IMPERIAL

MEng Individual Project

Imperial College London

Department of Computing

Exploring the Potential of Speculative Parallelisation

Author:
Louis Aldous

Supervisor:
Prof. Paul Kelly

Second Marker:
Prof. Peter Pietzuch

June 14, 2024

Abstract

Modern compilers can introduce significant performance improvements to programs by exploiting parallelism in code. This allows multiple loop iterations to operate concurrently if no dependencies exist. Determining if this is the case is imprecise at compile-time, leading to missed instances of parallelism that leave performance on the table. Speculative parallelisation, where we assume no dependencies exist, can help solve this problem.

We implement LLVM passes to outline loop bodies and instrument them with calls to an external library that handles the scheduling of parallel executions, conflict-checking, and rolling back if there are conflicts. The library ensures semantic correctness in the case of a conflict by restoring the program to the state before the speculative-parallel execution.

A suite of benchmarks evaluates the key bottlenecks in this approach to understand the limitations of the implementation and where future work can make improvements. These benchmarks demonstrate that run-time conflict detection is slow and the software implementation slows program execution by over 50x in cases where stores and loads are abundant in parallel sections. However, in scientific benchmarks we only experience a 2x slowdown on average, suggesting potential for performance gains if we have hardware-accelerated run-time detection. Although conflict detection is slow, it consistently finds conflicts early in the parallel execution if they exist, decreasing wasted work in the parallel execution.

We then explore how these limitations can be overcome, by suggesting improvements to the software implementation and discussing how the software can be accelerated or fully implemented in hardware. In conclusion, if completely implemented in software the library's overhead is too high to achieve performance gains. However, key bottlenecks, such as conflict detection, can be implemented in hardware to overcome these overheads and, with additional heuristics collected by the hardware, increase the likelihood of realised performance gains through reduced program execution time using this implementation.

Acknowledgements

I would like to thank Prof. Paul Kelly for helping me find a new project after Christmas; 3 weeks before the Interim Report deadline. He is a fantastic supervisor, and his support throughout kept the project on course and would always motivate me to put in as much effort as I possibly could. He inspired me to think outside the box and helped me foresee issues before they became problems, saving time debugging so I could spend more time improving the implementation.

I would also like to thank Luke Panayi for providing help with LLVM throughout the project. The resources he provided at the start of my project allowed me to get my footing in LLVM. His assistance towards the end of the project with statically linking my passes to `Clang` was also very important for my testing and benchmarking.

Contents

List of Figures

Chapter 1

Introduction

An interesting issue in modern-day performance engineering is how to find and exploit obscured parallelism in programs; what is the extent of performance left on the table by missing parallelisable workloads due to compiler impreciseness? Detecting a workload's parallelism is determined by dependence and pointer analyses. These analyses are undecidable without constraints and give rise to imprecision in the compiler, resulting in runtime instances where a particular loop or block can be parallelised but is executed sequentially.

Overcoming these constraints could lead to major performance uplifts in numerous programs where these parallelisable runtime instances were not recognised. Given that we are almost at the limit of how far we can shrink transistor size finding techniques that essentially uncover hidden performance in programs is more important now than ever. It's a clear trend that the gain in generation-over-generation performance is beginning to slow, and finding novel techniques to extract as much performance as possible out of hardware will be a cornerstone of future software performance.

An area that can be investigated to solve this is speculative parallelisation using runtime conflict detection, providing two key opportunities: the imprecision from compiler analysis doesn't prevent parallelisation where it is available (since we speculate it always exists), and non-trivially parallel problems can be automatically parallelised at runtime without developer interaction. Typically, these workloads can be easily split into separate parallel sections, with examples including proof-of-work systems, matrix multiplication, and other scientific/calculation-heavy workloads such as genetic algorithms. However, some developers may fail to recognise when a problem can be executed in parallel or may not want to devote time to parallelising sequential implementations of these problems, leaving performance on the table. By using speculative parallelisation and runtime conflict detection, we can realise this lost performance by executing the job in parallel whilst checking for conflicts and handling them appropriately if they are present.

1.1 Objectives

This project aims to create a software implementation of a speculative parallelisation runtime with runtime conflict detection and rollback, to lay the groundwork for future optimisations in the software implementation or future acceleration or implementation by hardware. Using a compiler pass we can separate a loop body into multiple functions that can be executed in parallel and instrument their instructions with calls to a runtime. The runtime is responsible for conflict checking and has a threadpool implementation which schedules the parallel jobs in a particular order to minimise the chance of conflicts. If a conflict is detected, the runtime will rollback the program state to what it was before we attempted the parallel section and then execute the sequential code. Heuristics can be used to minimise the cost of rollbacks across a program execution.

The compiler we will be using for this project is LLVM. LLVM performs middle-end passes on LLVM IR, a custom intermediate representation produced from the user's source code by the

compiler front-end. This project implements a speculative-parallel runtime for programs compiled using a modified version of LLVM, which will explore the potential of speculative parallelisation and lay the groundwork for future research into runtime conflict detection techniques and potential hardware-accelerated implementations of the library for improved performance.

1.2 Contributions

There have been numerous past attempts at speculative hardware parallelisation, particularly in hardware. One such attempt at speculative parallelisation is a hardware implementation called SWARM[5] from a research group at MIT. However, due to its complicated hardware implementation and its need for some developer intervention, SWARM is not practical for widespread use and currently only a simulator for it exists.

