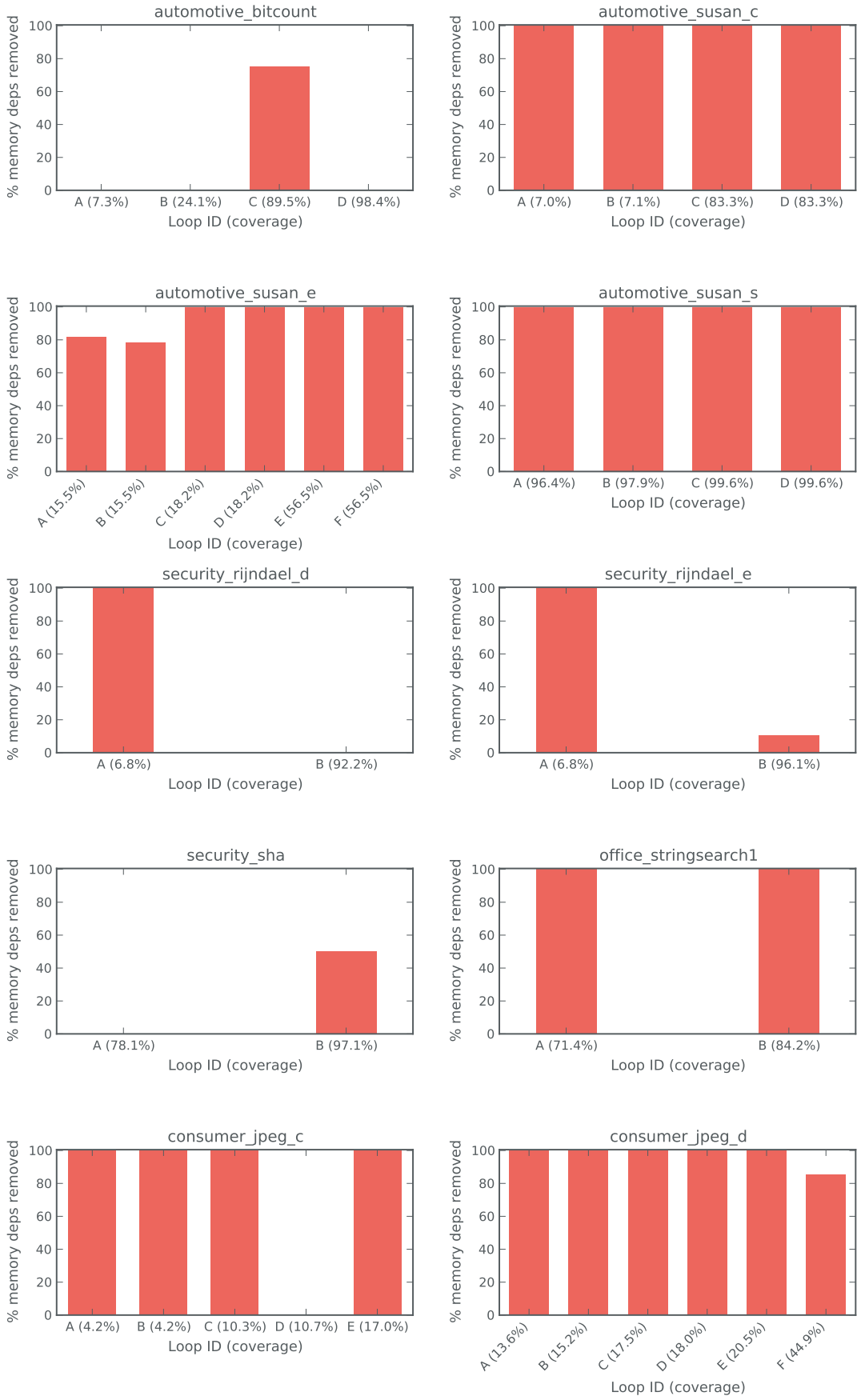

Figure 4.7: Percentage of memory dependences removed by the oracle analysis broken down by loop.

increased the overhead of the analysis and this scenario does not occur frequently I decided to omit local variables from the analysis. Therefore, all loop-carried dependences through local variables will always be included in the oracle DDG when it is used as input to the compiler.

Figure 4.8 shows that 100% of memory dependences are removed for some loops. This does not necessarily mean the iterations can run completely independently since dependences caused by local variables may still cause the loop to be sequentialised.

4.2.3 Parallel performance with the oracle DDG

So far I have simulated a perfect static analysis to produce the most accurate DDG that a compiler could achieve and find the upper limit of the accuracy of static analysis. This section shows what the upper limits of HELIX-style parallel performance are, given a perfect static analysis. This is useful to determine exactly how much performance is being lost as a result of poor compile-time analysis. The HELIX paralleliser is run as normal but, rather than performing the static dependence analysis, the compiler simply reads the oracle DDG previously generated. The compiler then proceeds to parallelise the code based entirely on dependences that have been identified as real by the oracle analysis.

Figure 4.8 shows the performance of the individual loops[2] in the various benchmarks when running with 16 cores. Surprisingly, despite the greatly increased accuracy of the DDG, only two loops in the entire suite experienced any noticeable speedup whatsoever. For some loops HELIX performance is already so good that no further improvements are possible. For example, in automotive_susan_s loop C already achieves linear speedup with plain HELIX. Therefore, it was impossible to improve on the performance that HELIX was already offering. Evidently, the dependences which were removed from the DDG for this benchmark were not restricting HELIX's ability to effectively parallelise the code.

However, there are still unanswered questions. Loops A and B in automotive_susan_c achieve no speedup with HELIX and still did not improve even though almost all of the memory dependence edges for these loops were removed. There are many other loops which do not appear to be affected by the improved dependence analysis at all even though there appears to be considerable scope for further speedup. These loops will be analysed in section 4.3 to gain further insight.

4.3 Analysis

Consider loop B in automotive_susan_e: the oracle DDG analysis was capable of removing almost 80% of the dependences (figure 4.7) but the resultant reduced DDG did not enable the extraction of any additional parallelism (figure 4.8). However, there is considerable parallelism available in this loop, as shall be demonstrated in section 5.2 (figure 5.3). The source for this loop is shown in listing 5.2 and will be discussed in more detail in section 5.2.1.2. The key issue in this kernel is that the induction variables are potentially modified in the body of the loop. This makes it impossible to start an iteration until the previous iteration has completed. The static DDG analysis detects a number of dependences since

[2]Since the oracle analysis is run for each loop individually it was not trivial to parallelise several loops at once to give an overall program speedup since the oracle DDG could be different for each loop. Therefore only individual loop spedups are shown.

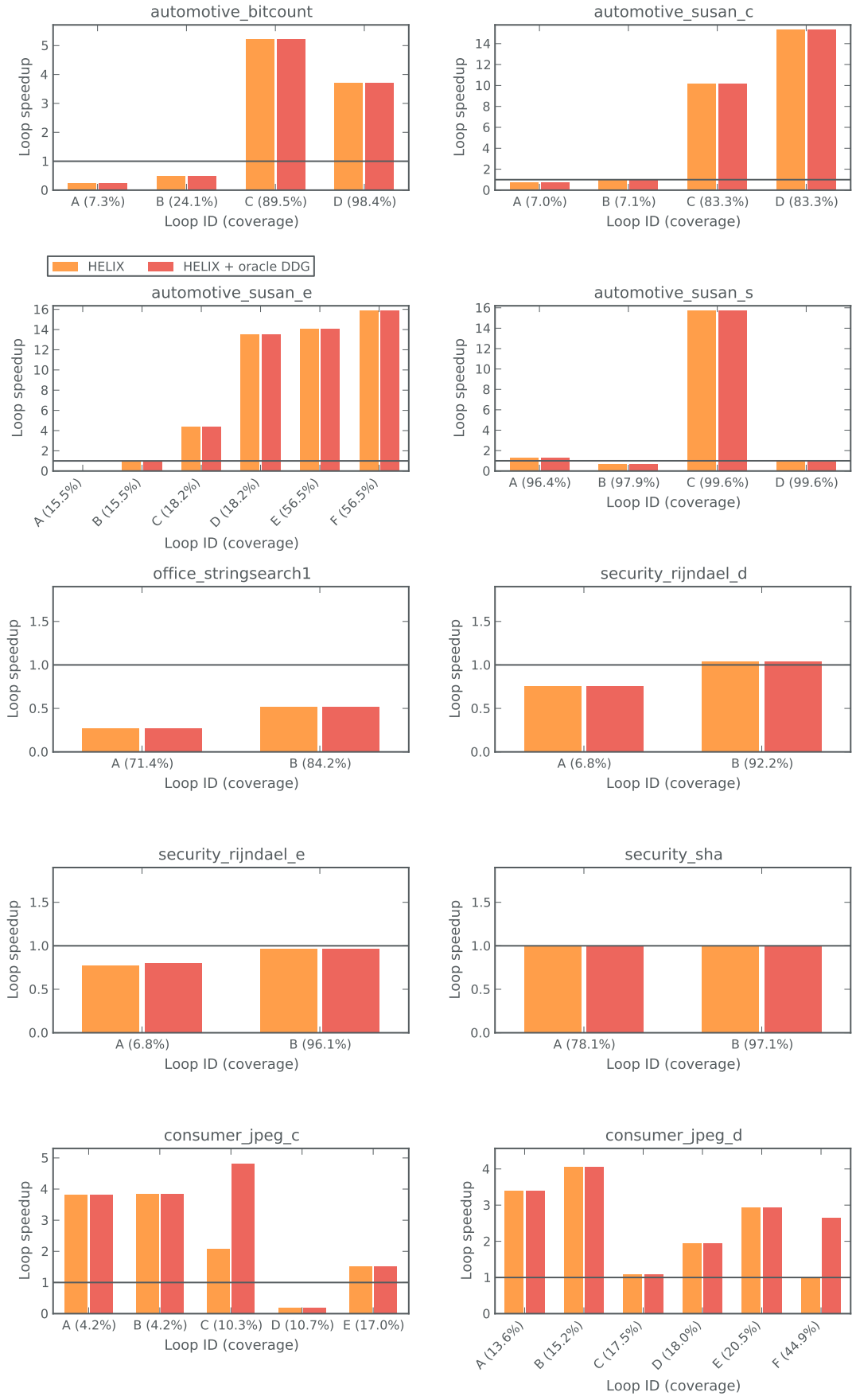

Figure 4.8: Performance of individual loops with the oracle DDG running with 16 cores.

Listing 4.4: Loop A in security_rijndael_e.

```
for(i = 0; i < 16; ++i){
    /* Enter sequential segment */
    inbuf[i] ^= outbuf[i];
    /* Exit sequential segment */
}
```

Listing 4.5: Loop C in consumer_jpeg_c.

```
for (col = 0; col < num_cols; col++) {
  r = GETJSAMPLE(inptr[RGB_RED]);
  g = GETJSAMPLE(inptr[RGB_GREEN]);
  b = GETJSAMPLE(inptr[RGB_BLUE]);
  inptr += RGB_PIXELSIZE;
  /* Enter sequential segment */
  outptr0[col] = (JSAMPLE) ((ctab[r+R_Y_OFF] + ctab[g+G_Y_OFF] + ctab[b+B_Y_OFF]) >>
      SCALEBITS);
  outptr1[col] = (JSAMPLE) ((ctab[r+R_CB_OFF] + ctab[g+G_CB_OFF] + ctab[b+B_CB_OFF]) >>
      SCALEBITS);
  outptr2[col] = (JSAMPLE) ((ctab[r+R_CR_OFF] + ctab[g+G_CR_OFF] + ctab[b+B_CR_OFF]) >>
      SCALEBITS);
  /* Exit sequential segment */
}
```

an array which is indexed by these induction variables is written and read a number of times within the loop body. In practice, many of these individual accesses never conflict with each other, resulting in a substantial reduction in the number of dependences in the oracle DDG. However, the *only* dependence that really matters is the dependence caused by the induction variables which cannot be removed because of the unpredictable way they are modified within the loop.

Loop A in security_rijndael_e does enjoy a very slight speedup for the oracle DDG relative to plain HELIX. The code is shown in listing 4.4. The dependence analysis is unable to determine with certainty that pointers `inbuf` and `outbuf` point to separate allocations and therefore must enforce sequential execution of the statement. The oracle analysis detects that these references never conflict and enables the removal of that sequential segment. Unfortunately the loop body is too short to experience speedup relative to the sequential baseline.

Loop C in consumer_jpeg_c gained a significant speedup with the oracle DDG. This loop converts a block of an image from RGB encoding to YCC encoding. The source is shown in listing 4.5. The pointers `outptr0`, `outptr1` and `outptr2` all point into different parts of a single dynamically allocated memory region. Although they never alias, HELIX is unable to disambiguate them and creates the sequential segment marked in the listing. With the oracle analysis this sequential segment is removed. Loop F in consumer_jpeg_d performs the inverse discrete cosine transform (IDCT) on a row of an image. The code for this loop is too large to reproduce here. Again HELIX generates a spurious sequential segment due to the inability to precisely disambiguate array references.

Hind [64] observes the difficulty of finding a meaningful metric to quantify the accuracy of pointer analysis. In figures 4.6 and 4.7 I have reported what is essentially equivalent to what Hind describes as the *direct metric*: the number of pairs of objects which the analysis determines may alias. As an alternative, Hind suggests reporting how the analysis affects some runtime property, in this case the performance of the parallelised code. The

discrepancy between these two results (huge improvements with the direct metric versus no improvement with the runtime property metric) demonstrates the challenge of choosing an appropriate metric.

4.4 Summary

In this chapter I have described a method for creating an oracle data dependence graph which includes only those data dependences which are actually manifested at runtime. I evaluated the analysis, showing that the compiler significantly overestimates the size of the DDG. Surprisingly, removing spurious dependences from the graph did not result in improved parallel performance for these benchmarks in most cases. It would appear that, without the implementation of additional sophisticated transformations which can improve privatisation, the dependence analysis is already good enough to extract whatever parallelism is available in these benchmarks to a HELIX-style parallelisation model. However, the benchmarks that have been studied are fairly small and do not exhibit the complex behaviour found in larger benchmarks which is more difficult for the compiler to analyse. Therefore, it would be necessary to increase the scope of this limit study to include such large benchmarks to test whether this result holds in general.

The oracle DDG is not sufficient to represent the complete dynamic behaviour of the program since even a single manifestation of a dependence at runtime results in an edge being inserted in the DDG. The question of whether or not additional dynamic parallelism exists for these benchmarks is still unanswered. This will be the subject of the next chapter.

Chapter 5

Uncovering dynamic parallelism

In chapter 4 I looked at the limits of what could be achieved using purely static parallelisation by simulating a perfect dependence analysis. It was found that even with an ideal analysis, performance could not be improved above the current HELIX baseline. However, the oracle DDG includes *all* dependences, even ones which only occur once throughout the execution of the program. Therefore it does not give an accurate account of dynamic behaviour. In this chapter I go further towards identifying the ultimate limits of cyclic-multithreaded parallelism by delving into the realm of speculative execution.

Thread-level speculation (TLS) involves many complicated trade-offs associated with the runtime expenses of tracking memory references, identifying conflicts and re-executing incorrect computation. To get a pure idea of the available parallelism, independent of the restrictions of any specific TLS implementations, I created an ideal speculative model which is restricted only by dataflow limits, i.e. reads cannot be reordered with respect to the write which they consume. It should be noted that this model is purely hypothetical and a realistic implementation of this model is not offered. It serves to provide an upper bound on what could be achieved with TLS based on a HELIX-style cyclic-multithreaded parallelisation.

This chapter begins with a precise description of the model and a theoretical outline of how such a model might be implemented in section 5.1. Results are presented in section 5.2 showing an upper bound on speculative performance and an analysis of why speculation achieves speedups where the oracle DDG does not. Finally I look at patterns and statistics that can be obtained from the model which can be used to direct practical speculation implementations in section 5.3.

5.1 Ideal dataflow speculation

Write-after-read dependences and write-after-write dependences can, in theory, be removed with memory privatisation so the true limit to performance enhancement with speculation is read-after-write dependences: the dataflow limit [88]. The ideal dataflow speculation model is restricted only by data which is produced in one iteration and then consumed in a later iteration. In this model each thread can make whatever writes it likes without delay, but must stall any read which consumes data from a previous iteration. It is assumed, for the purposes of finding an upper bound, that data can be propagated at zero cost from the producing thread to the consuming thread such that a thread which is stalled on a consuming read can proceed instantly once the producing write has been executed.

In this section the term "dataflow" is used to describe execution determined by the flow of data between iterations of the loop. This is in contrast to the classical usage of the term to refer to execution of instructions based on the availability of inputs to the instructions. My dataflow model does not take advantage of any parallelism which is available within a single iteration of the loop. Rather, each individual iteration uses ordinary control flow and the execution of instructions which consume data produced by previous iterations is dependent on that data becoming available. Classical dataflow architectures are generally based on the compiler producing a static dataflow graph so that for each instruction it can be determined before the execution of the instruction what its dependences are. By contrast, I do not assume that an instruction's memory-based dependences can be determined prior to execution and instead rely on speculative execution to achieve the maximum possible parallelism, independent of the compiler.

5.1.1 Hypothetical implementation

Ideal dataflow speculation is primarily a limit study and is not considered to be a realistic model of implementation for a TLS system. However, it is possible to imagine a hypothetical implementation of such a model on a real system. It is worthwhile discussing the possibility of a real implementation to justify the belief that this model represents a realistic upper limit of what could be achieved.

Firstly, a mechanism is needed to break write-after-read and write-after-write dependences. This is simplified by the nature of the HELIX parallelisation model. For each thread a write buffer is needed which stores all non-thread-local writes of an iteration. This allows arbitrary reordering of write instructions between different parallel threads. In each write buffer only a single entry is stored per address so that each write to an address overwrites the value previously stored. The value which is ultimately visible to subsequent threads is the last value written in an iteration. In addition to the address and value of each write, a timestamp must also be stored to allow the rollback of specific writes.

Ensuring the satisfaction of read-after-write dependences is more complex since it is necessary to ensure that each read returns the value last written within the current thread or, if no writes have occurred, the final value of the most recent thread which wrote to that address. So each time a read is encountered, the runtime attempts to look up the address in the following order:

1. The current thread's write buffer.
2. The write buffers of other threads starting with the thread immediately older than the current thread and working back to the oldest.
3. Main memory.

Of course, because the threads are executing speculatively, it is possible that after a read is executed an older thread will write to the same address. This will cause the read, and all subsequent execution, to become invalid. To allow fine-grained rollback, a read log must be kept which records the timestamp of the first read in the iteration for each address and the program counter address of the read. In addition, the write buffer maintains a log of all local writes and timestamps so that these can also be rolled back.

To facilitate rollback of invalid execution, each time a thread performs a non-local write, it must check the read logs of all *younger* threads and invalidate any threads which

have read from that address. Any addresses stored in the local write log with younger timestamps are reverted to their original values and the program counter is restored to the point where it can begin executing again.

The overheads of this system would likely be prohibitively high in a real implementation due to the expense of looking up multiple buffers and logs on each read or write.

5.1.2 Timing model implementation

The dataflow model is built as an extension to the HELIX timing model which was described in section 3.5. Since all the code is actually executed completely sequentially, there is no need to implement a complex fine-grained rollback scheme. As each iteration is executed, the model records which core the iteration is assigned to and how many cycles are needed to execute each instruction. Memory accesses are instrumented with callbacks to the model. For each callback the model simulates the hypothetical implementation described above to determine on which cycle each instruction would have actually executed.