This project takes inspiration from particular aspects of SWARM's implementation to make the following contributions:

1. Create an LLVM pass that can detect, simplify, and outline loop bodies into a function representing a single loop iteration, with support for loop nesting and other control-flow changes.

2. Produce an external library that handles the speculative-parallel execution of a loop and correctly detects read-after-write, write-after-write, and write-after-read violations at run-time; restoring the program to a correct state and continuing execution sequentially if such a violation is detected.

3. Investigate and assess the effect of the library's implementation on a program's execution time; understanding and explaining where slow-downs occur.

4. Highlight how the library can be improved to reduce the performance penalty from conflict detection and overheads handling the speculative-parallel execution.

1.3 Conclusion

In benchmarks focussing on scientific computation, such as sparse matrix multiplication and Monte Carlo simulations, we experience a slowdown of 2x on average when using speculative parallelisation. However, workloads heavy on writes suffer significantly greater slow-downs, such as normal matrix multiplication which results in a 1000x slow-down. Across micro-benchmarks, most workloads suffer around a 50x slow-down. The slow-downs are caused by run-time conflict detection, which creates lots of overhead for each load and store. However, the conflict detection implementation consistently finds dependency violations early in a parallel execution when they exist, preventing the program from wasting too much work.

These findings suggest that a hardware-accelerated implementation of the conflict-detection could be the key to unlocking the performance brought about by speculative parallelisation as this would eliminate the overheads that currently hinder the software implementation.

Chapter 2

Background

This chapter provides the knowledge of the concepts required to understand the project and the tools that will be used to achieve the project's objectives.

2.1 LLVM

LLVM is an open-source project containing many different components: front-ends for numerous languages including C/C++, Haskell, and Rust, an intermediate representation that is used for LLVM passes and analysis (LLVM IR), and a compiler back-end for translating LLVM intermediate representation into machine-targeted binaries.

2.1.1 Compilers

Compilers are programs which take user source code and turn it into a format that can be linked with other programs and executed by hardware. Compilers also optimise programs, achieved by turning source code into an intermediate representation and performing different analyses on it. For example, one such analysis is constant propagation, which examines the intermediate representation to determine if some expressions can be evaluated into a constant value during compilation. This moves the runtime cost of an expression's evaluation into the compile-time, inherently increasing the performance of the compiled source code.

2.1.2 Compiler Architecture

Generating an executable binary from a user's source code takes numerous steps. Figure 2.1 demonstrates the required steps to turn source code into an executable format. Lexical analysis, syntax analysis, semantic analysis, and intermediate code generation are performed by the compiler's front end. This is language-specific, although it can be generalised to a family of languages such as the C family, and transforms user source code into an intermediate representation for the back-end of the compiler to analyse. At this stage, the optimisations performed are not machine or architecture-specific and can be applied generally without regard for hardware limitations (e.g. the number of registers a processor has). After this stage, the optimised intermediate code will be used to generate machine-dependent code, which accounts for different architectures or instruction sets that a processor may have (e.g. compiling to ARM or x86-64).

2.1.3 Clang

Clang is the compiler front-end used to convert C into LLVM IR. It is directly integrated into the LLVM project. Clang supports the C language family, including C++, CUDA, and Objective C/C++. Although it serves as a front-end, it can also compile directly to binary by using LLVM as a back-end.

2.1.4 LLVM Intermediate Representation (LLVM IR)

The LLVM intermediate representation (LLVM IR for short) is a RISC programming language, similar to assembly. However, machine-specific aspects of assembly, such as addressing registers,

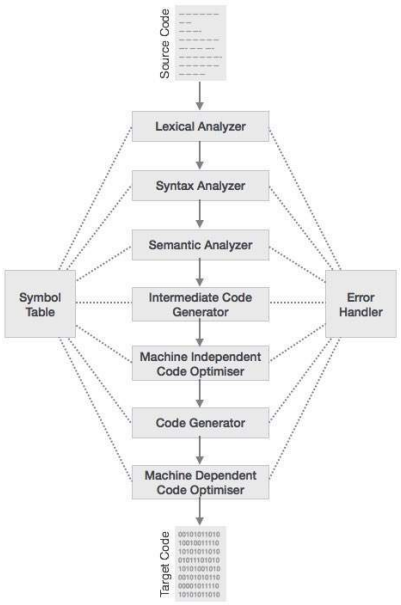


Figure 2.1: Steps of a compiler, taken from [1].

are abstracted away and LLVM IR instead uses a limitless number of register names. Local variables are prefixed with `"%"` and global identifiers are prefixed with `"@"`.

LLVM programs use a module structure. Modules consist of functions, global variables, and symbol table entries and are linked together by the LLVM linker. It also supports linkage types, which affect the visibility of objects (functions, global variables, etc.) within modules, allowing for private accessibility or external linkage with global variables in other modules.

A `BasicBlock` in LLVM IR is a block of instructions that ends with a terminator instruction. A terminator instruction can be a `ret` instruction but could also be a `br` (branch) instruction to another `BasicBlock`. A `br` instruction forms an "edge" between two `BasicBlocks`.