For each thread, a write set is created which records the address and timestamp of each write in an iteration. The write set is stored as a hash table to facilitate quick lookup for large sets. For each read, the model first does a lookup in the write set of the currently executing thread. If the write set contains the address then execution may proceed without other action since the value is already available. Otherwise a lookup must be performed in the write sets of each previous thread. If the address has been written by a previous iteration, the currently executing thread is delayed until its cycle counter is equal to the timestamp of the latest write to that address by the most recent thread. In this way dataflow dependences are preserved by ensuring that no read can execute until the completion of the write on which it depends.

A sample execution of the dataflow model is depicted in figure 5.1. When the first iteration is executed, the cycles elapsed are assigned to virtual core 0 and the cycle number of each store is recorded. During the second iteration when `Load X` is executed, the model will add cycles to virtual core 1 until its counter is greater than the cycle number of the most recent `Store X` (in this case, the one recorded by virtual core 0). In accordance with the hypothetical implementation (section 5.1.1), stores are modelled as being buffered by the virtual core so there is no need to stall `Store Y` on virtual core 1 until after the same store on virtual core 0. Delays are only added to allow loads to always see the most sequentially recent value.

Certain events such as calls to the operating system cannot be executed speculatively and cannot be analysed and instrumented by the compiler. I modelled these as memory operations which both read and write to every location in memory. When such an event occurs, the model delays the executing thread until all writes in older iterations have completed. In addition, all younger iterations cannot perform any reads until the event completes (writes are modelled as buffered so are safe to execute anyway).

5.2 Results

Figure 5.2 shows the results of the ideal dataflow timing model compared to the performance of HELIX for the cbench benchmarks I am studying. These results were obtained by parallelising each loop individually, running the benchmark once for each parallelised

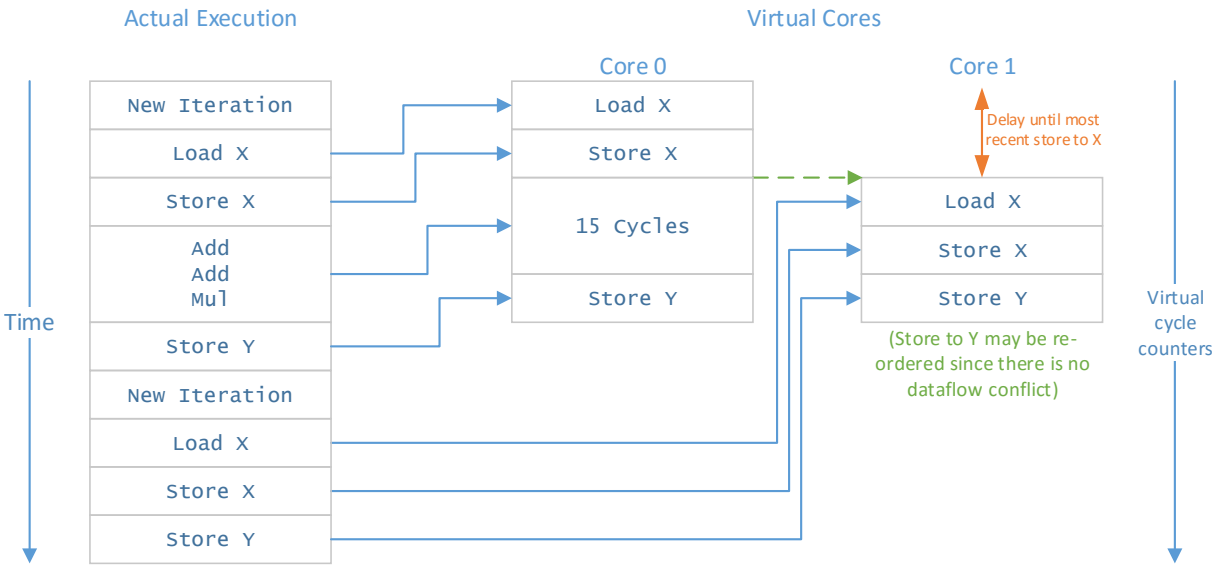

Figure 5.1: Sample execution of the dataflow model.

loop, running the loop selection algorithm (section 3.2.5) and, finally, running the benchmark again with the best selection of loops. Speedups are measured relative to the execution time of the original sequential code produced by ILDJIT. From these results it is evident that for some cbench benchmarks there is significantly more parallelism available than is currently being exploited by HELIX. However, for some benchmarks, such as office_stringsearch1, there is no parallelism available whether using HELIX or the dataflow model. The lack of parallelism may be inherent to the algorithm or may be due to the manner in which the algorithm was expressed by the programmer. The model cannot automatically distinguish between these causes but in section 5.2.1 I identify several cases where coding artefacts have restricted the exploitation of parallelism.

To gain some insight into this it is useful to look at a breakdown of the speedups of the various loops that constitute the benchmark. Figure 5.3 shows the individual loop speedups when run with 16 cores. Each loop is labelled with its coverage, defined as the proportion of total program execution time spent within the loop during sequential execution. Having broken down the benchmark into its constituent loops a much more pronounced difference can be seen between the dataflow performance and the HELIX performance. While HELIX performs well on some high-coverage loops, it is generally beneficial to improve the speedups of low-coverage loops since it has previously been shown that a combination of multiple low-coverage loops can give the best speedups [2, 4].

Some loops, such as loop B in automotive_susan_c, have no speedup with HELIX but almost linear speedup with the dataflow model. For some loops HELIX is not capable of exploiting the large amount of parallelism which exists. In the next section I will look in more detail at some individual benchmarks and loops to understand why HELIX is not showing speedups in some cases.

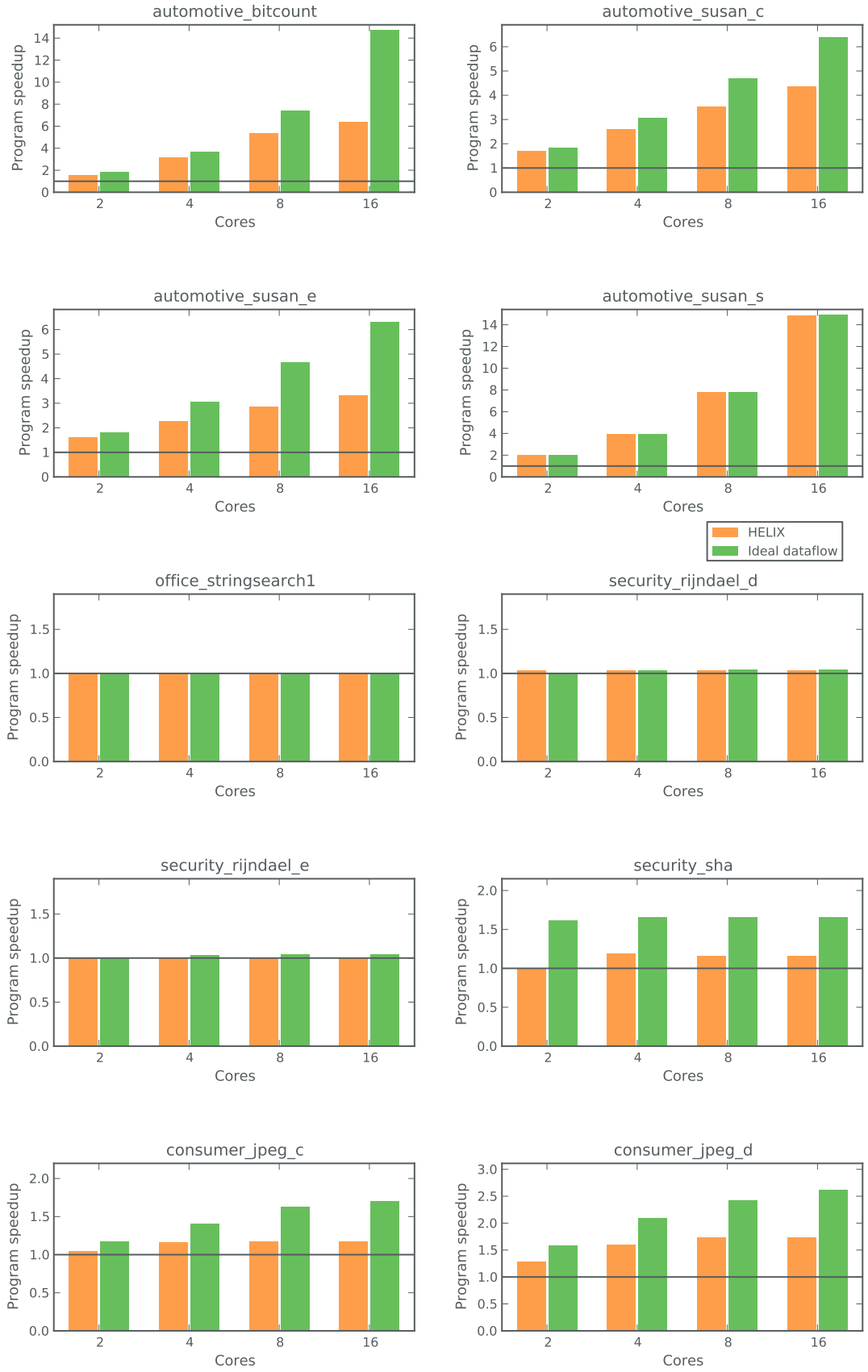

Figure 5.2: Cbench results for ideal dataflow model.

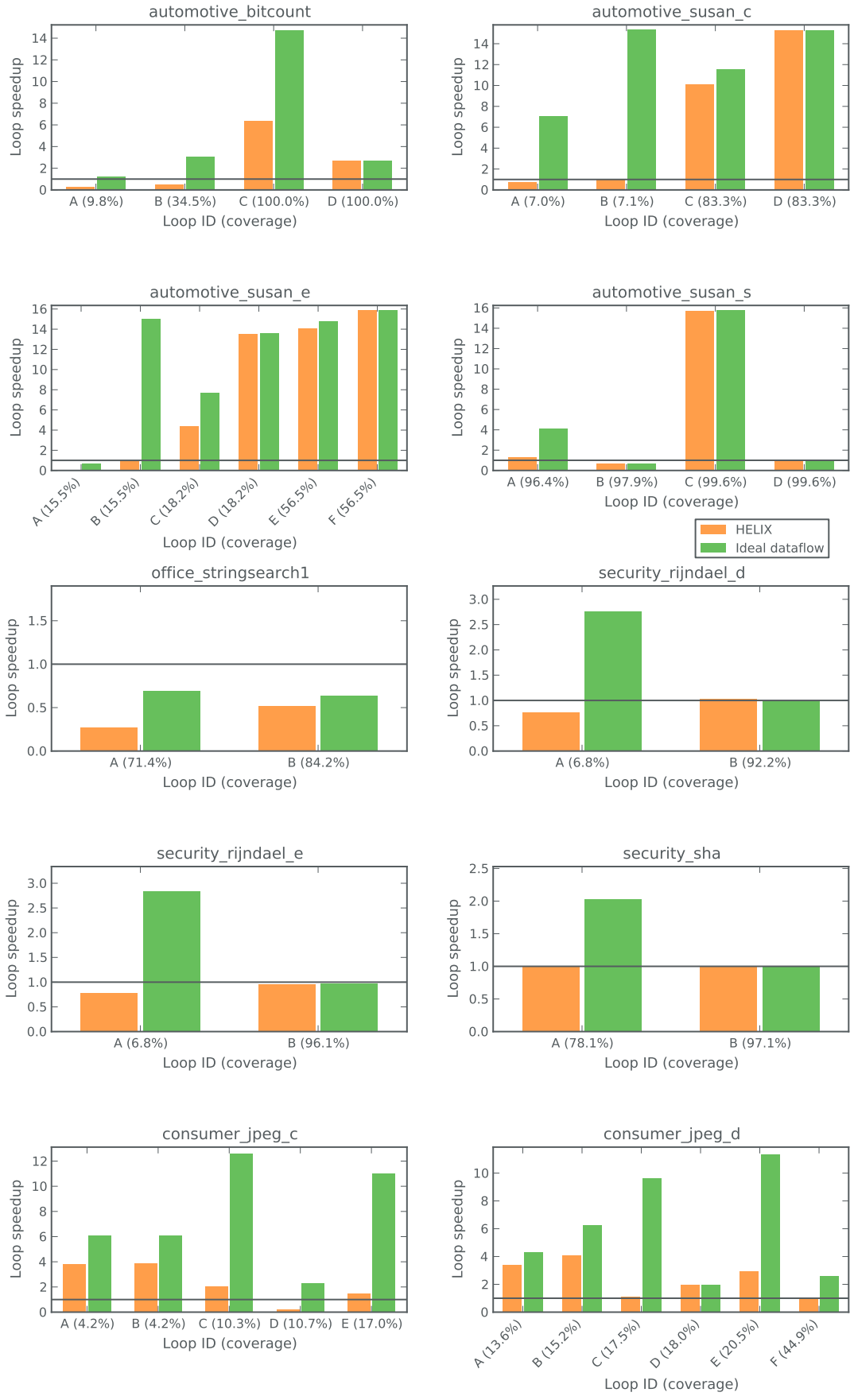

Figure 5.3: Loop breakdown for ideal dataflow model run with 16 cores.

Listing 5.1: Second kernel from automotive_susan_c.

```
n=0;
for (i=5;i<y_size-5;i++){ /* Loop B */
    for (j=5;j<x_size-5;j++) { /* Loop A */
        x = r[i][j];
        /* Enter sequential segment */
        if (x>0)  {
            if (/* Abbreviated: compare x to each pixel in window */) {
                corner_list[n].info=0;
                corner_list[n].x=j;
                corner_list[n].y=i;
                corner_list[n].dx=cgx[i][j];
                corner_list[n].dy=cgy[i][j];
                corner_list[n].I=in[i][j];
                n++;
                if(n==MAX_CORNERS){
                    fprintf(stderr,                    );
                    exit(1);
                }
            }
        }
        /* Exit sequential segment */
    }
}
```

5.2.1 Case studies

5.2.1.1 automotive_susan_c

This benchmark is based on the SUSAN algorithm to find corners in an image [89]. The benchmark consists of two kernels, and each one iterates across the image and performs some processing at each pixel site. The first kernel, comprising loops C and D, creates an initial response output highlighting features which look like corners. The second kernel, comprising loops A and B, passes a window across this initial response identifying local maxima within the window, with each maximum being added to a list of corners (see figure 5.4). Smith et al. [89] have published full details of the algorithm.

Of interest is the second kernel which finds local maxima in a sliding window. The source code of this kernel is shown in listing 5.1, which has been abbreviated here due to the verbosity of the original. The loops have been labelled in correspondence with the labels in figure 5.3.

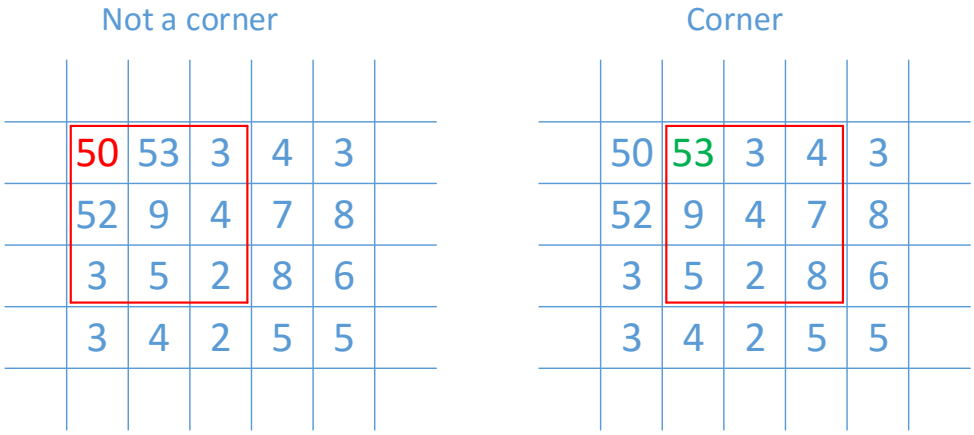

Figure 5.4: Second kernel from automotive_susan_c performs non-maximal suppression.

The difficulty with this benchmark centres around the calculation of the variable `n`. When a corner is detected, the comparison between `x` and each of the other pixels in the window returns true and an entry is inserted into the array `corner_list`. `n` is the index into this array and it only gets updated when an entry is inserted. Therefore, on any given iteration of the loop, it is unknown what the value of `n` is until the `x` comparison tests from all previous iterations have completed. When HELIX compiles this loop, it must insert a sequential segment to order the accesses to `n`, sequentialising a portion of the loop. This results in a major performance hit for HELIX which can now achieve no speedup whatsoever.

By contrast, the dataflow model can execute in parallel freely until it actually needs to read the value of `n`. At this point, whichever thread is reading `n` must stall until the cycle on which the most up-to-date value of `n` was written. By the nature of the algorithm, the number of corners in an image will likely be small relative to the number of pixels. Therefore, it is likely that many iterations will pass between one corner and the next and the dataflow stall time will be minimal.

Of interest is the fact that the outer loop, B, achieves almost linear speedup compared to the inner loop, A, which achieves a speedup of only 8 on 16 cores. The initial response created by the first kernel is a sparse image where most pixels are set to zero apart from the points suspected of being corners. These suspected corners tend to come in clusters together and the kernel being studied reduces these clusters to a single point which is recorded as the corner.

So, for the vast majority of iterations of A, there is essentially no work to be done and these take a negligible amount of time. For the input image used in this experiment for example, only 0.4% of pixels were non-zero. As a result, even though the inner loop performs a large number of iterations, on average for this input data, the number of suspected corners per row of the image is only around 10. With only 10 iterations per invocation of the loop actually performing any work, speedups larger than 10 cannot be expected.

It may be noted that, unless there is a requirement that the corners be entered into `corner_list` in a specific order, there need not be any dependence caused by this data structure since it would be possible to create private arrays for each thread which are then merged once the loops completes. However, without some information from the programmer to indicate that the order is not important, it is necessary for the paralleliser to preserve sequential semantics and ensure the order is the same as in the sequential version. This is an example of how coding artefacts may restrict the parallelism which can be exploited automatically.

5.2.1.2 automotive_susan_e

This benchmark is very similar to automotive_susan_c but rather than detecting corners it detects edges. As shown in figure 5.3 the benchmark contains 6 significant loops. These are organised into 3 kernels, each comprising two loops which iterate over the image to be processed. The first two kernels create an initial response image. The third kernel "thins" the initial response by looking at each pixel site and applying a number of rules based on the number of connected neighbours each identified edge site is permitted to have. This creates an output image with thin continuous edges.