Here is an example of an LLVM IR "hello world" module, provided by [7]:

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(ptr nocapture) nounwind

; Definition of main function
define i32 @main() {
  ; Call puts function to write out the string to stdout.
  call i32 @puts(ptr @.str)
  ret i32 0
}
```

First, we define a private global variable `.str` which holds our "hello world" string, and an external declaration of the `puts` function which takes a string pointer as a parameter and writes its value to `stdout`. Our `main` function is then defined underneath and calls the `puts` function, passing the pointer to `.str` as its argument.

2.1.5 LLVM Passes

Passes in LLVM implement the key optimisations that LLVM provides. They traverse the code to collect information and make transformations to the LLVM IR accordingly.

There are two main types of passes in LLVM:

- **Analysis Passes** analyse the code in some way to gather and create useful information that

can be used by later passes to apply transformations to the LLVM IR. For example, `memdep` is an analysis pass determining what preceding operations a memory operation depends on, this can be useful for finding dependencies between instructions.

- **Transform Passes** use analysis passes to transform the LLVM IR in some way. Loop invariant code motion is an example of a transform pass. The LLVM passes we implement in this project are also transform passes.

There are also **utility passes** which provide utility but can't be categorized into analysis or transform passes. For example, we use a utility pass to clone `BasicBlock`s to extract loop bodies. Passes are registered and initialised in the `PassRegistry` class.

2.2 Loop Terminology

The LLVM passes that this project contributes focuses on transforming loops and loop bodies in a way that allows for easy parallelisation at run-time. Some important loop terminology is as follows[8]:

- A loop header dominates all instructions in a loop and is part of the loop.
- A loop preheader dominates all instructions in a loop but isn't part of the loop itself.
- A latch has an edge to the loop header, this edge is called a backedge.
- Exiting blocks are blocks that have edges that leave the loop. The targets of these edges are called exit blocks.

2.2.1 Loop Canonical Forms

Canonical forms, particularly in the context of loops, provide invariants that can aid future analysis and transformations. LLVM provides a canonical form for loops using `LoopSimplify`, which, when successful, ensures the loop has[8]:

- A preheader.
- A single backedge.
- Dedicated exits, so no exit block has a predecessor outside the loop.

This form allows us to be sure of control-flow behaviour within the loops we are transforming. For example, we can ensure that we can't enter the loop from anywhere other than the preheader, so any variables we wish to add to the loop's scope can be added to the preheader.

2.3 Barriers to Parallelisation

Parallelisation can be hindered by pointer aliasing and dependencies between instructions. To use parallelisation and gain its benefits, we must first understand data hazards and how we can handle them, to allow instructions to execute independently of each other whilst handling conflicts dynamically at run-time. Typically, if a compiler can determine that a loop can be parallelised it will add vectorisation (where a single instruction can modify multiple memory addresses at once.)

2.3.1 Dependencies

Dependencies describe how memory locations in two different instructions are related to each other. Take two instructions, S_1 and S_2, where the execution of S_1 precedes the execution of S_2. Four types of data dependence could exist between them:

- **Flow dependence** (also known as **true dependence**) occurs when S_1 writes to a memory location that S_2 reads. This is also referred to as a read-after-write hazard.
- **Anti-dependence** occurs when S_2 writes to a memory location that S_1 reads. This is commonly found in loops that iterate over a data structure and update the current index using information from its previous index. This is a write-after-read hazard.

- **Output dependence** occurs when S_1 and S_2 write to the same memory location. This is a write-after-write hazard
- **Input dependence** occurs when S_1 and S_2 read from the same memory location.

Dependency analysis finds the dependencies between two given code statements throughout the instructions of a program, which is used to determine if code can be safely parallelised or not. In most cases, a loop can be parallelised if it has loop-independent dependence, which is when no instructions in an iteration of a loop depend on any future or previous loop iterations. If an instruction does depend on an instruction from another iteration, we call the instruction a loop-carried dependence.

Take the following loop as an example of what can be parallelised:

```
for(int i = 0; i < size; ++i){
    A[i] = A[i] * 2;
}
```

Performing dependence analysis here shows that the instruction is loop-independent, as indexing `A` doesn't depend on any other future or previous loop iterations. During compilation, this loop should be turned into a single vector instruction that simultaneously multiplies the data in `A` by 2, saving the work done by performing the loop iterations sequentially.

Now let's look at a loop that cannot be safely parallelised:

```
for(int i = 1; i < size; ++i){
    A[i] = A[i] * A[i-1];
}
```

Dependence analysis here yields that we have an anti-dependence between two iterations of the loop, as the location `A[i-1]` is written to in the previous iteration, and therefore we have a loop-carried dependence. This is a case where the processor must process each iteration sequentially.

2.3.2 Pointer Aliasing

Pointer aliasing is when two pointers refer to the same memory location. Pointer analysis is a static analysis that attempts to establish if two pointers map to the same memory address or have overlapping memory addresses. Exact pointer analysis is undecidable; implementations in compilers use constraints such as field sensitivity, context sensitivity, and flow sensitivity. This leads to imprecision, and as correctness must be preserved the compiler assumes that two pointers alias if it cannot be proven otherwise.

For example, take the following function:

```
void timesTwo(int *a, int *b, int valuesToDouble){
    for( int i = 0; i < valuesToDouble; ++i ){
        *a = *a * 2;
        *b = *b * 2;
        a++;
        b++;
    }
}
```

This function takes two lists of integers and how many elements from the start of the list should be doubled. Static analysis performed by the compiler cannot determine if `a` and `b` point to different memory locations. Therefore, the compiler cannot confidently parallelise this loop, as this creates an output dependence. However, that's not the only concern; `a` and `b` could iterate on overlapping memory regions, which creates a loop-carried dependency. This is how pointer aliasing can block parallelisation at compile-time, as there is no way to statically determine if `a` or `b` alias within or between iterations of the loop.

Chapter 3

Related Work

This chapter reviews the literature surrounding attempts to find parallelism and parallelise code. The SWARM architecture[5] is also introduced, which attempts to exploit parallelism absent in normal architectures using speculative parallelisation.

3.1 Inspector-Executor Schemes

`DOALL` and `DOACROSS` loops can execute their instructions in parallel across multiple cores and are statically detected by the compiler.

3.1.1 `DOALL` and `DOACROSS` Parallelism

`DOALL` loops have no loop-carried dependencies and all iterations can be parallelised and split across multiple processors. However, the creation of `DOALL` loops can be blocked by loop-carried dependencies. A `DOACROSS` loop uses synchronisation primitives between loop-carried dependencies to split the iterations across multiple cores and increase parallelism. However, this method introduces significant overhead owing to the granularity of the synchronization primitives. The inspector-executor method [9] attempts to provide a dynamic analysis of loop dependencies at runtime. During compilation, the compiler sets up a block of code that performs loop dependency analysis at runtime, which is then used to create wavefronts of loop iterations that can be concurrently executed, allowing for increased parallelism.

3.1.2 `DOCONSIDER` Loops

A study [9] proposes the `DOCONSIDER` loop and the idea of a start-time schedulable loop. Dependency and memory access patterns in a nest of loops sometimes don't depend on variables or array elements computed in the loop nest, if this is the case we call them a nest of start-time schedulable loops. If the compiler fails its static analysis, parallelization can be applied during program execution to such a loop. The compiler transforms a loop into two parts: the inspector, which reorders and partitions loop indexes, and the executor which carries out the work scheduled by the inspector.

The study proposes two types of executors:

- Prescheduled executors where loop iterations are pre-assigned to processors and the instructions corresponding to each schedule are parallelised. All processors must complete their schedules before the next set of schedules can start.
- Self-executing executors use a `DOACROSS` loop to compute the sorted loop indexes and allow for each processor to begin work on a wavefront before all the dependencies within it are satisfied.

Its results found that a self-executing executor combined with local index scheduling provides robust performance with a low overhead for set-up time, and is recommended by the study when creating an inspector executor scheme. However, the same executor with global scheduling provides the most robust performance at the cost of a high setup time.

section	1	2	3
iterations	1 2 3 4 5	6 7 8 9 10	11 12 13 14 15

(a) Partitioning iterations into sections

section	1		2		3		
wavefront number in sub-schedule	1	2	1	2	1	2	3
wavefront number in overall schedule	1	2	3	4	5	6	7
iterations in wavefront	1	3	6	8	11	12	13
	2	4	7	10	14		15
	5		9				

(b) Concatenating sub-schedules

Figure 3.1: An example of sectioning a loop with 15 iterations, taken from [2].

3.1.3 Parallelising the Inspector

J H Saltz et al. [9] found that the inspector accounted for 80% of the parallel execution time, sparking research for improving the performance of the inspector itself. A paper [2] improves the efficiency of the inspector using two different methods.

Sectioning where iterations are partitioned into consecutive sub-ranges called sections, which are then assigned across multiple processors. Each processor computes a valid schedule for the iterations, ignoring dependencies on iterations outside of its given section. This provides a schedule for each section, called a sub-schedule. Figure 3.1 gives an example of sectioning a loop with 15 iterations. It was found that using this method produced 'nearly perfect speedups'[2, p.86] as synchronisation isn't needed in any of the iterations but 'at the cost of a schedule that is deeper than necessary'[2, p.86] .

Bootstrapping aims to guarantee a minimal depth schedule at the cost of more inspector execution time. The essential idea is that if the inspector loop is similar to the source loop, in the way that it has the same dependencies between iterations, then a parallel schedule for the source loop can also be applied to the inspector. First, sectioning is used to create a valid schedule for the source loop, which is then applied to the inspector loop to guarantee a minimal depth schedule. After this, iterations can be assigned to processors accordingly and if the schedule is imbalanced then overloaded processors can reassign iterations to underloaded ones, done in parallel.