Loop B is the outer loop of the third kernel which performs thinning on the initial response image. The source code for this kernel is too large to reproduce here, but the

Listing 5.2: Third kernel from automotive_susan_e.

```
for (i=4;i<y_size-4;i++){ /* Loop B */
  for (j=4;j<x_size-4;j++){ /* Loop A */
    if (mid[i][j] is an edge point) {
      n = number of neighbours to mid[i][j] which are edge points
      if (n==0) /* No neighbours, remove point */
        mid[i][j]=100;

      if ( (n==1) && (mid[i][j]<6) ) { /* One neighbour, extend line */
        Extend the line in the direction opposite to the neighbour
        If we extended backwards relative to the direction of the window,
          i and j must be decremented to reprocess the point that was added.
      }

      if (n==2) { /* Two neighbours, straighten the line */
        Alter window so that line is straighter, example:
          X O X       X X X
          O X O  -->  O O O
          O O O       O O O
        i and j may be decremented to reprocess a point that was added
      }

      if (n>2) { /* More than two neighbours, thin out the window */
        Remove points so that the edge is a single thin line.
        i and j may decremented to reprocess altered pixels
      }
    }
  }
}
```

algorithm is roughly described by the pseudocode in listing 5.2. The loops have been labelled to correspond with the labels in figure 5.3. This kernel passes a 3 by 3 window across the initial edge response image. At each site where the centre pixel was identified as an edge point, the number of neighbours which are also edge points are calculated and stored in variable `n`. Then the kernel performs various actions depending on how many neighbours the edge point has (also depicted in figure 5.5):

- `n == 0`: The pixel has no neighbours, it is an orphan and unlikely to be part of a real edge. Remove the edge point.
- `n == 1`: The pixel looks to be at the end of an edge. Attempt to extend the edge in the correct direction (i.e. the direction opposite to the neighbour). Move the window so that any added pixels get reprocessed.
- `n == 2`: The pixel is part of a line. Attempt to straighten the line. Move the window to reprocess any modified pixels.
- `n > 2`: The pixel is in a crowded region of edge points. The edge should be a well-defined thin line so some edge points in the window are removed. Move the window to reprocess any sites where pixels have been removed.

A similar situation to automotive_susan_c arises where there is a fairly sparse data structure with most pixels in the image requiring no processing whatsoever. The edge image is considerably more dense than the corner image from automotive_susan_c, however, with 6.5% of pixels requiring some processing in this case.

For HELIX this kernel is extremely problematic for two reasons. Firstly, the induction variables `i` and `j` are modified within the loop in a data-dependent manner. This means

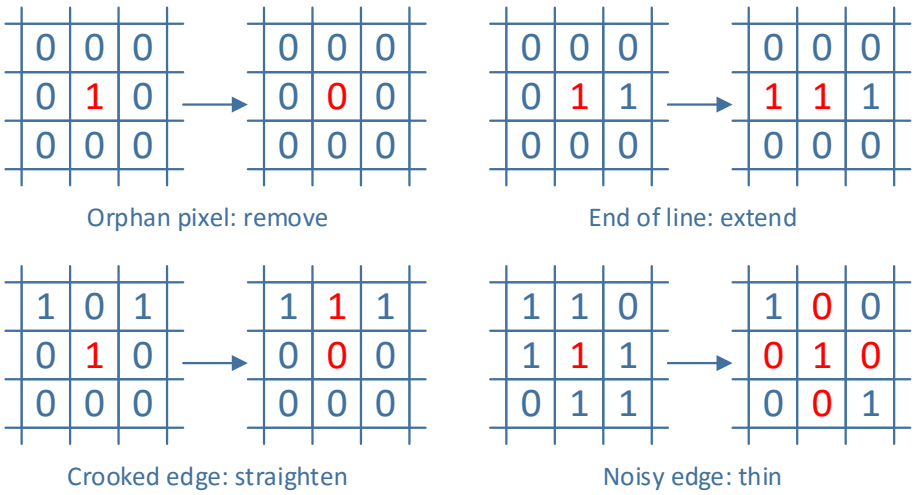

Figure 5.5: Third kernel from automotive_susan_e performs edge thinning.

that HELIX cannot privatise the variables as is normally done for such induction variables as the value in any particular iteration can not be known until the previous iteration has completed. This by itself would probably have the effect of sequentialising the entire loop. Even without that issue, the kernel works on the initial response image, `mid`, in-place. Entries in the image may be read in iterations subsequent to them being written. This would also prevent HELIX from effectively parallelising the loops.

This explains the poor HELIX performance, but what does it say about the dataflow model? The kernel does contain a significant amount of inherent parallelism since the window works on a small portion of the image at a time and what happens in one part of the image in general does not affect another. However, the paralleliser is limited by the iteration order specified in the original algorithm and so even the parallelised version of the loop must consider each pixel in the order of the source loop. The pixels are processed in order along each row, one row after the other. Edges are generally continuous lines so processing the image in this order means that the pixel sites which cause conflicts are likely to be processed at around the same time for horizontal lines. This limits the amount of parallelism which can be exploited by the dataflow model.

For the inner loop, A, this clustering of conflicts severely affects performance and no speedup is achieved for this loop whatsoever. However, according to figure 5.3 loop B achieves a linear speedup! To understand this, consider the work that is performed by an iteration in each of these loops. For loop A, an iteration is a single site and the subsequent iteration will be a site offset by one pixel in some direction. Any iteration which performs work is almost certain to conflict with the subsequent one because a pixel written in one iteration will be read by the next.

Contrast this with loop B where each iteration processes an entire row of the image. There will naturally be conflicts between adjacent rows of the image as well. However, any particular edge is likely to be localised to a specific section of the image. Since each iteration of loop B traverses the entire width of the image, there is always enough room for each of the cores to operate on independent sections of the data. What happens during the execution of the parallelised version of loop B is that, in the initial phase, the threads spend a lot of time stalled while data dependences are being resolved. After this initial phase the threads become staggered along the width of the image so that each one is operating on completely separate parts of the data structure.

This is an interesting kernel from the point of view of parallelism because the overly specific nature of the algorithmic description in C prevents any effective parallelisation by HELIX. However, there is not necessarily a single acceptable output result and even choosing to traverse the pixels in a different order could result in slightly different output. Linear speedup could be achieved on this kernel with purely static parallelisation if we could relax the semantics a little such that overlapping regions did not need to be processed in a specific order. Some previous work has looked at taking advantage of parallelism in this manner by allowing the programmer to indicate a relaxing of the requirement that the output must be identical to that obtained by running sequentially. For instance, the Galois infrastructure [25, 5] uses the notion of *unordered-set iterators* to express a loop in which any sequential ordering of the iterations produces an acceptable output.

5.2.1.3 security_sha

This benchmark implements the SHA-1 hashing algorithm and is designed to take a large chunk of data, such as a file, and create a 160-bit digest. The most significant loops, A and B, are shown in listing 5.3. Loop B reads chunks of input data from a file and calls `sha_update` on each chunk to update the output digest. Loop A, in turn, breaks these chunks into 64-byte chunks which are used as input to the next round of the hash. Since each iteration of each of these loops reads the output digest, performs some transformations and writes the result back to the digest, the loop contains definite loop-carried dependences and is not parallelisable. However, the dataflow model can still achieve a small speedup for loop A. Function `sha_transform` does some transformation of the input chunk before reading the digest, updating it and writing it back. Since the transformation of the input can be overlapped with the digest update of the previous iteration, a small speedup can be achieved.

5.2.1.4 automotive_bitcount

This benchmark consists of a number of different methods of counting bits in a chunk of memory. Loop C experiences a large speedup with the dataflow model. This is a very straightforward loop which runs a particular bit-counting algorithm on a number of different inputs. The source code for this loop is shown in listing 5.4.

Variables `n`, `j` and `seed` do not cause dependences since these can be calculated locally within each thread. For example, rather than communicating `j` from one thread to the next, the compiler creates a local copy of `j` in each thread and increments it by the total number of threads on each iteration. However, despite not having any real loop-carried dependences, the benchmark does not achieve linear speedup with HELIX. To explain this, it is useful to study the speedup of this loop for various numbers of cores, as shown in figure 5.6. The performance of HELIX is comparable to the dataflow model up until 8 cores, but then it plateaus. This is caused by the HELIX loop prologue. Every HELIX loop contains at least one sequential segment which executes the minimum amount of code to determine whether or not the next iteration should execute. In this case, the prologue must calculate the new value of `j` and check the value of the condition `j < iterations`. The loop body is sufficiently short that this prologue forms a non-negligible part of each iteration and it must be executed sequentially. In addition, there is some communication overhead in synchronising the prologue to ensure it runs in order. By Amdahl's law, this implies a limit on the maximum speedup possible. For example, if the test accounts for 15% of the total loop body, the maximum possible speedup is 6.67.

Listing 5.3: Loops G and H from security_sha.

```
void sha_stream(){
    BYTE data[BLOCK_SIZE];
    while ((i = fread(data, 1, BLOCK_SIZE, fin)) > 0) { /* Loop B */
        sha_update(sha_info, data, i);
    }
}

void sha_update(){
    while (count >= SHA_BLOCKSIZE) { /* Loop A */
        memcpy(sha_info->data, buffer, SHA_BLOCKSIZE);
        sha_transform(sha_info);
        buffer += SHA_BLOCKSIZE;
        count -= SHA_BLOCKSIZE;
    }
}

void sha_transform(){
    int i;
    LONG temp, A, B, C, D, E, W[80];

    for (i = 0; i < 16; ++i) {
        W[i] = sha_info->data[i];
    }
    for (i = 16; i < 80; ++i) {
        W[i] = W[i-3] ^ W[i-8] ^ W[i-14] ^ W[i-16];
    }

    /* Abbreviated: read digest */

    /* Abbreviated: update digest */

    /* Abbreviated: write digest */
}
```

Listing 5.4: Loop C from automotive_bitcount.

```
for (j = n = 0, seed = 1; j < iterations; j++, seed += 13)
    n += pBitCntFunc[i](seed);
```

HELIX could achieve comparable performance to the dataflow model with purely static parallelisation in this case by eliminating the loop prologue. This is possible because for this loop each thread could determine independently if its next iteration should run. HELIX does not currently implement this optimisation. The results of the dataflow model are useful, not just as a limit study, but also to automatically detect scenarios where HELIX is underperforming. This is helpful for directing future enhancements to the compiler.

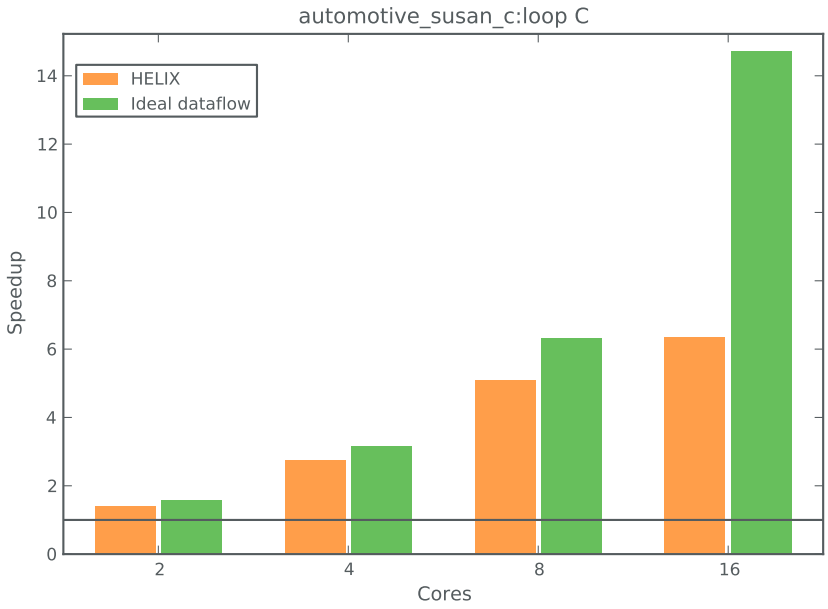

Figure 5.6: Loop C from automotive_bitcount with ideal dataflow for various numbers of cores.

5.2.2 Load balancing

One particular limitation of the ideal dataflow model is that the HELIX constraint of in-order iteration start is still enforced, i.e. iteration $i + 1$ cannot start until iteration i has started. As a result there may be performance degradation due to load imbalance when cores are stalled waiting for the completion of a particularly long-running iteration (see figure 5.7). It was observed in section 5.2.1 that some benchmarks such as automotive_susan_c do have significant imbalance in the amount of work performed by each iteration. In this section I will show how much performance degradation is caused by this limitation.

I implemented an extension to the dataflow model which removes the restriction that an iteration can only start once the previous iteration has started (see figure 5.7). This prevents long-running iterations from causing other cores to stall. Figure 5.8 compares the performance of the original dataflow model (*in-order iteration start*, IOIS) with the new model (*out-of-order iteration start*, OoOIS). On the whole, it does not appear that forcing iterations to begin in order is a major performance limitation for the benchmarks being studied. For automotive_susan_c and automotive_susan_e there are several loops which benefit from OoOIS. As discussed in the case studies, these are loops which operate on pixels in an image and the amount of work done varies depending on the value at a

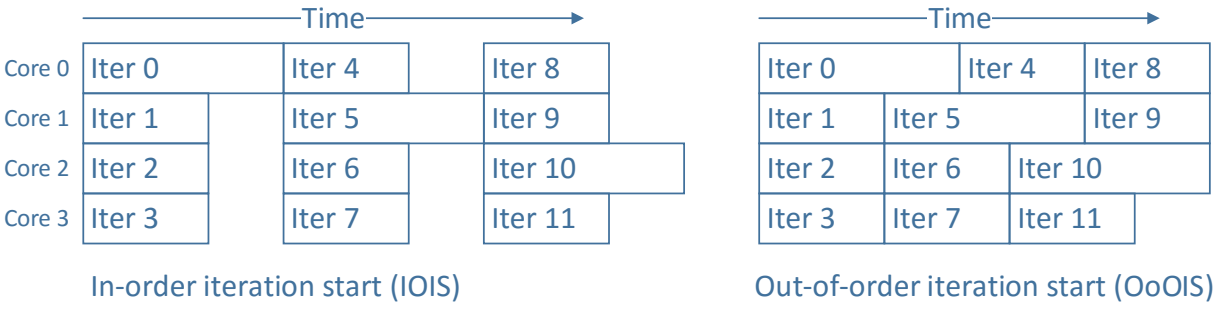

Figure 5.7: Out-of-order iteration start is beneficial for unbalanced iterations.

particular site. Loop A in automotive_susan_c for instance does not perform any work at most pixel sites so permitting OoOIS allows each core to progress quickly to a pixel where some work is actually performed.

Figure 5.9 shows the performance of loop A in automotive_susan_c in more detail. This figure shows the results of running the benchmark with the original 264KB input image and also a larger 6MB input image with wider rows. The 6MB image is noisier, resulting in more corners being detected in the initial response image. Firstly, looking at results for just the IOIS model, it is evident that the speedup is much better for the large image with 2 cores but for 16 cores the performance is virtually the same. This is because the noisier image makes it more likely for corners to be suggested in adjacent pixels, whereas in the quiet image corners are spaced out and cannot be processed concurrently.

Secondly, the OoOIS performance increases dramatically for the large image relative to the IOIS performance as core count is increased. As described in section 5.2.1.1, the number of pixels for which any significant work must be performed is relatively small in the smaller image, so performance cannot scale beyond 8x. For the larger image there is considerably more work to do in each image row but the IOIS model does not scale well. This demonstrates that the importance of load balancing becomes more critical for these benchmarks as input size and core count are increased.

5.3 Patterns and Statistics

So far in this chapter I have shown that there is a significant amount of extra parallelism available in cbench which is not being exploited by plain HELIX. The dataflow model can be used to go a step further towards understanding the nature of this parallelism by collecting various statistics as the program executes. By understanding the behaviour of dependences at runtime it will be possible to exploit that behaviour in a real system.

5.3.1 Sequential segment conflicts

The HELIX paralleliser identifies dependences statically and creates sequential segments which ensure that these dependences are satisfied. However, the HELIX algorithm must be conservative so that any dependences which may occur are synchronised at runtime. As a result, it is likely that some sequential segments may not be needed at all or may only be needed on certain iterations. This section looks at the behaviour exhibited by sequential segments at runtime to get a better understanding of how the impact of this conservativeness can be reduced while still taking advantage of the analysis that HELIX

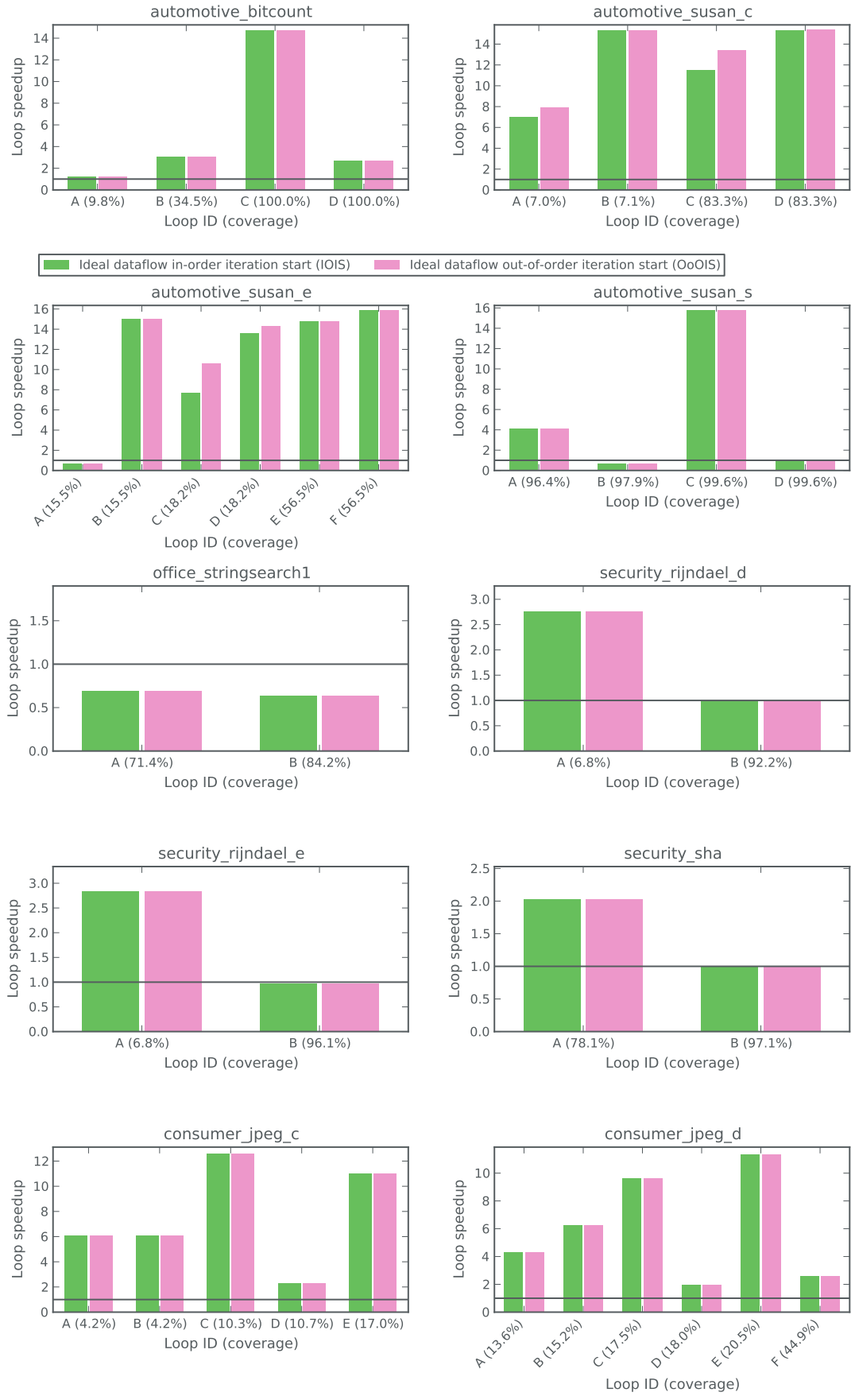

Figure 5.8: Dataflow model with out-of-order iteration start.