The paper [9] found that sectioning does produce a deeper schedule than bootstrapping, but this doesn't matter if there aren't enough iterations per wavefront to keep all processors busy. It was also observed that the cost of bootstrapping was justified if the executor was executed many times in the program. Figure 3.2 shows the instances where either bootstrapping or sectioning is preferable.

3.1.4 Applications of Inspector-Executor Schemes

M Arenaz et al. [10] proposes a runtime technique using the inspector-executor model to parallelise irregular assignments. Irregular assignments take the form of indexing some array A by a function $f(h)$, where h is the enclosing loop's index.

Their inspector partitions the array A into P sections, where P is the number of processors, and each section is assigned to a processor, p. The loop iteration space is therefore partitioned into sets f_p that perform write operations on disjoint sections of the array, implemented as linked lists where the next value is the processor's next loop index to perform the irregular assignment. In the paper [10] the linked list is called *next*. An array called *count* is also used to keep track of how many indexes a processor has within their block to execute. Figure 3.3a shows how this process is executed with two processors, P_1 and P_2, using solid and dashed lines respectively. The *his* ar-

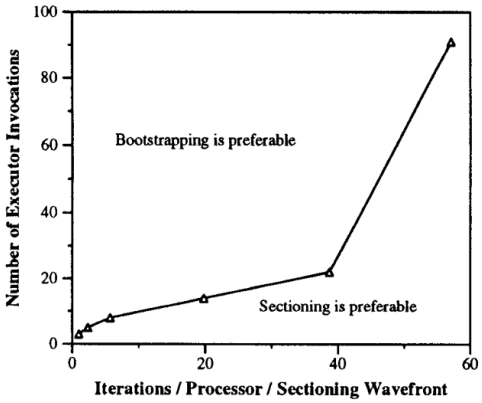
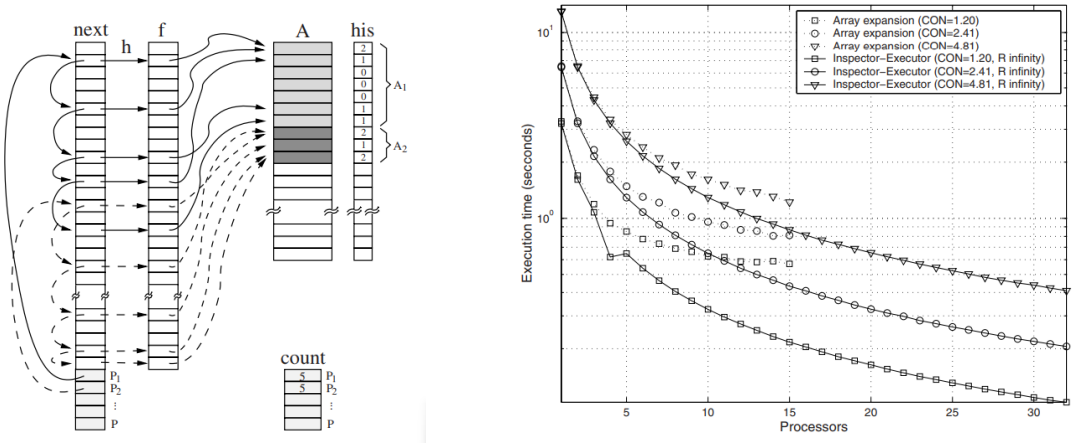


Figure 3.2: 'Number of Exectuor Invocations Required for Equal Sectioning and Bootstrapping Total Execution Time'[2, p.91].



(a) Inspector-executor for irregular assignments, taken from [10, p.6]

(b) Performance of the Inspector-Executor model for irregular assignments, taken from [10].

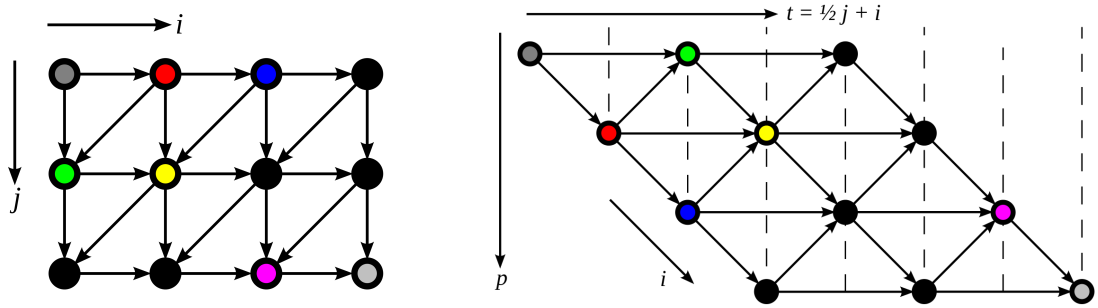
Figure 3.3: Visualisation of the inspector-executor scheme (left) and its performance comparison to array expansion (right)

ray in figure 3.3a keeps track of the number of writes to each index in the array A for load balancing.

The results prove this to be a scalable method to parallelise loops with irregular assignments by comparing it to another method called array expansion, which is a method that achieves parallelism by expanding the array A so that different processors can store their partial results in memory. Figure 3.3b shows the results (CON is the ratio between the number of loop iterations and the number of distinct array elements referenced within the loop).

3.2 OpenMP

OpenMP is an API that supports shared memory parallel processing for C/C++ and has been used in a similar way to inspector-executor schemes to enhance dynamic checking of loop-carried dependencies to further exploit parallelism. A paper [11] describes two new loops: dynamic `DOALL` loops (`D-DOALL`) and dynamic `DOACROSS` loops (`D-DOAX`). `D-DOALL` are loops that may have no loop-carried dependencies at runtime but the compiler fails to find this statically. `D-DOAX` loops have loop-carried dependencies at runtime. To detect these, the paper proposes a tool called `LoopAnalyzer` based on OpenMP and integrated into LLVM. It also provides a custom runtime that collects metrics such as a loop's CPU time and how much it contributes to the overall execution time of the application (loops with less than 10% of total CPU time are discarded from analysis).



(a) A polyhedral model for a simple loop with a nested loop. The arrow indicates a dependency between two iterations

(b) By skewing the polyhedral model, using the transformation $(p, t) = (i, 2j + 1)$, we can create wavefronts to execute iterations in parallel(shown by the dashed lines).

Figure 3.4: An example of a polyhedral model (left) with loop-skewing applied (right), taken from [3],

The paper [11] proposes an OpenMP clause called `check` which instruments a loop with calls to a custom runtime library which collects dependence information of each visit to the loop, which then is responsible for classifying the loop as `D-DOALL` or `D-DOAX`. `LoopAnalyzer` found that 'over 36% of loops across all applications from well-known benchmarks (cBench, Parboil, and Rodinia) have parallelism opportunities not explored by existing production-ready compilers'[11, p.243].

3.3 Polyhedral Frameworks

Polyhedral models characterize iteration spaces with a uniform dependence structure, which can be used to transform the schedule of the iteration space to tile loops and provide parallelism between multiple iterations. The polyhedral model provides the compiler with a global view of the operations within a loop and the dependencies between each iteration. Each loop instance is represented by a pair made of the outer-loop and inner-loop index. The dependencies are then calculated between each instance, which is later used to tile the loop to exploit parallelism.