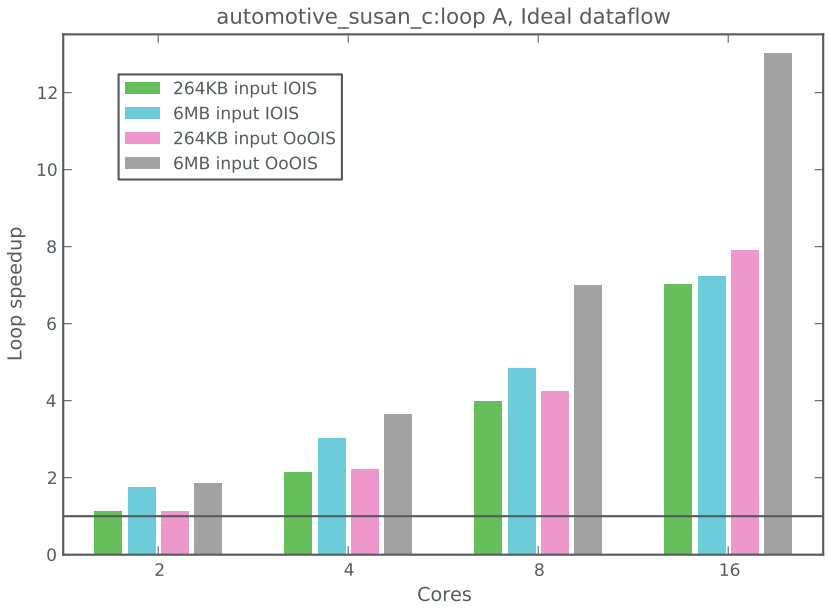

Figure 5.9: Loop A with original 264KB input and large 6MB input.

has performed.

To study sequential segment conflicts, the timing model is run with all memory accesses instrumented, and read and write sets are recorded for each sequential segment. Each time a sequential segment finishes, the model compares all the addresses touched in this segment with all the addresses touched by the corresponding sequential segment in each of the $N-1$ previous iterations, where N is the number of cores. If at least one address conflicts then it would have been necessary to synchronise that sequential segment.

In the following sections I will look at the sequential segment statistics for some of the cbench benchmarks. The tables show the sequential segment ID, the total number of iterations that executed for a loop and the number of iterations on which conflicts occurred. These results were obtained by running the timing model with 16 cores. A conflict is recorded for a sequential segment if any address touched during a particular iteration was also touched by any of the previous 15 iterations in the same sequential segment. Each sequential segment executes once per loop iteration. Sequential segment ID 0 is always the loop prologue (i.e. that section of the loop which must be executed to determine if the loop has reached its exit condition).

5.3.1.1 automotive_susan_c

Table 5.1 shows the conflict rates for the sequential segments in automotive_susan_c. Each loop in this benchmark contains only a single sequential segment, the loop prologue. The size of the prologue itself varies greatly. In loops C and D, the prologue accounts for only a small proportion of the loop body, whereas in loops A and B the prologue takes up the vast majority of the loop. This is due to HELIX performing sequential segment *merging* to reduce the number of signals which must be communicated at runtime. Looking at the HELIX performance in Figure 5.3 it is evident that the different sequential segment size results in vastly different performance for these two pairs of loops.

As described in section 5.2.1.1, loops A and B iterate across an image finding local

Loop ID	SS ID	Iterations	Conflicts	Conflicts %
A	0	203351280	28934	0.0%
B	0	344862	115736	33.6%
C	0	203351280	0	0.0%
D	0	344862	344080	99.8%

Table 5.1: Sequential segment statistics for automotive_susan_c

Loop ID	SS ID	Iterations	Conflicts	Conflicts %
A	0	1884245974	0	0.0%
B	0	211680000	0	0.0%
	1	198450000	185220000	93.3%
	2	198450000	185220000	93.3%
	3	198450000	185220000	93.3%
C	0	13252050	0	0.0%
D	0	22099	0	0.0%
	1	22050	22001	99.8%
	2	22050	0	0.0%
	3	22050	0	0.0%
	4	22050	22001	99.8%

Table 5.2: Sequential segment statistics for automotive_susan_s

maxima. Each iteration of the inner loop, A, accounts for a single pixel whereas each iteration of the outer loop, B, accounts for an entire row of the image. Whenever a corner is found, a shared array is modified and this causes a conflict. Because the outer loop accounts for a much larger portion of the image, it is more likely that any given row will contain a corner and this results in a much higher conflict rate for loop B. It is interesting that, despite having a much higher conflict rate, loop B actually performs better than A in the dataflow model. This is due to iterations of the outer loop becoming staggered over time such that they naturally begin to process different sections of the image.

5.3.1.2 automotive_susan_s

Table 5.2 shows the sequential segment statistics for automotive_susan_s. This benchmark performs smoothing on an image by passing a window across the image and calculating new values at each site based on a gaussian function. The outer loops, C and D, iterate across the image and the inner loops, A and B, iterate within the window. Of the inner loops, loop A has no conflicts while loop B has a large number of conflicts. This is reflected in the results in Figure 5.3 where A shows some speedup but not B.

5.3.1.3 security_rijndael_d

Table 5.3 shows the sequential segment statistics for security_rijndael_d. This program performs AES decryption on a piece of encrypted input text. The source code for the kernel of this benchmark is shown in Listing 5.5. The kernel consists of two loops, the outer loop, B, which reads a chunk of the input text and performs the decryption and the inner loop, A, which exclusive-ors each input block with the previous block.

Listing 5.5: Loops A and B from security_rijndael_d.

```
while(1) { /* Loop B */
    i = fread(bp1, 1, 16, fin);     /* read next encrypted block    */
                                    /* to first input buffer        */
    if(i != 16)          /* no more bytes in input - the decrypted   */
        break;           /* partial final buffer needs to be output  */

    /* if a block has been read the previous block must have been    */
    /* full length so we can now write it out                       */
    if(fwrite(outbuf + 16 - l, 1, l, fout) != (unsigned long)l) {
        printf(                                      , ofn);
        return -11;
    }

    decrypt((const byte*)bp1, (byte*)outbuf, ctx); /* decrypt the new input block */
    for(i = 0; i < 16; ++i)  /* Loop A */
        outbuf[i] ^= bp2[i]; /* xor it with previous input block */

    /* set byte count to 16 and swap buffer pointers                */
    l = i; tp = bp1, bp1 = bp2, bp2 = tp;
}
```

Loop ID	SS ID	Iterations	Conflicts	Conflicts %
A	0	541700393	0	0.0%
B	0	31993736	31864729	99.6%
	1	31864729	31735722	99.6%

Table 5.3: Sequential segment statistics for security_rijndael_d

This outer loop contains two sequential segments. SS ID 0 (the prologue) performs the `fread` library call and checks if data is available. SS ID 1 covers the entire remainder of the loop body. This loop is inherently sequential since the decryption of each block depends on the decrypted value of the previous block. This is reflected by the statistics in the table which show that both sequential segments conflict on almost every iteration. The reason there is not a conflict on 100% of the iterations is that each time the loop is invoked, the first iteration will have no previous iterations to conflict with.

The inner loop, A, is trivially parallelisable and contains no conflicts. Figure 5.3 showed a small speedup for this loop with the dataflow model which is due to the removal of the requirement to sequentialise the loop prologue.

5.4 Summary

In this chapter I have shown that there is extra parallelism that cannot be exploited by the oracle DDG and that it is necessary to take advantage of the dynamic behaviour of the program to gain further speedups. I demonstrated an ideal dataflow model which was used place an upper limit on the speedup achievable with a cyclic-multithreading parallelisation model. I showed that the HELIX-imposed constraint of starting loop iterations in order does not significantly reduce performance. Finally I presented some statistics for sequential segment conflicts. While this has shown that extra parallelism is available, the dataflow model is not considered to be a realistic approach to employ on a real machine without adding substantial hardware support for the hypothetical implementation described in section 5.1.1. In the next chapter I will propose some realistic techniques to show how much of this parallelism can be exploited in practice.

Chapter 6

Practical speculative parallelism

In chapter 4 it was shown that improving static dependence analysis does not enhance parallel performance for a set of embedded benchmarks. However, chapter 5 proved that a significant amount of further parallelism is available in these benchmarks if the dynamic behaviour of the program can be exploited. This chapter discusses practical techniques for extracting this parallelism and shows what performance can be expected from a realistic system.

Various approaches are considered to run the HELIX code speculatively, all using a conflict resolution scheme based on transactional memory. I will explore the various trade-offs involved in speculative execution and discuss the characteristics required of a thread-level speculation (TLS) implementation to achieve superior performance.

Speculation is not always profitable. Loops with frequent dependences are unlikely to benefit from speculation due to the overhead of the speculation support and the costs of rollback and re-execution. Ultimately the key is finding the balance between HELIX-style static parallelisation and dynamic speculative execution and identifying which technique to use in which situation.

To evaluate the various schemes, I have implemented a number of HELIX timing models which simulate how execution would proceed on a real system. Using the timing models allows evaluation of a wide range of implementations and parameters. In addition it allows easy exploration of the complete design space of TLS without the excessive engineering overhead of providing multiple fully-functional implementations.

This chapter begins with a discussion of the overlapping features of TLS and transactional memory (TM) and how to leverage previous exploration of the trade-offs in TM to improve our understanding of the TLS design space (section 6.1). This motivates the extension of the HELIX timing model to include simulation of TM in section 6.2, where a parameterisation of various TM implementations is presented. Section 6.3 describes a pure speculation model which discards all the HELIX generated synchronisations and relies solely on speculation to preserve program correctness. This model is further refined in section 6.4 by using profile data to determine whether it is sensible to speculate or not. Finally I present a study of the sizes of transactions using my scheme and show that current proposals for hardware TM support would comfortably accommodate this scheme (section 6.5).

6.1 Supporting speculation with transactions

Transactional memory (TM) and thread-level speculation (TLS) have many common features and so speculation implementations are commonly based on transactional memory algorithms or even incorporate existing TM implementations. Essentially a TM system is a means of sandboxing portions of execution which run in parallel such that their memory reference conflicts are resolved in a well-defined manner. The TM ensures that the ultimate result of the complete execution is equivalent to some sequential execution schedule where all the portions are executed in some order one after the other.

Broadly speaking, a TM implementation will have the following features:

- Conflict checking system which identifies memory addresses touched by multiple transactions.
- Rollback capability to erase the effects of invalid computation.

TLS also requires these features so there is significant overlap between the two concepts. TLS generally has an additional restriction that the equivalent sequential execution schedule must have some specific ordering. In HELIX-style parallelisation the portions that need to be sandboxed are iterations and it is strictly required that the effects of each iteration appear to happen in the order prescribed by the original sequential loop (i.e. loop-iteration order).

6.1.1 Design decisions

The TM design space is large and a number of design decisions must be made to create an efficient implementation for TLS. In some cases, the specialised nature of loop-iteration-based speculation enforces a particular decision. In this section I will describe some of the major decisions that need to be made.

Deferred versus direct update The most important factor affecting the performance and implementation of the TM system will be the manner in which transactional writes are propagated to non-transactional memory state. This decision dictates what happens when a transaction performs a write. A deferred update system will implement a write buffer which records all writes performed during the transaction. At commit-time the transaction must transfer all the values in the write buffer to the main memory store.

In contrast, a direct update system modifies the memory in place. This has the advantage that it does not need to calculate the appropriate buffer location on each write and does not incur the overhead of having to flush the buffer to memory on commit. However, a direct update system must implement some system for logging modifications such that they can be rolled back if a transaction fails.

In general, the trade-off is that deferred update systems can handle rollbacks more efficiently by simply invalidating the write buffer, while direct update systems can handle commit more efficiently since the memory has already been updated. There is an additional complication in implementing direct update for TLS because of the strict loop-iteration ordering of transactions. When a younger transaction writes to an address and then an older transaction reads the address, the older transaction must take precedence so the younger transaction will be aborted. In a general purpose TM, the ordering of transactions is irrelevant and this scenario would not necessarily result in rollbacks. HTM

systems are more commonly based on the deferred update model using a private cache as the write buffer, which allows efficient conflict detection and commit. For these reasons I have used the deferred update model for all the experiments in this chapter.

Lazy versus eager conflict checking Deciding at what point in the lifetime of the transaction the system should attempt to detect conflicts is of significant consequence. In general there are two broad possibilities: detect conflicts each time the transaction performs a memory access (eager) or detect conflicts at the end of the transaction just before it commits (lazy).

The advantage of eager conflict checking is that the system can react to conflicts as soon as they occur, thus avoiding having the transaction continue executing even though it is destined to abort. However this results in a much larger overhead on each access. In addition, for TLS implementation it would also be necessary to perform all the checks again at commit-time to ensure an older transaction has not written one of the addresses in the intervening period. As such I have largely focussed on lazy conflict checking to keep the overhead of performing transactional memory accesses to a minimum.

Weak versus strong isolation A crucial decision for general purpose TM implementation is how to deal with non-transactional memory accesses executing concurrently with transactional memory accesses. In a weak isolation system, code executing outside a transaction is not modified and conflicts with a running transaction will not be detected. It is the programmer's responsibility to ensure such conflicts do not occur. Strong isolation implementations transform all memory accesses even outside transactions such that a transaction that conflicts with non-transactional code will be aborted.

The pure speculation model (section 6.3) runs all loop code transactionally so isolation is not an issue. The judicious speculation model (section 6.4) does allow concurrent execution of speculative and non-speculative code, however, the compiler analysis ensures that conflicts cannot occur between transactional and non-transactional code so the weak/strong isolation distinction is not applicable. Parallel code can only run outside a transaction if it has been proven by the compiler to be safe.

Contention management Contention management determines what happens when a conflict is detected between two transactions and must be resolved. In general this is largely concerned with determining whether a transaction needs to be rolled back and deciding which transaction must be aborted. This decision is made simple for TLS by the restriction that transactions must commit in loop iteration order. Therefore when a conflict arises, the younger transaction is always the one to be aborted.

Granularity The decision as to what granularity conflicts should be detected at can be a key factor in the performance of the system. Large granularities (e.g. cache line size) allow for more efficient data structures and can result in quicker conflict checking if multiple memory references within a transaction are stored in the same bin. However, false conflicts are more likely. Small granularities (e.g. byte or word size) increase overhead but result in more accurate conflict detection. Some systems operate at the granularity of objects in an object-oriented program [34]. This has the advantage of tailoring the conflict detection granularity to the nature of the program in question; however, it is not applicable to general purpose programs written in C and I will therefore not consider this

option. I have used word-sized conflict detection granularity since it is not desirable that loop iterations which operate on adjacent array indices should be recorded as a single transactional object. This may still present problems for byte arrays however.

6.2 Speculation timing model

In section 3.5 I described the HELIX timing model which estimates the parallel speedups which can be achieved by running an instrumented sequential version of the program. The timing model is independent of specific architectural features and provides an upper bound on the potential of the execution styles described in this chapter. However the model was shown to accurately predict speedups for a range of loops in cbench on real machines in section 3.5.2. Now the timing model is enhanced with a TM-like extension which can be used to estimate the speedups obtainable with speculation. The extension operates in much the same manner as described previously for the HELIX timing model, by counting the clock cycles elapsed on a number of virtual processors in response to events which occur during execution of the program. In addition to the events recorded in the original model, it is now also necessary to record all memory accesses to facilitate accurate simulation of the TM.

Loops are parallelised one at a time as this is most conducive to understanding the behaviour of the code under speculative execution. The HELIX transformations are applied to the loop to produce a parallelised version. Every memory access to a non-local variable is then instrumented with a callback to the timing model, passing the address which is read or written. HELIX converts loop-carried register and local variable dependences into non-local memory accesses so these can be tracked by the TM (see section 3.2.4 for more details). As part of the standard HELIX optimisations, variables which can be privatised and loop induction variables are converted into local variables and these are not tracked by the TM since they cannot cause dependences.[1]

6.2.1 Model implementation

The initial implementation of a timing model to enhance HELIX performance uses a simple writeback TM with lazy conflict checking. The loop is parallelised using the standard HELIX algorithm but all sequential segment ordering constraints are ignored. Instead, the entire iteration, including the loop prologue, is run using a TM implementation to ensure that data dependence conflicts are respected.

This model is based on a deferred update TM. Speculative writes which occur in the loop are buffered by the transaction and do not become visible to other threads until after the iteration has completed and the transaction has been validated. A hash table is created for each transaction to store speculative writes to allow efficient querying for the existence of addresses. This is called the *write set*.

In addition to the write set, a record of all addresses which were read during the transaction must be maintained, along with the times at which 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 iteration has completed

[1] An overview of these standard parallelisation optimisations is provided in section 2.1.1.

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 contained and the conflicting transaction committed after the address was read in the current transaction, the current transaction must be rolled back and re-executed.

6.2.2 TM implementation

The TM design space is large and I was faced with deciding between many potential implementation styles. Therefore, to the greatest extent possible, I have parameterised the model to facilitate the exploration of the design space. Of crucial importance is accurately modelling the overheads which are caused by the addition of TM instrumentation. From my experience of implementing TM, from studying existing TM schemes [90] and from consulting with industry partners [91], I have determined that overheads can be categorised according to six main sources:

- **Transaction start:** When the transaction starts there is at least 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, the implementation may have some overhead to set up data structures.
- **Transactional load:** When a transaction performs a load it will incur 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 will incur overhead from adding the address to a write set and/or buffering the new value.
- **Validate reads:** When a transaction attempts to commit, it will generally perform some verification of 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.
- **Commit writes:** 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.

The parameterisation is primarily modelled on a deferred update system. It would be possible to model a direct update system with some minor modifications and changes to the parameter values. However the challenges of using direct update for speculation have already been discussed (section 6.1.1), and consequently, I have chosen to focus on a deferred update implementation. These overheads could be further categorised to increase accuracy for specific TM designs. For instance, the overhead of a load may be different if it is to a location which has previously been read in the same transaction. However, from discussion with industry partners [91] and from the previous use of a similar parameterisation by Olukotun et al. [92], I have concluded that these parameters are sufficient to provide adequate estimates of performance with speculation.

6.2.3 Determining parameter values

The main advantage of parameterising the design is that it allows estimation of the performance of speculation under various TM designs and observation of the maximum overheads which can be borne while still achieving speedup. In light of this, I have done studies to determine the appropriate parameter values to model various styles of TM. This section discusses available TM implementations and how TM can be best designed to suit TLS. Three TM models are considered:

1. **TinySTM: An existing general purpose STM implementation [90].**
2. **TLS-STM: An STM design currently being implemented which is optimised for TLS.**
3. **TCC-HTM: A HTM model described by Olukotun et al. [92].**

The overheads I determined are summarised in table 6.2 on page 93.

6.2.3.1 TinySTM

TinySTM [90] is an open source software TM implementation which supports various TM designs. TinySTM is word-based which makes it suitable for parallelising C programs where it is necessary to track memory references based on runtime addresses. TinySTM tracks memory references through a shared array of locks. When a memory access is executed transactionally, the address is hashed to give an offset into the array. The value of each array element either stores the current version number of the lock or the address of the transaction which currently holds the lock if the lock is taken.

TinySTM has three designs which can be chosen at compile-time:

1. **Write-back encounter-time-locking:** A deferred update design where the transaction takes a lock on a location as soon as that location is written.
2. **Write-back commit-time-locking:** A deferred update design where the transaction only takes locks on written locations at commit-time.
3. **Write-through:** A direct update design.

I primarily considered the write-back commit-time-locking (WB-CTL) design since direct update designs introduce complications for TLS (as described previously) and taking locks at commit-time most closely followed my own TM implementation which will be described later.

The overheads of the STM are characterised by looking at a sample application: conservative smoothing [93]. This is an image-processing algorithm which removes noise spikes from an image, i.e. isolated pixels which have much higher or lower intensity than the surrounding pixels. The kernel of the application is shown in listing 6.1. For this experiment the outer loop is parallelised, i.e. processing rows of the image in parallel. This is an interesting loop for TLS since, if noise is relatively rare, the rows can generally be processed independently. The inner loop would also be a candidate for speculation but each iteration would be very short and the fixed overheads of ordering these iterations would outweigh the benefit of speculation. This kernel was chosen because it is fairly

Listing 6.1: Kernel of conservative smoothing image filter.

```
for(int i = 1; i < y_size-1; i++){

  int v00, v01, v02, v10, v11, v12, v20, v21, v22;
  v01 = R[i-1][1];    v02 = R[i-1][2];
  v11 = R[i  ][1];    v12 = R[i  ][2];
  v21 = R[i+1][1];    v22 = R[i+1][2];

  for(int j = 1; j < x_size-1; j++){
      v00 = v01;      v01 = v02;  //
      v10 = v11;      v11 = v12;  //
      v20 = v21;      v21 = v22;  //
      v02 = R[i-1][j+1];          // Shift window to the right
      v12 = R[i  ][j+1];          //
      v22 = R[i+1][j+1];          //

      int local_max = 0;
      if(v00 > local_max) local_max = v00;   //
      if(v01 > local_max) local_max = v01;   //
      if(v02 > local_max) local_max = v02;   //
      if(v10 > local_max) local_max = v10;   //
      if(v12 > local_max) local_max = v12;   // Find local maximum
      if(v20 > local_max) local_max = v20;   //
      if(v21 > local_max) local_max = v21;   //
      if(v22 > local_max) local_max = v22;   //
      if(v11 > local_max*MAX_GRAD)
        R[i][j] =  local_max;

      int local_min = 0;
      if(v00 < local_min) local_min = v00;   //
      if(v01 < local_min) local_min = v01;   //
      if(v02 < local_min) local_min = v02;   //
      if(v10 < local_min) local_min = v10;   //
      if(v12 < local_min) local_min = v12;   // Find local minimum
      if(v20 < local_min) local_min = v20;   //
      if(v21 < local_min) local_min = v21;   //
      if(v22 < local_min) local_min = v22;   //
      if(v11*MAX_GRAD < local_min)
        R[i][j] = local_min;
  }
}
```

small and easy to instrument, it exhibited behaviour which could be exploited with TLS, it has a high density of memory accesses which exposes the overhead of the TM system and it allowed the addition of extra memory accesses easily which facilitated obtaining more robust results.

This kernel was manually instrumented with calls to TinySTM to study the performance that could be achieved by parallelising the loop with TLS. In addition to instrumenting all loads and stores in the program, it is necessary to add support for in-order commit[2]. The modifications are shown in listing 6.2. The outer loop is modified such that each thread processes a row of the image in turn. `sigsetjmp` is called to save the environment at the point where the transaction begins. TinySTM uses `siglongjmp` to return to this point if the transaction aborts and re-execution is necessary. Every access to the array `R` is converted to a call to `stm_load` or `stm_store`. To enforce in-order commit of transactions (necessary to preserve the sequential semantics of the original loop) a shared variable, `global_commit_stamp`, is added along with a thread-local variable, `txn_stamp`. A transaction may only commit when its `txn_stamp` is equal to the `global_commit_stamp`. The `txn_stamp` is essentially an induction variable and can be calculated locally by each thread. Variables `v00`, `v01` etc. are all written first in

Listing 6.2: Conservative smoothing following manual instrumentation to support TLS parallelisation of the outer loop.

```
/* This code runs on each processor */
int txn_stamp = thread_id;
for(int i = thread_id+1; i < y_size-1; i+=NTHREADS){

    sigjmp_buf e = stm_start((stm_tx_attr_t)0);
    sigsetjmp(*e,0); // Save environment for re-execution

    int v00, v01, v02, v10, v11, v12, v20, v21, v22;
    v01 = stm_load(&R[i-1][1]);    v02 = stm_load(&R[i-1][2]);
    v11 = stm_load(&R[i  ][1]);    v12 = stm_load(&R[i  ][2]);
    v21 = stm_load(&R[i+1][1]);    v22 = stm_load(&R[i+1][2]);

    for(int j = 1; j < x_size-1; j++){
        v00 = v01;     v01 = v02;
        v10 = v11;     v11 = v12;
        v20 = v21;     v21 = v22;
        v02 = stm_load(&R[i-1][j+1]);
        v12 = stm_load(&R[i  ][j+1]);
        v22 = stm_load(&R[i+1][j+1]);

        int local_max = 0;
        /* find local_max ... */
        if(v11 > local_max*MAX_GRAD)
            stm_store(&R[i][j], local_max);

        int local_min = 0;
        /* find local_min ... */
        if(v11*MAX_GRAD < local_min)
            stm_store(&R[i][j], local_min);
    }

    while(global_order_stamp != txn_stamp); // Wait for turn
    stm_commit();
    global_order_stamp++; // Allow next iteration to commit
    txn_stamp += NTHREADS;
}
```

each iteration of the outer loop and can therefore be privatised to each thread.

Determining the overheads of the various TM operations was non-trivial since the operations generally run for a very short period of time (<100 clock cycles). Most methods to measure cpu usage (`clock`, `clock_gettime` etc.) have a relatively large overhead at that scale and it is difficult to distinguish the overhead of the TM from the overhead of the measurement instrumentation. I experimented with the possibility of timing a whole iteration and then progressively removing calls to the STM so that the reduction in overall execution time would be equal to the overhead of the removed calls. This produced inconsistent results and does not seem to be a viable method since removing calls affects the overhead of other calls, e.g. removing the calls to `stm_store` reduces the overhead of `stm_commit` since there are now no buffered stores to be written out during commit.

Ultimately I settled on using the x86 `RDTSC` instruction which accesses the processor's timestamp counter, a 64-bit register which counts the number of clock cycles elapsed since the processor reset. I measured the overhead of `RDTSC` to be 24 cycles on our test machines (a 4-core Haswell server and an Ivy Bridge 16-core server). An example of how the code was instrumented is shown in listing 6.3. For loads and stores, rather than timing a single

[2]The most recent version of TinySTM includes support for in-order commit of transactions but this is only available in the encounter-time-locking design.

Listing 6.3: An example of how RDTSC was used to measure overheads.

```
unsigned long long start_clock = rdtsc();
v11 = stm_load(R[i][j]); // Actual load
tmp = stm_load(A[0]);    //
tmp = stm_load(A[1]);    //
tmp = stm_load(A[2]);    //
tmp = stm_load(A[3]);    // Extra loads inserted to reduce variability of results.
tmp = stm_load(A[4]);    // 'A' is a stack-allocated array and will not
tmp = stm_load(A[5]);    // cause inter-thread conflicts.
tmp = stm_load(A[6]);    //
tmp = stm_load(A[7]);    //
tmp = stm_load(A[8]);    //
unsigned long long end_clock = rdtsc();
totalclocks += end_clock - start_clock - 24 /* RDTSC overhead */;
totalloads += 10;
```

call, I added a number of extra calls to the same TM operation. For example, for loads I added a number of extra calls to `stm_load` for different addresses and then recorded the time for all loads to complete. This gave more repeatable results than timing a single operation.

Additional overhead fluctuations may occur since TinySTM uses an array of locks to manage contention between transactions. If, for example, a load attempts to read the current version of a location from the lock table but the lock is currently held by a committing transaction, the load will spin wait for the lock to come free. I was primarily looking at low contention cases — high contention would not be suitable for TLS anyway — so this variability is unlikely to significantly affect performance.

Start TinySTM uses `sigsetjmp` to save the environment at the point where the transaction begins. In addition, a call to `stm_start` is made to initialise variables such as the starting timestamp.

Loads In the WB-CTL design of TinySTM it is not possible to accurately model the overhead of a load as a single number as there is a dependence on the size of the write set. When a transactional load is executed, the TM searches the write set to see if the location has previously been written. This is so that the transaction can read its own buffered writes. The TinySTM write set is simply an array recording the address and value of each write. Searching for matches in the write set is linear complexity. Therefore two parameters must be found, the base time for adding the read to the read set, and the time taken per store to search the write set. To measure this I first recorded the time for a load in a transaction with no writes to obtain the base time. Then I gradually added stores to the transaction and measured how much the load time increased for each extra store.

Stores Stores are modelled similarly since the TM also looks up the write set for previous stores to the same address. This is to prevent duplicates in the write set. Overheads were measured using the same technique as for loads.

Validation Validation ensures that all values read transactionally are still valid. In TinySTM this is achieved by recording the version of the lock corresponding to the location

	Ivy Bridge
Processor	Xeon(R) E5-2667 v2
Frequency	3.3GHz
Sockets	2
Cores	8 per socket (hyperthreaded)
Cache	32KB + 256KB private25MB shared

Table 6.1: Specification of machine used to evaluate TinySTM performance.

when it is first read and then confirming that the lock version has not changed by the time the transaction commits. Validation is a relatively straightforward operation, simply iterating through the read set, checking each version. It is therefore linear complexity, proportional to the size of the read set. To measure the overhead I timed the entire validation for a read set and then divided by the size of the read set to give a per-read cost.

Commit Once the read set has been validated the buffered writes can be copied out to main memory. Since I am enforcing fully-serialised commit, it would now be possible to simply copy these values out with no further checks. However, since TinySTM is designed to support parallel commits it must do some extra work (superfluous in this case) to protect the memory's consistency during commit. Locks are taken out on all addresses in the write set, then each location is atomically updated with the buffered value and finally the locks are released. Commit is linear complexity in the size of the write set.

Abort Abort is essentially free for TinySTM with the WB-CTL design since no locks are taken out until commit and serial commit ensures that once the read set has been validated, the transaction will never abort. The read and write sets are stored as arrays so they can be invalidated by simply setting the array size to zero.

Validation of TinySTM speculation model

Ideally the speculation model would have been validated by modifying the compiler to insert calls to TinySTM into the parallelised code, running this modified code on a real machine and comparing the speedups to those reported by the timing model. However, implementing this in the compiler led to major technical challenges due to the difficulty of debugging parallel machine code and the instability of the current version of TinySTM. Therefore as an approximate validation of the model, I compiled the manually-parallelised version of the conservative smoothing benchmark shown in listing 6.2 with `GCC`. The performance of the parallelised version was then compared to the baseline sequential version (listing 6.1), also compiled with `GCC`. The benchmarks were timed on an Ivy Bridge dual-socket 16-core server machine. The configuration of the machine is shown in table 6.1.

The timing model was implemented to apply the overheads measured for TinySTM as shown in table 6.2. The sequential version of the benchmark was then automatically

	TinySTM on Haswell	TinySTM on Ivy Bridge	GABP-STM	TCC-like HTM
Start	120	150	95	0
Load	20 + 2*num_writes	25 + 3*num_writes	14	0
Store	25 + 2*num_writes	32 + 3*num_writes	16	0
Validate	5*num_reads	6*num_reads	3*num_reads	0
Commit	55*num_writes	66*num_writes	3*num_writes	5
Abort	0	0	3*(num_reads + num_writes)	0

Table 6.2: Overheads of various TM designs in cycles.

parallelised with HELIX and run through this model. The results for this experiment are shown in figure 6.1. The baseline for the Ivy Bridge results is the execution time when the original sequential version of the code was compiled with GCC. The baseline for the timing model results is the execution time of the sequential code produced by ILDJIT. The graph shows that the speedup trends are similar across cores between the speculation model and manually parallelised version, although the performance of the GCC-compiled version is not as good as suggested by the timing model. The reason for this is difficult to surmise since the code being compared is generated by two different compilers; however, a possible explanation is that the sequential code generated by GCC is superior to that generated by ILDJIT and so the overheads of parallelisation are relatively more significant. The machine used has 8 cores per socket and performance does not scale well above 8 cores due to the increased latency of communicating between sockets. In addition, the speculation model is not a perfect simulation of TinySTM since it does not model the shared array of locks which is used to detect conflicts and compare versions, and this may lead to unpredictable behaviour in the real implementation. Since TinySTM transactions may spin while waiting for locks to come free the overheads may be quite variable in practice. TinySTM also performs eager conflict checks for writes which may result in multiple rollbacks for a single transaction which is particularly detrimental for this benchmark, since if a conflict exists then the iteration will only be able to run successfully once the previous iteration has committed its writes.

6.2.3.2 TLS-STM

While TinySTM is a fairly simple implementation of STM, it is more general purpose than what is required for TLS. In particular, since there is an implicit ordering of transactions (dictated by iteration order) and in-order commit enforces serialisation of commits, there is no need for the STM to take out locks on locations which will be written during commit. To take advantage of these possible optimisations an alternative STM implementation is proposed, TLS-STM, which eliminates some of the overheads experienced by TinySTM. The concept of TLS-STM is the work of Kevin Zhou.

TLS-STM is also a deferred update design but with lazy conflict checking so reads are not validated in any way until commit-time. The STM uses value-based conflict detection: when an address is first read by a transaction, a copy is made of the current value at that address. Validating the read set is a simple matter of checking that the value at the address is still equal to the saved value. This has the added benefit of avoiding conflicts in the case

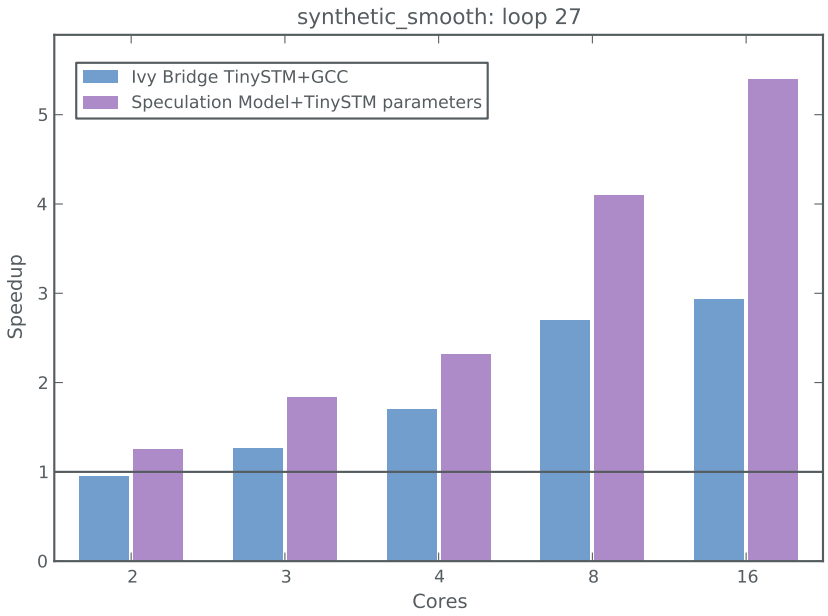

Figure 6.1: Comparison of speedups for conservative smoothing benchmark (listing 6.1). Baselines are sequential execution when compiled with GCC and sequential execution in the timing model respectively.

of silent stores (i.e. when a location is overwritten with its current value). For TLS this approach is also attractive since the oldest transaction can run without instrumentation, as any writes made will still be detected by subsequent transactions if they read a stale value. This method was inspired by JudoSTM [94], an STM implementation for dynamic binary rewriting.

TLS-STM can efficiently deal with transactions with many writes using a combination of a linear probe hash table, referred to here as the translation table, and two arrays, one for reads and one for writes. On a transactional load, the runtime address is hashed to find an entry into the translation table, as shown in figure 6.2. If this is the first time the address has been accessed, an entry is inserted into the translation table recording the address, whether the access is a read or write and the address of the next available entry in the read set. At this entry in the read set, the accessed address and the current value at that location are recorded. A transactional store is recorded similarly, but with an entry being inserted into the write set and the data stored being the updated value of that location. This implementation is not dependent on the size of the write set as in TinySTM, since detection of previous writes to the same address entails no extra cost by the translation table lookup.

Validation of the read set is of linear complexity in the size of the read set and is performed by comparing the recorded data to the current value at the access location. If the current value is not equal to that stored in the read set then the value that was read transactionally is stale and the transaction is aborted. Once the read set has been validated, the write set is committed to main memory by copying each value to its corresponding address. No fine-grain locks are required since validation and commit are performed serially and in iteration order. To abort a transaction it is necessary to clear out the translation table and read/write sets. Clearing the sets is a simple matter of setting the array sizes to zero; however, it is necessary to clear each entry from the translation table by setting

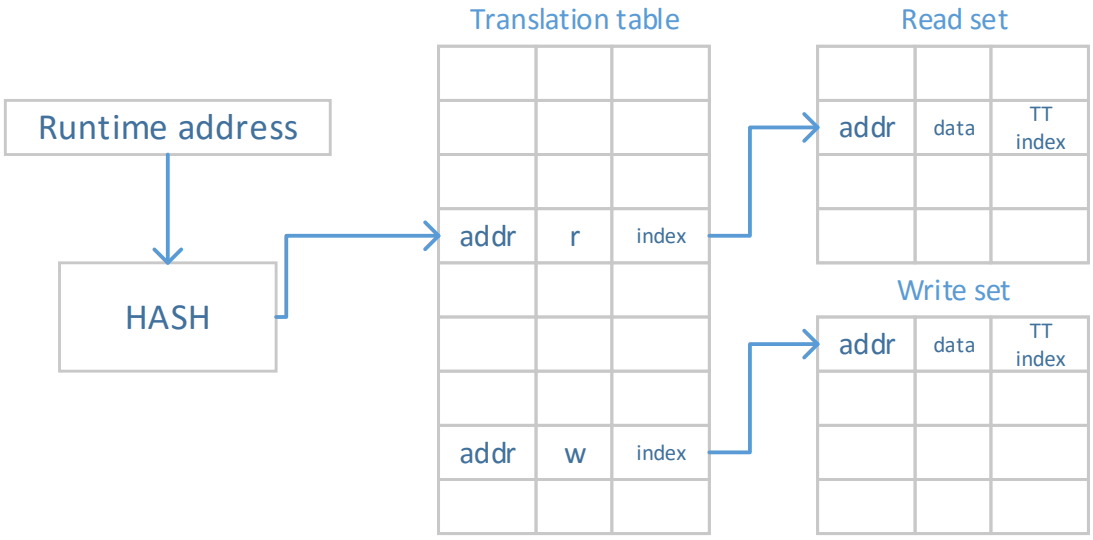

Figure 6.2: Translation of a runtime address in TLS-STM.

the recorded address to zero. To achieve this with linear complexity, each entry in the read/write sets also contains a pointer back to the corresponding translation table entry. Clearing the translation table is then linear in the size of the read and write sets.