This model also allows affine transformations to be applied to each loop instance, called loop-skewing, to allow some loop iterations to be executed in parallel. Figure 3.4 shows a polyhedral model before and after being skewed. Polyhedral frameworks have been used to implement several optimisations in modern-day compilers, including loop-skewing as well as loop permutation and tiling, to exploit parallelism across multiple cores.

3.3.1 Polyhedral Optimisation in LLVM

T Grosser et al. [12] have produced polyhedral infrastructure for LLVM, called Polly, which supports 'fully automatic transformation of existing programs' [12, p.1] and 'extracts relevant code regions without any interaction'[12, p.1]. Polly is built around an advanced polyhedral library with dependency analysis and supports OpenMP and SIMD code generation to take advantage of parallelism.

It takes LLVM IR and finds static control parts (SCoPs) which are used for polyhedral optimisations. It detects two types of SCoPs: region-based and semantic. Region-based SCoPs are detected by creating a program tree and finding valid regions, defined by the paper as a CFG subgraph connected by only an entry and exit edge. As this does not change control flow it can be replaced by a function call to an optimised version of itself. The other type, semantic SCoPs, are exclusively statements that store the result of an expression to an array element. The statement also has additional constraints: there must exist an induction variable with a lower bound and upper bound, either incremented or decremented by one each iteration, where both the upper and lower bounds must be affine in parameters that are not modified within the SCoP itself. The SCoP is then described as a set of statements each defined by a domain (the set of values the induction variable(s) can take), a schedule (relation between the domain and when each operation

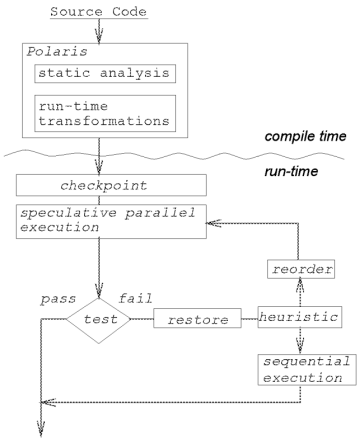


Figure 3.5: 'Integration of speculative run-time parallelization'[4, p.163]

in a statement should be executed), and a set of memory accesses. Transformations are then applied to the schedule which provides compositionality for each transformation as the composition of transformations becomes the composition of the relations for each schedule. The polyhedral representation is converted back into a generic AST by CLooG, a polyhedral code generator. Polly then detects parallel loops and inserts calls to OpenMP to take advantage of parallel code already present in the source code or exposed by other compiler passes. It also supports finding what the paper calls trivially SIMDizable loops. These loops don't have control flow statements and must take a number of iterations in an order of magnitude of the SIMD vector width.

The results of the paper [12] show that it introduced little to no overhead in compile time, and, when applying tiling using PLUTO, 6 benchmarks showed more than a 3x speedup.

3.4 Speculative Parallelisation

Speculative parallelization is where we assume a loop has no data dependencies and speculatively execute it as a `DOALL` loop while applying a parallel data dependence test. If a dependence is found, then the loop is re-executed serially, discarding the work done by the `DOALL` loop.

3.4.1 The LRPD Test

L Rauchwerger et al. [4] proposes the use of runtime tests to validate a loop that has been speculatively executed in parallel, which has the advantage of not needing to extract or analyse memory access patterns like in inspector executor schemes. To speculatively execute a loop in parallel, the paper proposes that the following is required:

- A way to save state, in case the loop needs to be re-executed sequentially. The paper proposes two ways to do this: checkpointing before the speculative execution if the resources needed aren't too big or privatising shared variables, copying them in as required, and copying out live variables given the execution is valid.
- An error detection method to ensure the speculative parallelisation is valid. Two types of errors could occur: exceptions or loop-carried dependencies in the loop. If one of these occurs, the execution is considered invalid.
- A strategy to determine when speculative parallelisation should be used. The paper proposes that a cost/benefit analysis can be run at compile time to determine if a loop can greatly benefit from speculative parallelisation.

The paper proposes 3.5 as a possible integration of speculative parallelisation at runtime, and uses what it calls the LRPD test to determine runtime validation of a speculatively executed loop.

A run-time technique is required for data dependence analysis in loops. If loop-carried dependencies exist, the test marks their existence. The lazy privatising `DOALL` (LPD) test is applied to a privatised array and tests if it is validly privatisable as well as applying dependency testing. First, it marks the values of the array that are read or written to and stores the total number of write accesses to the array in preparation for analysis.

The analysis of LPD works as follows:

1. The analysis determines loop-carried dependencies using the difference between the total number of definitions marked by all iterations in the loop and the total number of marks in the array. If this difference is greater than zero it implies the existence of a loop-carried dependency.