Work on implementing this STM is currently being undertaken by Kevin Zhou. A stable version of the STM is not yet available and therefore it was not possible to show the performance of the conservative smoothing benchmark with TLS-STM. However, it was possible to estimate the overheads by counting the number of instructions required to implement each action and assuming an execution rate of one instruction per cycle. These overheads are shown in table 6.2.

6.2.3.3 Hardware TM

It is inevitable that any software implementation of TM is going to add significant overheads in terms of execution time and memory footprint. This motivates using the speculation model to also simulate the overheads of TLS with full hardware support. While there are several commercially available hardware TM implementations, none provide the full complement of advanced features which are needed to effectively support TLS such as ordered transactions, data forwarding and word-level conflict detection [63]. However, previous work in modelling hardware TM for speculation has suggested overhead parameters which can be used to model the performance of such a system. Olukotun et al. [92] describe an execution-driven simulator executing instructions at a rate of one instruction per cycle with a parameterised TM system known as TCC [29].

A TCC-based processor adds additional hardware to buffer writes, detect conflicts and order commits. Transaction control bits are added to the L1 cache to indicate that particular cache lines have been speculatively read or modified. Transactional stores are not broadcast over the coherence bus as they occur but are combined into a commit packet which is broadcast once the transaction commits. The commit packet is snooped by the other caches to detect conflicts.

In this scheme transactional loads and stores do not incur any extra overhead which is visible to the software since the hardware takes care of recording them in the cache. In addition, no overhead is incurred for validation which takes place continuously as

Listing 6.4: Loop C from automotive_bitcount.

```
for (j = n = 0, seed = 1; j < iterations; j++, seed += 13)
    n += pBitCntFunc[i](seed);
```

the caches snoop the bus for commit packets. Committing a transaction does incur a delay since the hardware must do a processor-wide arbitration for the right to commit. Olukotun et al. [92] suggest that this amounts to a delay of 5 cycles on average for a chip multiprocessor. Aborting a transaction is achieved by invalidating all transactional loads and stores in the L1 cache which can be performed in parallel by the hardware. Olukotun et al. propose that no overhead is incurred for this operation in a chip multiprocessor. These overheads are summarised in table 6.2.

6.3 Pure speculation

I use the term *pure speculation* to refer to the speculation model I have described so far, i.e. always speculating during loop execution with each transaction covering one entire iteration. The pure speculation model was run on the cbench benchmarks with the three sets of TM parameters shown in table 6.2 (for TinySTM I simulated with the Ivy Bridge figures only since there was not sufficient difference between the two machines for the Haswell parameters to be of interest). The results are shown in figure 6.3. These were obtained by first measuring the speedup for each individual loop and using the loop selection algorithm (section 3.2.5) to choose the best loops. In the case that no speedups were achieved for any loops, it was not profitable to parallelise at all so the speedup is simply unity. This is the case for office_stringsearch1 and security_rijndael_e. As can be seen from these results, it was possible to achieve some speedups using this timing model for most benchmarks. However, for some benchmark the performance of HELIX is superior to pure speculation. In this section I will look in more detail at these benchmarks to understand their performance.

6.3.1 Case studies

6.3.1.1 automotive_bitcount

Figure 6.4 shows the breakdown of loops for automotive_bitcount when run with 16 cores. The variation in performance between different loops is significant. Only loop C performs better with the speculation model than with plain HELIX. The original source of this loop can be seen in listing 6.4.

The HELIX inter-procedural dependence analysis is already capable of determining that the bitcount function does not contain loop-carried dependences, so the function itself runs in parallel resulting in a good speedup for HELIX. What limits HELIX speedup in this case is the overhead of synchronising the loop prologue. The speculation model is not restricted by enforced ordering of the prologue since the body of the loop can be executed speculatively even if the loop should have completed. This potential enhancement was previously noted for the ideal dataflow model in section 5.2.1.4. In this case, the pure speculation model with TCC HTM parameters was able to achieve comparable performance to the theoretical maximum.

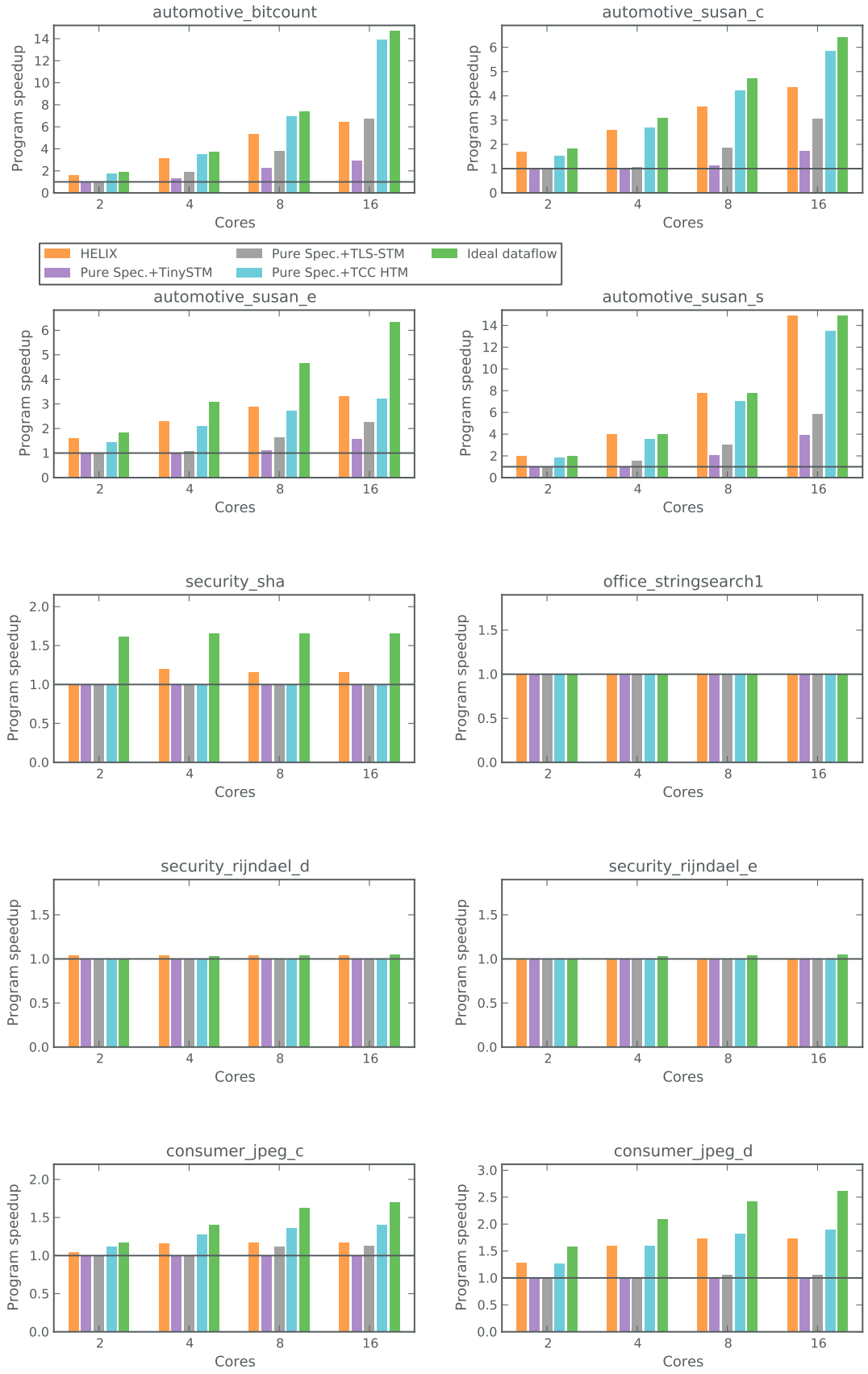

Figure 6.3: Cbench results for plain HELIX and speculation model with various TM parameters.

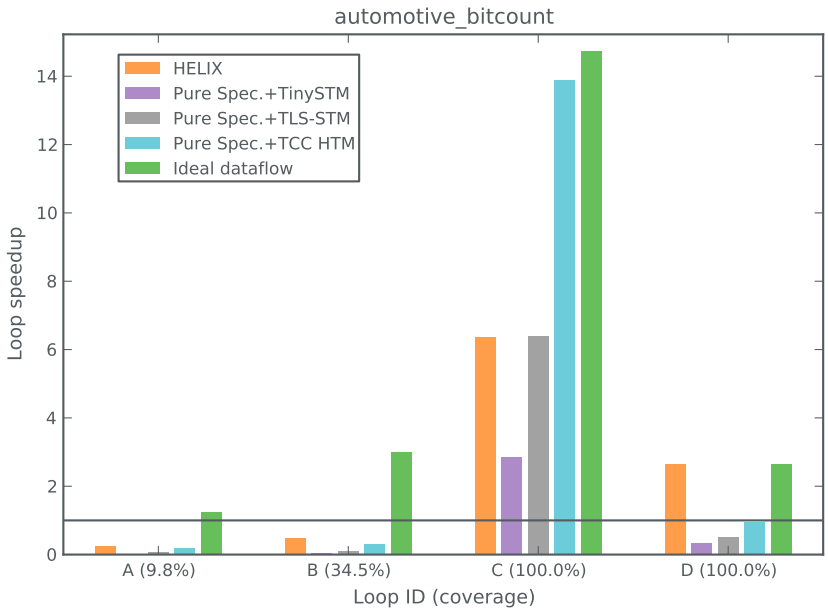

Figure 6.4: Loop breakdown for automotive_bitcount with 16 cores. Coverage indicates percentage of total execution time during which loop is running.

6.3.1.2 automotive_susan_s

Figure 6.5 shows the speedups of the individual loops in automotive_susan_s. HELIX achieves linear speedup for loop C although other loops do not perform well. In this case the speculation model does show good performance for loop C with HTM parameters compared to the sequential baseline but lags behind HELIX.

To understand the performance profile for this benchmark it is necessary to look at the code for the loop, as shown in listing 6.5. The benchmark contains a single kernel of four nested loops. The loops have been labelled in the code to correspond with the labels in figure 6.5.

The benchmark carries out noise reduction on an image by passing a mask across the image and performing Gaussian smoothing within the mask at each step. The outer loops, D and C, iterate vertically and horizontally across the image. The inner loops, B and A, iterate vertically and horizontally across the mask.

Looking at loop C, each iteration places the mask over a portion of the image, calculates a new value for the upper left hand pixel and writes it back to the image (`*out++=new_value`). In the next iteration, the mask is moved one pixel to the right so it no longer overlaps the written pixel. Each iteration of loop C is completely independent, containing no loop-carried dependences. HELIX does not generate any sequential segments and the loop can run entirely in parallel (apart from the proportionally short prologue).

Speculation performance is not as good because essentially the same code is being run, but with the addition of the overheads of instrumenting memory accesses, performing conflict checks and committing buffered writes to memory. Loop B contains a loop-carried dependence on `ip` and therefore is not a good candidate for speculation. However, loop A contains no loop-carried dependences and therefore performs well with speculation with the TCC HTM parameters. The density of memory accesses in loop A causes the STM

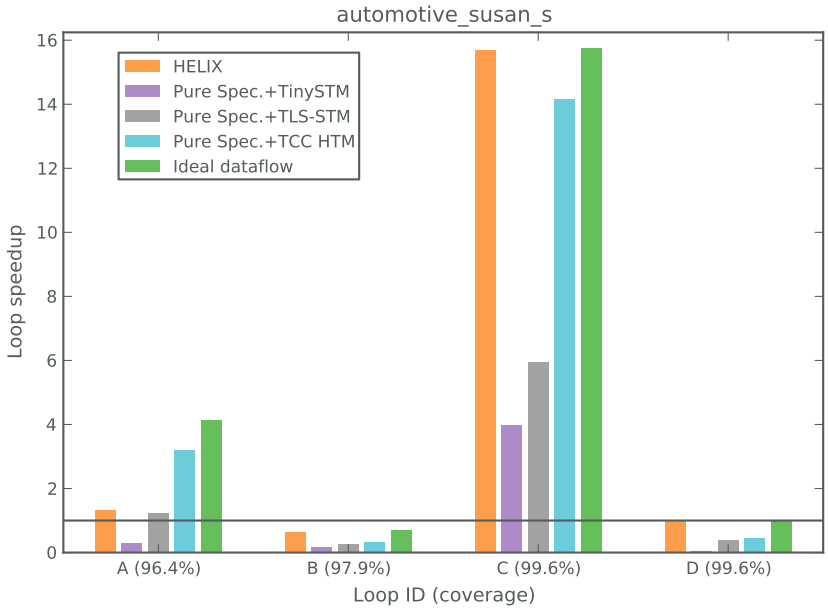

Figure 6.5: Loop breakdown for automotive_susan_s with 16 cores. Coverage indicates percentage of total execution time during which loop is running.

models to perform poorly due to the higher overheads.

6.4 Judicious speculation

In the previous section I described a simple model of speculation where every memory reference is tracked and the entire loop body is run in a transaction. While this model did show potential for gaining some speedups, in some cases the performance did not compare favourably with HELIX. This was largely due to a small number of dependences causing very expensive rollbacks and due to the large overhead of instrumenting every memory access in the loop body.

However, this model has essentially ignored all the hard work HELIX has already done in identifying dependences and consolidating dependence cycles into sequential segments. The memory accesses which occur outside sequential segments have been proven by the interprocedural analysis not to cause dependences. Therefore these can, in theory, be executed without instrumentation. In the speculation model described previously this was not possible, however, since it was necessary to buffer all writes so they could be rolled back in the case of a conflict.

In addition to this undesirable overhead there are also situations where a dependence identified by HELIX may occur on every single iteration (see listing 6.6). Speculation is guaranteed to fail in these instances since the entire loop will become sequentialised. It would be ideal to take advantage of profiling data to enable a judicious application of speculative execution such that the system never attempts to speculate on code which is guaranteed to fail.

Therefore I propose a new model which speculates judiciously, only when there is reason to believe that speculation will be profitable. The sequential segments previously identified by HELIX provide a convenient unit for choosing whether to speculate or not.

Listing 6.5: Kernel from automotive_susan_s.

```
// Outer loops iterate across image
for (i=mask_size;i<y_size-mask_size;i++){ /* Loop D */
    for (j=mask_size;j<x_size-mask_size;j++){ /* Loop C */
        area = 0;
        total = 0;
        dpt = dp;
        ip = in + ((i-mask_size)*x_size) + j - mask_size;//Mask starting point
        centre = in[i*x_size+j];
        cp = bp + centre;

        // Inner loops iterate within mask
        for(y=-mask_size; y<=mask_size; y++){ /* Loop B */
            for(x=-mask_size; x<=mask_size; x++){ /* Loop A */
                brightness = *ip++;
                tmp = *dpt++ * *(cp-brightness);
                area += tmp;
                total += tmp * brightness;
            }
            ip += increment;
        }
        tmp = area-10000;
        if (tmp==0)
            *out++=median(in,i,j,x_size);
        else
            *out++=((total-(centre*10000))/tmp);
    }
}
```

Listing 6.6: Sample code showing an always-true dependence.

```
for(int i = 0; i < size; i++){
    /* iter will always cause a dependence! */
    iter = iter->next;
    operate(iter->data);
}
```

The model works in two stages:

1. First a profiling stage is run which detects which sequential segments are actually causing conflicts at runtime. This is based on the method used to collect sequential segment conflict statistics in section 5.3.1.

2. A threshold percentage is chosen such that sequential segments with a conflict percentage above the threshold are synchronised and those below the threshold are executed speculatively. I implemented a specialised TM model which allows for multiple *domains* of transactions such that conflict detection only occurs between transactions in the same domain. Each static sequential segment corresponds to a single domain.

This has various advantages over the previous model, most notably that having a single always-true dependence does not kill the performance of speculation. In addition, this model may significantly reduce the number of transactional memory accesses that need to be performed. For example, if a loop has a single very short sequential segment, only the accesses within the segment need to be recorded transactionally whereas in the pure speculation model all accesses in the entire loop body would have been instrumented to facilitate rollback.

There are some potential downsides to this model however. Every time a sequential segment completes, it is necessary to stall the thread until that segment has completed in all previous iterations. This is because the code after the sequential segment is not being executed in a transaction so the sequential segment code executed speculatively must be validated and committed before the thread can continue. 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.

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. Indeed, Edler von Koch et al. [95] have shown that up to 100 different data sets may be required for these benchmarks to exhibit all data dependence behaviours. However, the current evaluation is still useful for finding an upper limit to the potential of judicious speculation.

Figure 6.6 shows a sample execution of HELIX with judicious speculation. The loop contains three sequential segments: SS0, SS1 and SS2. Profiling the loop has indicated that SS0 and SS2 rarely cause data dependence conflicts during actual execution whereas SS1 contains genuine dependences. Therefore SS0 and SS2 are executed speculatively (green shading) and their write sets are committed to main memory before continuing. SS1 is executed using the standard HELIX synchronisation primitives (yellow shading). The blue shaded sections are those which have been proven by the compiler to be free of conflicts and are executed without speculation or synchronisation.

6.4.1 Worked example

To demonstrate the manner in which this model can take advantage of both HELIX-style, synchronisation-based parallelism and speculative parallelism, it is useful to walk through a simple example. Consider the code example in listing 6.7. This is a simple loop which increments a global scalar variable, `glob`, and conditionally updates a global integer array, `A`.

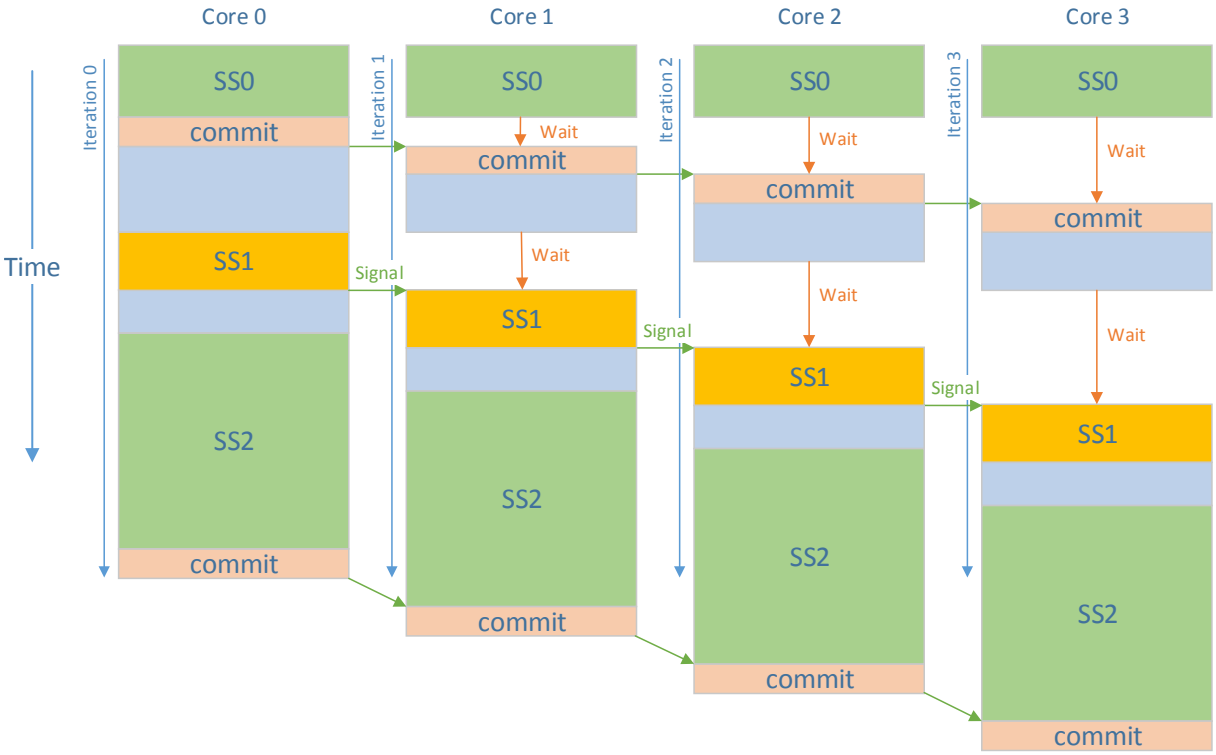

Figure 6.6: HELIX judicious speculation: Profiling indicates that SS0 and SS2 rarely cause data dependence conflicts during actual execution and can be executed speculatively (indicated by green shading).