2. If marked areas are common anywhere between iterations of the loop, then the loop is not `DOALL`, and we exit the analysis.

3. At this stage, if the initial difference in the first step is zero then we determine the loop is a valid `DOALL` without privatisation.

4. If there is a write to a value of the array that has already been determined to not be privatisable, then the array is not privatisable either and therefore a `DOALL` loop cannot be used.

5. Otherwise, the loop can be made into a `DOALL` by privatising it.

The test's advantage over the privatising `DOALL` test is that it has what the paper calls dynamic dead reference elimination. 'A dynamic dead reference in a loop is a read access of a shared variable that does not contribute to the computation of any other shared variable which is live at the loop end'[4, p.164] and does not contribute to the data flow of the loop, therefore it shouldn't introduce false dependencies.

The LRPD test proposed by the paper is an extension of an LPD test with a test for reduction validation using the following modifications:

1. In the marking phase of the LPD test, a suspected reduction variable is marked as not a reduction variable if a reference to it is outside the reduction statement or is contained in the reduction expression itself.

2. Steps 4 and 5 of the analysis are replaced by: If a variable has been written to, and is not privatisable or a reduction variable, then the loop is not a `DOALL` loop. Otherwise, the loop can be made into a `DOALL` by parallelising the reduction and privatizing the shared array.

At compile time, after a static cost/benefit analysis, it is determined if a loop should be: speculatively executed using the LRPD test, first tested for full parallelism and then executed using the inspector/executor version of the LRPD test, or executed sequentially. The compiler then generates the code for the speculative parallel execution, augmenting the original loop with `markread, markwrite, and markredux` operations for the LRPD test. Code is also generated for the analysis of the LRPD test, sequential re-execution of the loop, and checkpointing of any variables. At runtime, the program will checkpoint the program variables if necessary and execute the parallel version of the loop (including the marking phase of the test). The analysis of the loop is then executed and, if the test passes, the final results of the reduction operations are computed using the partial results from the multiple processors. Finally, any live variables are copied out.

Across the seven benchmarks that the paper ran from the PERFECT Benchmark Suite, all apart from one demonstrated a speed-up arising from speculative parallelisation and the LRPD test, as shown in figure 3.6. As the experiments demonstrated widespread speed-ups, the paper can be considered a success. It's also worth noting that the inspector executor version of the LRPD test also provided substantial speed-ups, but was beaten in most cases by the speculative version.

Benchmark[2] Subroutine Loop	Experimental Results			Tested	Description of Loop (% of sequential execution time of program)	Inspector (computation)
	Technique	Speedup	potential Slowdown			
MDG	14 processors					privatization
INTERF	speculative	11.55	1.09	doall	accesses to a privatizable vector guarded	data accesses
loop 1000	insp/exec	8.77	1.03	privat	by loop computed predicates (92% T_{seq})	branch predicate
BDNA	14 processors					privatization
ACTFOR	speculative	10.65	1.09	doall	accesses privatizable array indexed by a	data accesses
loop 240	insp/exec	7.72	1.04	privat	subscript array computed inside loop (32% T_{seq})	subscript array
	8 processors					
TRFD	speculative	.85	2.17	doall	small triangular loop accesses a vector	data accesses
INTGRL	sched reuse	1.93	2.17		indexed by a subscript array computed	replicates loop
loop 140	insp/exec	1.05	1.74		outside loop (5% T_{seq})	
	sched reuse	2.10	1.74			
TRACK					accesses array indexed by subscript array	
NLFILT	8 processors			doall	computed outside loop, access pattern	not applicable
loop 300	speculative	4.21	1.01		guarded by loop computed predicates (39% T_{seq})	
ADM					accesses privatizable array thru aliases,	
RUN	14 processors			doall	array re–dimensioned, access	not applicable
loop 20	speculative	9.01	1.02	privat	pattern control flow dependent (44% T_{seq})	
OCEAN	8 processors				kernel–like loop accesses a vector with	data accesses
FTRVMT	speculative	2.23	1.45	doall	run–time determined strides	replicates loop
loop 109	insp/exec	2.14	1.30		26K invocations account for 43% T_{seq}	
SPICE					traverses linked list terminated by a NULL	
LOAD	8 processors			doall	pointer, all referenced arrays equivalenced	data accesses
loop 40	insp/exec	2.75	1.09	reduct	to a global work array	

Figure 3.6: Results of testing the LRPD test across the seven benchmarks, taken from [4]

3.4.2 Thread-Level Speculation