HELIX creates three sequential segments for this loop:

- **SS ID 0**: As always, HELIX must sequentialise the loop prologue and is assigned the ID 0. In this case the prologue will simply consist of a check on the induction variable `count`. Remember that `count` can be privatised to each thread so it does not cause any dependences.
- **SS ID 1**: This sequential segment preserves data dependences on the global variable `glob` which is updated on each iteration.
- **SS ID 2**: This sequentialises accesses to the global array `A` between different iterations. On each iteration of the inner loop an index into the array is computed, an entry is read from the array at this index and the entry is conditionally updated. If two iterations were to ever compute the same index it may cause a data dependence so HELIX must be conservative here and sequentialise the entire inner loop.

It may be noted that sequential segment 1 will cause data dependences on every iteration since it is always updating the same location. Sequential segment 2 is more interesting, however, since the existence of an actual data dependence at runtime depends on the indices into the array. In fact, the calculation of the index, `factor*(count%16) + i`, is guaranteed to access a different range of indices in each iteration of the outer loop assuming the loop is not run with more than 16 cores. The compiler is unable to prove this and, indeed, does not know the number of cores until runtime so must be conservative in sequentialising this loop.

Listing 6.7: Synthetic loop with sequential segments indicated.

```
for (count = 0; count < weight; count++){
  /* Enter sequential segment 1 */
  glob++; /* Global scalar */
  /* Exit sequential segment 1 */
  /* Enter sequential segment 2 */
  for(i = 0; i < factor; i++){
    int tmp = A[factor*(count%16) + i]; /* Global array */
    tmp += count*5;
    if(tmp%2 == 0){
      A[factor*(count%16) + i] = tmp;
    }
  }
  /* Exit sequential segment 2 */
}
```

Loop ID	SS ID	Iterations	Conflicts	Conflicts %
A	0	1001	0	0.0%
	1	1000	999	99.9%
	2	1000	0	0.0%

Table 6.3: Sequential segment statistics for loop in listing 6.7

This inspection can be confirmed by running the sequential segment conflict analysis (as described in section 5.3.1). The results are shown in table 6.3. As expected, sequential segment 2 is completely conflict free while sequential segment 1 causes conflicts on every iteration. In addition, sequential segment 0 (the prologue) is also conflict free since the calculation of induction variable `count` has been privatised. This gives a little more scope for finding extra parallelism.

The results for this loop are shown in figure 6.7. Only the TCC HTM parameters are included here for clarity. The HELIX model shows negligible speedup. Some performance improvement could have been possible since three separate sequential segments were generated, thus allowing the threads to overlap different parts of the iteration concurrently. However, the proportion of execution time spent executing sequential segment 2 dominated the overall execution time of the loop so without being able to parallelise that portion, only a tiny speedup would have been possible.

The speculation model performs significantly worse than HELIX in this case. No speedup would have been possible due to the data dependence cycle caused by the update of `glob`. In this case, the model will execute the entire iteration speculatively, wait for all previous iterations to complete and then check for conflicts. On each occasion a conflict will be detected so the transaction will then be rolled back and the entire iteration will be executed again. Thus the entire loop is essentially sequentialised. The performance degradation relative to the baseline is due to the overhead of tracking memory references, conflict checking and rolling back.

The judicious speculation model does not suffer from the pure speculation model's shortcoming of having to speculate on variables which have been proven to cause conflicts. On each iteration, this model will synchronise 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 the compiler has proven this to be safe.

Now when a thread reaches the entrance to sequential segment 2, rather than waiting

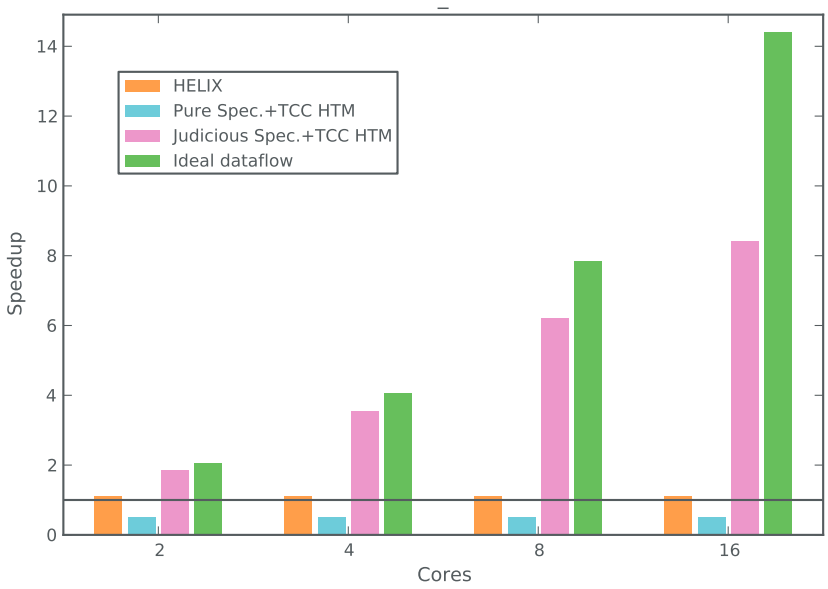

Figure 6.7: Perfomance of various models for benchmark in listing 6.7.

for all previous threads to complete the segment, it continues to execute. All memory references are tracked and writes are buffered. When the sequential segment is complete, the thread stalls until all previous threads have completed and committed sequential segment 2. Now the read set of the transaction is validated against the write sets of all previous transactions within the domain of sequential segment 2. If any previous such transaction made a write to an address which was read by the current transaction, the current transaction must be rolled back and re-executed.

As can be seen from the results, the judicious speculation model gains a significant performance boost above the baseline, HELIX and the pure speculation model. Also shown in figure 6.7 are the results for the ideal dataflow model, as described in section 5.1. The judicious speculation model does not go all the way to exploiting the speedup which the dataflow model shows is theoretically available. The dataflow model does not suffer the overheads of tracking memory references and performing conflict checking so this accounts for some of the discrepancy in performance.

However, it is interesting that while the judicious speculation model tracks the performance of the dataflow model fairly well up to 8 cores, the model seems to plateau for higher numbers of cores. 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.

In this case, the sequential segment contains 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.

6.4.2 Results

Figure 6.8 shows the loop speedups under judicious speculation for all the significant loops in the cbench applications under study. From these results it is evident that on a wide range of loops the judicious model is significantly more reliable in gaining performance above the baseline on a wide range of loops in comparison to both plain HELIX and the pure speculation model. In addition, while the pure speculation model often suffered a significant performance degradation in comparison to HELIX, the judicious model performs at least as well as HELIX in almost every case. This can be attributed to the profiling-driven nature of the new model. In cases where speculation is unlikely to perform well the model can fall back on the safer HELIX alternative while still allowing extra performance to be gained by speculating at a finer granularity in comparison to pure speculation.

It is interesting to note that, in general, the difference between the different transactional memory implementations is much less pronounced for judicious speculation than for pure speculation. This is due to the significant reduction in the number of memory accesses which must be instrumented in the judicious model since accesses outside sequential segments may be executed non-transactionally.

6.5 Transaction size

The software TM models are generally fairly robust in dealing with large transactions. TinySTM, for example, may need to resize 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 may be severely constrained by the fixed sizes of the transactional write buffer or L1 cache. In most implementations of hardware TM or hardware-supported TLS, if a transaction exceeds the maximum allowable size it must be stalled until it is safe to run non-transactionally. Obviously this results in serialisation of transactions and the complete loss of any possible performance gains. Currently my models assume infinite resources, therefore to be confident in the performance claimed by the simulation it is necessary to study the typical sizes of the transactions that are running. This is also useful for making recommendations about the scale required for future hardware implementations to effectively support TLS.

Table 6.4 shows the limitations imposed by hardware on the size of transactions for various research and commercial systems. In general, read state is 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. The figures for TCC were proposed by Hammond et. al [29] as the minimum required to effectively support TM across a broad range of applications.

Figure 6.9 shows the average and maximum sizes for all the loops I have studied in cbench when using the pure 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 reasonable write buffer size according to previous implementations. From these results it may be

	Per-transaction resources
Stampede [59]	32KB L1 cache (reads and writes)
Hydra [56]	16KB L1 cache + 2KB write buffer
Haswell TSX [96]	32KB L1 cache (reads and writes)
TCC [29]	6-12KB read state + 4-8KB write buffer

Table 6.4: Hardware resource limitations in current research and commercial HTM offerings.

concluded that the measured speedups will not be affected by limitations of the hardware and that relatively modest architectural support, on the scale of the L1 cache, could be used to achieve significant performance gains with TLS. The raw data that was used to generate these histograms is included in full in appendix B.

6.6 Summary

In this chapter I have demonstrated practical methods for exploiting the parallelism discovered in chapter 5. I presented a discussion of TM implementation styles and described the design decisions that must be made to efficiently support TLS. A survey of TM schemes and a parameterisation of such schemes was shown which allowed a meaningful comparison to be made between different styles. I proposed a timing model to simulate TM which permitted a study of the performance of the cbench loops when running each iteration in a transaction (pure speculation). This model was further refined to take advantage of profiling data and apply speculation judiciously, thus taking advantage of the best aspects of both HELIX and TLS. In general, judicious speculation is the best way to approach the theoretical limit of parallelism presented in chapter 5. Finally I have shown that the transactions I have proposed would comfortably fit within the limitations imposed by various current hardware TLS techniques.

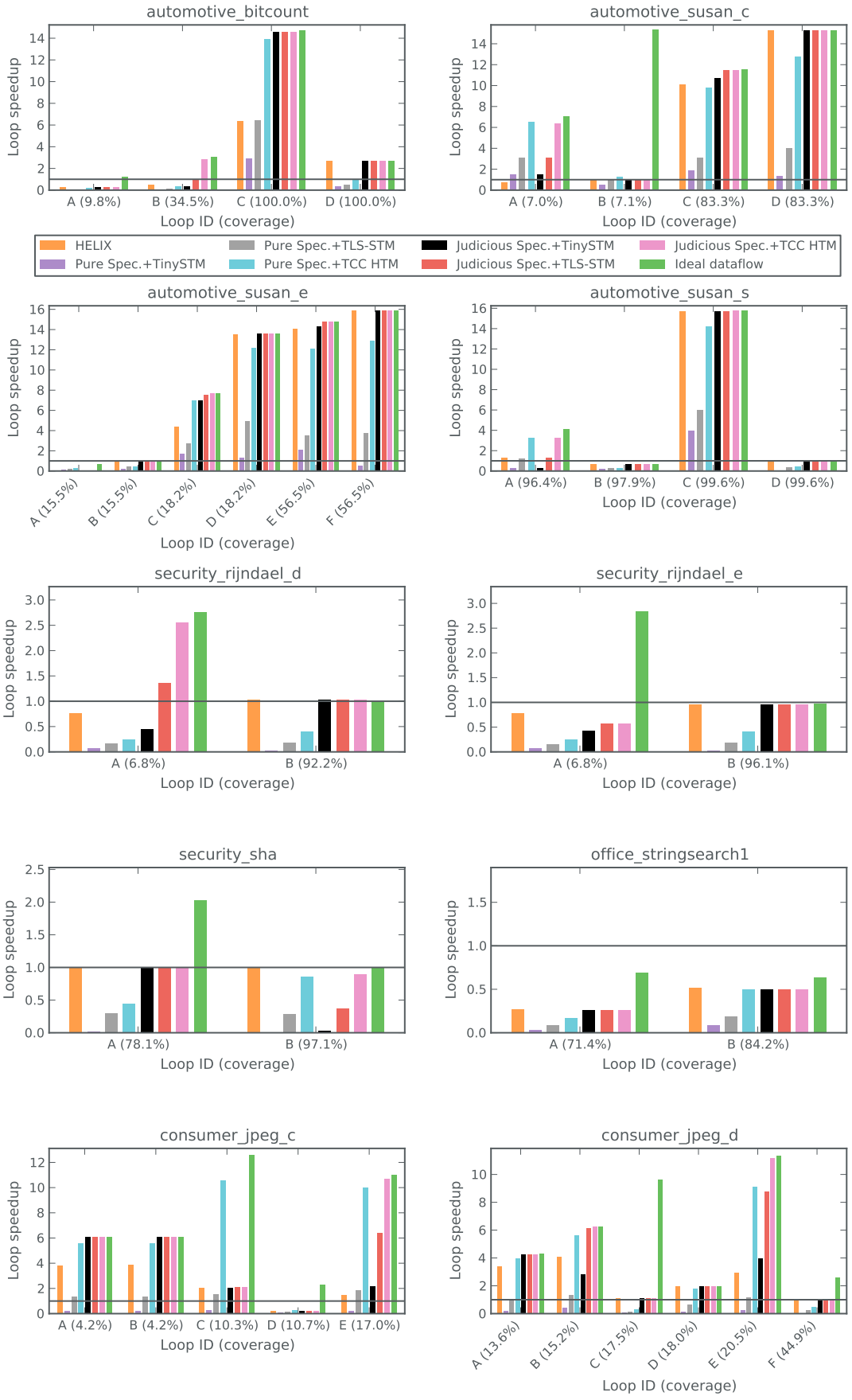

Figure 6.8: Loop breakdown for judicious speculation model with 16 cores. Coverage indicates percentage of total execution time during which loop is running.

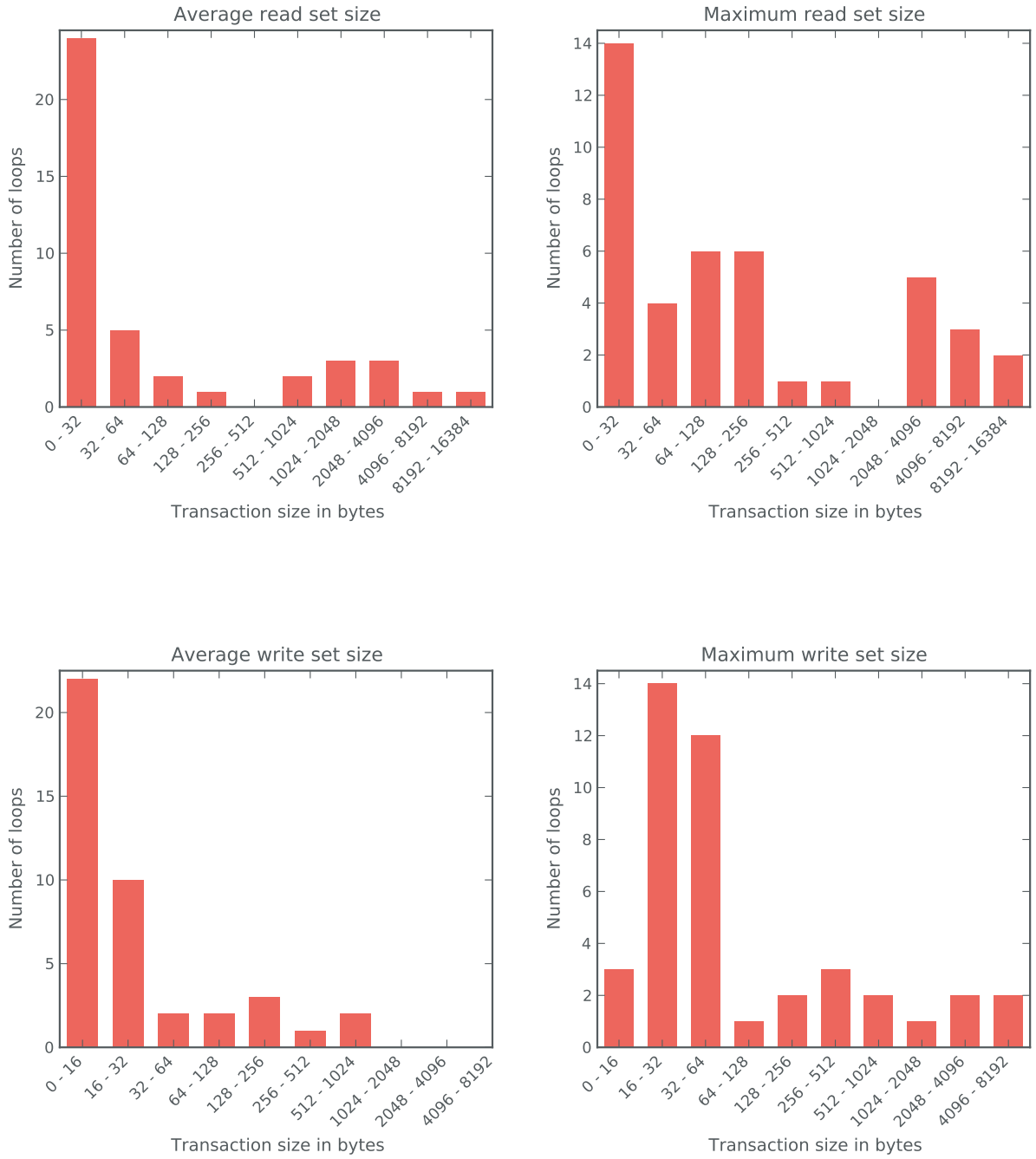

Figure 6.9: Histograms of average/maximum read/write set sizes.

Chapter 7

Conclusion

Automatic parallelisation is a promising approach to extract performance from multicore processors without burdening the programmer. However, it is not yet clear what techniques are best suited to parallelising general purpose code. Conservative strategies which rely solely on the compiler are hampered by the intractable nature of dependence analysis and cannot exploit dynamic behaviour. On the other hand, optimistic strategies which use speculation suffer from the large overheads associated with detecting data dependence conflicts at runtime. So far, the lack of hardware support in commercial processors has limited thread-level speculation to being primarily an academic curiosity.

In this dissertation I have explored a range of techniques on the conservative-optimistic spectrum. I have explored the limits of conservative automatic parallelisation in HELIX by simulating an oracle dependence analysis and have shown that improving static dependence analysis does not result in better performance for a set of embedded benchmarks. Leading on from this I answered the question of whether there is any additional parallelism available by simulating execution on an ideal dataflow machine which exploits all available inter-iteration parallelism in a loop. My analysis of these benchmarks shows that, indeed, further parallelism is available and that exploiting dynamic behaviour is required. Finally, I compare two models of dynamic execution which improve performance over the HELIX baseline by simulating a transactional memory implementation. The first is purely optimistic, speculating on every dependence. The second is a balanced approach, taking advantage of both conservative and speculative parallelism. Both models are evaluated for three different implementations of transactional memory. The results show that it is possible to improve on both the purely conservative and purely optimistic models with the balanced approach. In addition, the overheads of the transactional memory implementation are particularly significant and can make the difference in whether or not speedups are achieved.

A number of conclusions may be drawn from the results presented:

1. Improving compile-time dependence analysis is not sufficient to improve the performance of HELIX-like conservative automatic parallelising compilers.

2. Significant additional parallelism is available for some benchmarks when we look at dynamic behaviour, and requires optimistic parallelisation techniques to exploit.

3. The performance of the TM is a very significant factor and adapting commercial hardware TM extensions to support speculation will be crucial in enabling the widespread adoption of optimistic parallelisation.

4. Conservative and optimistic parallelisation both have their strengths and weaknesses and balancing these in a co-operative system will give the best opportunities to achieve the optimum performance.

7.1 Future work

Finally, I will suggest some interesting directions to expand on the work I have presented.

7.1.1 Going beyond sequential sementics

An issue which was shown to be repeatedly problematic in extracting parallelism in section 5.2.1 was that of coding artefacts which restrict the exploitation of parallelism which inherently exists. This usually occurs when the programmer introduces unnecessary constraints through the sequential programming model, for example, requiring that data must be processed in a specific order. Some previous work has provided programmers with annotations to allow them to indicate to the compiler and runtime system that such constraints need not be enforced. For example Kulkarni et al. [5] use unordered iterators to allow the programmer to express the concept of a loop in which any sequential ordering of the iterations would be acceptable. However, in the spirit of automatic parallelisation, we would like to place less burden on the programmer.

An interesting approach for future work would therefore be to develop an automatic analysis which can detect the existence of such coding artefacts. The compiler could then present these detections to the user for interactive analysis, in a similar manner to Tournavitis et al. [67]. This work could also be integrated with the analyses described in this thesis which would allow a limit study to show the extent to which automatic parallelisation techniques are hampered by sequential coding artefacts.

7.1.2 Real implementation of judicious speculation

The obvious next step for this work is to integrate a real implementation of judicious speculation into HELIX. This could be achieved by adding an extra pass to the compiler following the normal HELIX transformation. The output of the profiling run which detects sequential segment conflicts described in section 5.3.1 would be used as input to this pass. The pass would decide, based on the conflict statistics, whether or not to replace specific sequential segments with speculative transactions. The synchronisation instructions would be replaced with calls to start and commit a transaction and all intervening memory accesses would be replaced with transactional loads and stores.

An implementation of TLS-STM is currently being developed and I have shown that this model experiences significantly reduced overheads for thread-level speculation compared to a more general purpose implementation like TinySTM. Ideally it would be best to take advantage of hardware TM support such as that offered by Intel Haswell TSX. However, the current implementation of TSX does not support in-order commit of transactions and communication between transactions cannot be achieved without causing aborts. This makes it next to impossible to effectively support TLS. It is hoped that in future microarchitectures, Intel may decide to provide suitable support for speculation. While IBM Blue Gene/Q [30] and POWER8 [31] provide sufficient support to order transactions, previous work has shown that an efficient TLS implementation requires more

advanced features such as data forwarding and word-level conflict detection which are not currently available on any current processors [63].

7.1.3 Exploiting more dynamic behaviours

The dynamic behaviour exploited in the current implementation of HELIX judicious speculation is based on the frequency with which dependences in the data dependence graph are realised at runtime. However, other dynamic behaviours may also be of interest.

7.1.3.1 Phase behaviour

Phase behaviour has been observed in some benchmarks, in particular automotive_susan_c where the loop goes through a phase where every iteration is completely independent followed by a phased where every iteration conflicts. This type of behaviour could be exploited by compiling a speculative and synchronised version of the loop and switching between them based on the runtime behaviour of the loop. For example, if speculation is resulting in multiple rollbacks, it is likely profitable to switch to the synchronised version. This idea provokes many interesting questions including what the role of profiling should be in directing switching between the two styles, whether there are patterns within phase behaviour which can be predicted, and how to know if it is profitable to switch from synchronised to speculative without incurring the overhead of dependence tracking.

7.1.3.2 Dependence distance

Dependence distance refers to the number of iterations which pass between a value being produced and consumed and is an interesting behaviour to exploit. For instance, if a profiling run can detect that some dependence pair always has a distance of 3, there may be scope for automatically forwarding the value 3 iterations on with synchronisation while the intervening iterations run freely with speculation.

Bibliography

[1] R. G. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–844, August 1986.

[2] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay J. Reddi, Gu Y. Wei, and David Brooks. HELIX: automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[3] S. Campanoni, T. M. Jones, G. Holloway, Gu-Yeon Wei, and D. Brooks. Helix: Making the Extraction of Thread-Level Parallelism Mainstream. *Micro, IEEE*, 32(4):8–18, July 2012.

[4] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu Y. Wei, and David Brooks. HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs. *SIGARCH Comput. Archit. News*, 42(3):217–228, June 2014.

[5] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? *SIGPLAN Not.*, 44(4):3–14, February 2009.

[6] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[7] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *In Proc. of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

[8] Victor Pankratius, Ali-Reza Adl-Tabatabai, and Walter Tichy. *Fundamentals of Multicore Software Development.* CRC Press, 2011.

[9] Peng Tu and David A. Padua. Automatic Array Privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, London, UK, UK, 1994. Springer-Verlag.

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

[11] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[12] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih W. Liao, and Monica S. Lam. Interprocedural Parallelization Analysis in SUIF. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, July 2005.

[13] W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Trans. Parallel Distrib. Syst.*, 3(6):643–656, November 1992.

[14] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The Next Generation in Parallelizing Compilers. In *PROCEEDINGS OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING*, pages 10–1, 1994.

[15] Wolfram Blume, Ramon Doallo, Rudi Eigenmann, J. Hoeflinger, and T. Lawrence. Parallel programming with Polaris. *Computer*, 29(12):78–82, December 1996.

[16] Kemal Ebcioğlu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, MICRO 20, pages 69–79, New York, NY, USA, 1987. ACM.

[17] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 177–188, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage Decoupled Software Pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[19] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Georgios Tournavitis and Björn Franke. Semi-automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 377–388, New York, NY, USA, 2010. ACM.

[21] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 370–380, New York, NY, USA, 2009. ACM.

[22] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative Decoupled Software Pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the Sequential Programming Model for the Multicore Era. *Micro, IEEE*, 28(1):12–20, January 2008.

[24] Nicholas Wang, Michael Fertig, and Sanjay Patel. Y-branches: when you come to a fork in the road, take it. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 56–66. IEEE, 2003.

[25] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 42 of *PLDI '07*, pages 211–222, New York, NY, USA, June 2007. ACM.

[26] Maurice Herlihy, Eliot, J. Eliot, and B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[27] Per Hammarlund, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Alberto J. Martinez, Tom Piazza, Ted Burton, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, and Mikal Hunsaker. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34(2):6–20, March 2014.

[28] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David Wood. LogTM: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254–265. IEEE, February 2006.

[29] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. *SIGARCH Comput. Archit. News*, 32(2), March 2004.

[30] Ruud A. Haring, Martin Ohmacht, T. W. Fox, Michael K. Gschwind, David L. Satterfield, Krishnan Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, George L. T. Chiu, Peter A. Boyle, Norman H. Chist, and Changhoan Kim. The IBM Blue Gene/Q Compute Chip. *Micro, IEEE*, 32(2):48–60, March 2012.

[31] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, January 2015.

[32] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[33] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007.

[34] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[35] William N. Scherer and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[36] Bratin Saha, Ali Reza Adl Tabatabai, Richard L. Hudson, Chi C. Minh, and Benjamin Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-core Runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[37] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing Memory Transactions. *SIGPLAN Not.*, 41(6):14–25, June 2006.

[38] Tom Knight. An Architecture for Mostly Functional Languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 105–112, New York, NY, USA, 1986. ACM.

[39] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 414–425. IEEE, June 1995.

[40] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, HPCA '98, Washington, DC, USA, 1998. IEEE Computer Society.

[41] Gurindar Sohi. Retrospective: Multiscalar Processors. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 111–114, New York, NY, USA, 1998. ACM.

[42] Manoj Franklin and Gurindar S. Sohi. Register Traffic Analysis for Streamlining Inter-operation Communication in Fine-grain Parallel Processors. *SIGMICRO Newsl.*, 23(1-2):236–245, December 1992.

[43] Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. The anatomy of the register file in a multiscalar processor. In *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on*, pages 181–190. IEEE, November 1994.

[44] Manoj Franklin and Gurindar S. Sohi. ARB: a hardware mechanism for dynamic reordering of memory references. *Computers, IEEE Transactions on*, 45(5):552–571, May 1996.

[45] T. N. Vijaykumar and Gurindar S. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 81–92, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[46] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, volume 30 of *PLDI '95*, pages 218–232, New York, NY, USA, June 1995. ACM.

[47] Lawrence Rauchwerger and David Padua. The Privatizing DOALL Test: A Run-time Technique for DOALL Loop Identification and Array Privatization. In *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, pages 33–43, New York, NY, USA, 1994. ACM.

[48] Manish Gupta and Rahul Nim. Techniques for Speculative Run-time Parallelization of Loops. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–12, Washington, DC, USA, 1998. IEEE Computer Society.

[49] Francis Dang and Lawrence Rauchwerger. Speculative Parallelization of Partially Parallel Loops. Technical report, College Station, TX, USA, 2000.

[50] Marcelo Cintra and Diego R. Llanos. Toward Efficient and Robust Software Speculative Parallelization on Multiprocessors. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 13–24, New York, NY, USA, 2003. ACM.

[51] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software Behavior Oriented Parallelization. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 223–234, New York, NY, USA, 2007. ACM.

[52] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or Discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 330–341, Washington, DC, USA, November 2008. IEEE Computer Society.

[53] Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. Enhanced speculative parallelization via incremental recovery. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 189–200, New York, NY, USA, 2011. ACM.

[54] Zhen Cao and Clark Verbrugge. Mixed Model Universal Software Thread-Level Speculation. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 651–660. IEEE, October 2013.

[55] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative Separation for Privatization and Reductions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 359–370, New York, NY, USA, 2012. ACM.

[56] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 32(5):58–69, October 1998.

[57] J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Architectural Support for Thread-Level Data Speculation. 1997.

[58] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, HPCA '98, Washington, DC, USA, 1998. IEEE Computer Society.

[59] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede Approach to Thread-level Speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, August 2005.

[60] Matthew Frank, C. Andras Moritz, Benjamin Greenwald, Saman Amarasinghe, and Anant Agarwal. Suds: Primitive mechanisms for memory dependence speculation. *Cambridge, UK, Tech. Rep*, 1999.

[61] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, September 1997.

[62] Norman P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, volume 18 of *ISCA '90*, pages 364–373, New York, NY, USA, 1990. ACM.

[63] R. Odaira and T. Nakaike. Thread-level speculation on off-the-shelf hardware transactional memory. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 212–221. IEEE, October 2014.

[64] Michael Hind. Pointer Analysis: Haven'T We Solved This Problem Yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[65] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. Practical and Accurate Low-Level Pointer Analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[66] James R. Larus. Loop-level parallelism in numeric and symbolic programs. *Parallel and Distributed Systems, IEEE Transactions on*, 4(7):812–826, July 1993.

[67] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F. P. O'Boyle. Towards a Holistic Approach to Auto-parallelization: Integrating Profile-driven Parallelism Detection and Machine-learning Based Mapping. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[68] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. Alchemist: A Transparent Dependence Distance Profiling Infrastructure. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 47–58, Washington, DC, USA, March 2009. IEEE.

[69] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 535–546. IEEE, December 2010.

[70] David W. Wall. Limits of Instruction-level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 19 of *ASPLOS IV*, pages 176–188, New York, NY, USA, April 1991. ACM.

[71] Todd M. Austin and Gurindar S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. *SIGARCH Comput. Archit. News*, 20(2):342–351, April 1992.

[72] Jonathan Mak and Alan Mycroft. Limits of Parallelism Using Dynamic Dependency Graphs. In *Proceedings of the Seventh International Workshop on Dynamic Analysis*, WODA '09, pages 42–48, New York, NY, USA, 2009. ACM.

[73] Nikolas Ioannou, Jeremy Singer, Salman Khan, Polychronis Xekalakis, Paraskevas Yiapanis, Adam Pocock, Gavin Brown, Mikel Luján, Ian Watson, and Marcelo Cintra. Toward a more accurate understanding of the limits of the TLS execution paradigm. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–12. IEEE, December 2010.

[74] Tobias J. K. Edler von Koch and Björn Franke. Limits of Region-based Dynamic Binary Parallelization. *SIGPLAN Not.*, 48(7):13–22, March 2013.

[75] Keith D. Cooper and John Lu. Register Promotion in C Programs. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI-97*, pages 308–319, 1997.

[76] Robert P. Wilson and Monica S. Lam. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, volume 30 of *PLDI '95*, pages 1–12, New York, NY, USA, June 1995. ACM.

[77] Rakesh Ghiya. On the importance of points-to analysis and other memory disambiguation methods for c programs. In *In Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–58, 2001.

[78] Simone Campanoni, Giovanni Agosta, Stefano Crespi Reghizzi, and Andrea Di Biagio. A highly flexible, parallel virtual machine: design and experience of ILDJIT. *Softw: Pract. Exper.*, 40(2):177–207, February 2010.

[79] GCC4CLI web page. https://gcc.gnu.org/projects/cli.html. Accessed: 2015-09-17.

[80] Alexandru Nicolau, Guangqiang Li, and Arun Kejariwal. Techniques for Efficient Placement of Synchronization Primitives. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 199–208, New York, NY, USA, 2009. ACM.

[81] cBench: Collective Benchmarks. http://www.ctuning.org/cbench. Accessed: 2015-09-17.

[82] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, volume 0 of *WWC '01*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[83] Simone Campanoni, Harvard University, personal communication.

[84] Tor E. Jeremiassen and Susan J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors Through Compile Time Data Transformations. *SIGPLAN Not.*, 30(8):179–188, August 1995.

[85] Nick P. Johnson, Taewook Oh, Ayal Zaks, and David I. August. Fast Condensation of the Program Dependence Graph. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 39–50, New York, NY, USA, 2013. ACM.

[86] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[87] Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional, third edition, July 1997.

[88] Mikko H. Lipasti and John P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society.

[89] StephenM Smith and Brady. SUSANA New Approach to Low Level Image Processing. 23(1):45–78, 1997.

[90] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based Transactional Memory with Scalable Time Bases. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[91] Stephan Dieselhorst, ARM, personal communication.

[92] K. Olukotun, L. Hammond, and J. Laudon. Chip Multiprocessor Architecture:Techniques to Improve Throughput and Latency. page 145.

[93] R. D. Boyle and R. C. Thomas. *Computer Vision: A First Course*. Blackwell Scientific Publications, Ltd., Oxford, UK, UK, 1988.

[94] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 365–375. IEEE, 2007.

[95] Tobias J. K. Edler von Koch and Bjorn Franke. Variability of data dependences and control flow. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 180–189. IEEE, March 2014.

[96] Zhaoguo Wang, Hao Qian, Haibo Chen, and Jinyang Li. Opportunities and Pitfalls of Multi-core Scaling Using Hardware Transaction Memory. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, New York, NY, USA, 2013. ACM.

Appendix A

Oracle DDG data

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	0	0	100.0%
B	0	0	100.0%
C	4	1	25.0%
D	58	58	100.0%

Table A.1: automotive_bitcount.

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	62	0	0.0%
B	62	0	0.0%
C	164	0	0.0%
D	164	0	0.0%

Table A.2: automotive_susan_c.

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	9726	1778	18.3%
B	9726	2110	21.7%
C	158	0	0.0%
D	158	0	0.0%
E	74	0	0.0%
F	74	0	0.0%

Table A.3: automotive_susan_e.

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	3	0	0.0%
B	3	0	0.0%
C	4	0	0.0%
D	4	0	0.0%

Table A.4: automotive_susan_s.

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	2	0	0.0%
B	9	0	0.0%

Table A.5: office_stringsearch1.

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	4	4	100.0%
B	2	1	50.0%

Table A.6: security_sha.

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	5	0	0.0%
B	16	16	100.0%

Table A.7: security_rijndael_d.

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	5	0	0.0%
B	19	17	89.5%

Table A.8: security_rijndael_e.

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	200	0	0.0%
B	200	0	0.0%
C	21	0	0.0%
D	9	9	100.0%
E	2	0	0.0%

Table A.9: consumer_jpeg_c.

Loop ID	Compile-time DDG edges	Oracle DDG edges	Accuracy
A	18	0	0.0%
B	19	0	0.0%
C	6	0	0.0%
D	48	0	0.0%
E	19	0	0.0%
F	76	11	14.5%

Table A.10: consumer_jpeg_d.

Appendix B

Transaction size data

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	4 B	4 B	5 B	12 B
B	4 B	12 B	4 B	16 B
C	12 B	24 B	0 B	20 B
D	48 B	268 B	155 B	604 B

Table B.1: automotive_bitcount.

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	4 B	212 B	0 B	36 B
B	2 KB	4 KB	20 B	196 B
C	48 B	176 B	0 B	36 B
D	2 KB	3 KB	29 B	328 B

Table B.2: automotive_susan_c.

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	12 B	68 B	4 B	56 B
B	1 KB	2 KB	37 B	124 B
C	14 B	192 B	0 B	32 B
D	3 KB	8 KB	37 B	260 B
E	76 B	176 B	0 B	28 B
F	4 KB	4 KB	206 B	1 KB

Table B.3: automotive_susan_e.

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	12 B	68 B	4 B	56 B
B	1 KB	2 KB	37 B	124 B
C	14 B	192 B	0 B	32 B
D	3 KB	8 KB	37 B	260 B
E	76 B	176 B	0 B	28 B
F	4 KB	4 KB	206 B	1 KB

Table B.4: automotive_susan_s.

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	10 B	12 B	8 B	20 B
B	18 B	172 B	13 B	16 B

Table B.5: office_stringsearch1.

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	23 B	24 B	19 B	20 B
B	23 B	24 B	19 B	20 B
C	23 B	24 B	19 B	20 B
D	23 B	24 B	19 B	20 B
E	15 B	16 B	4 B	8 B
F	82 B	84 B	397 B	404 B
G	181 B	3 KB	180 B	4 KB

Table B.6: security_sha.

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	7 B	8 B	4 B	16 B
B	1 KB	2 KB	107 B	2 KB

Table B.7: security_rijndael_d.

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	7 B	8 B	4 B	16 B
B	1 KB	3 KB	105 B	2 KB

Table B.8: security_rijndael_e.

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	16 B	32 B	16 B	32 B
B	16 B	32 B	16 B	32 B
C	40 B	44 B	12 B	24 B
D	12 B	68 B	4 B	36 B
E	7 B	8 B	4 B	20 B

Table B.9: consumer_jpeg_c.

Loop ID	Avg read size	Max read size	Avg write size	Max write size
A	23 B	64 B	16 B	32 B
B	24 B	68 B	5 B	16 B
C	14 B	16 B	11 B	12 B
D	22 B	184 B	22 B	164 B
E	36 B	40 B	6 B	40 B
F	911 B	4 KB	908 B	4 KB

Table B.10: consumer_jpeg_d.