Thread-level speculation (TLS) enables parallel threads to be created without concern for if the threads are independent of one another. J G Steffan et al. [13] proposes a scalable approach to this issue, using cache coherency to detect dependence violations. Its approach requires knowing whether a cache line has been speculatively loaded or modified and a guarantee that a speculative cache line does not get propagated to memory. Speculation also fails if the speculative cache line is replaced.

Each cache line is tracked to see if its data has been speculatively loaded (SL) or speculatively modified (SM), which it summarises with the states speculative-exclusive (SpE) and speculative-shared (SpS). These states are in addition to the usual states: invalid, exclusive, shared, and dirty. Three new coherence messages are also added: read-exclusive-speculative, invalidation-speculative, and upgrade-request-speculative, which are all given with the epoch number of the requester. The epoch number keeps track of a logical ordering between different speculative threads.

The coherence scheme then works as follows:

1. When a speculative memory reference is issued, transition to either SpE or SpS as appropriate, setting the SL or SM flag for a speculative load or speculative store respectively.

2. If a speculative load misses, issue a read from memory. If a speculative store misses, issue a read-exclusive-speculative with the current epoch number. When a speculative store hits and the cache line is shared, issue an upgrade-request-speculative containing the current epoch number.

3. If a cache line has been speculatively loaded then a read-after-write violation can occur if a normal cache invalidation occurs or a speculative invalidation from an earlier epoch invalidates the cache line.

4. A dirty cache line must flush to memory when a speculative store reads or writes to it.

The epochs are implemented as partial orderings comprising a thread identifier and a sequence number so that separate threads' epochs are unordered concerning one another. The state of a cache line uses 5 bits to cover all its states; 3 bits are used for basic coherence states, and two are used to differentiate speculative states. The hardware can also have multiple speculative contexts so that the state can be maintained across OS context-switching and support SMT. It also allows the processor to execute other epochs if the current one is suspended.

The paper used SPEC95 and SPEC2000 benchmarks to evaluate the performance of TLS. The

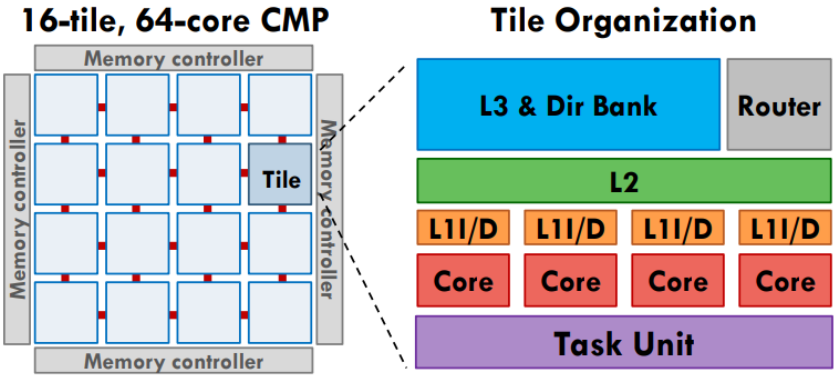


Figure 3.7: SWARM's architecture, taken from [5].

paper's [13] experimental results demonstrated speed-ups ranging from 27% to 126% in parallelised code regions. These speed-ups compare the execution time of the original binary running on a single processor without TLS overheads to running the program on multiple processors using TLS. The paper also measured parallel coverage across their benchmarks, which is what they claim the speed-up is limited by.

3.5 SWARM

SWARM is an architecture designed by M C Jeffrey et al. [5] that attempts to exploit ordered irregular parallelism, consisting of a task-based execution model, designed so that order constraints don't produce false data dependencies, and an efficiently scaling microarchitecture that uses this model.

Ordered irregular parallelism concerns applications that consist of tasks following a total or partial order, where the tasks are not known in advance, and have data dependencies that are not known before the application is executed. TLS struggles to parallelise these applications, as they tend to use data structures such as FIFO queues which introduce false dependencies due to their access patterns.

3.5.1 SWARM Architecture

SWARM provides a large speculative task window and is optimised to execute short tasks with specified order constraints. It uses a tiled architecture, as seen in figure 3.7, where each tile maintains the speculative states of finished tasks that cannot yet be committed within its task unit. The task units between tiles communicate when new tasks are sent to each other for load balancing. Each tile speculates beyond the current active task and allows speculative tasks to create additional speculative tasks, which the paper refers to as 'speculative ancestors'[5, p.231].

SWARM uses C++ extensions (`enqueue_task, dequeue_task, finish_task`) that allow the programmer to control how tasks are executed on the processor. Tasks have a timestamp associated with them; tasks with the same timestamp are executed atomically, and otherwise are run in timestamp order. Tasks can produce child tasks which are queued after its parent. A commit queue holds the speculative state of finished tasks that cannot yet be committed. SWARM uses eager versioning for speculative data, differing from TLS implementations such as in [13]. J G Steffan et al. [13] uses lazy versioning, which buffers speculative data in caches, whereas eager versioning used in SWARM stores speculative data in place and maintains a log of previous values. The paper [5] claims this makes commits fast, but slows down aborts where speculative parallelisation needs to roll-back. For aborts, a task tells its children to abort and are removed from its task queue, and the log from the eager versioning is walked to restore old values. If this conflicts with a later task (using timestamps) then that task is also aborted and the walk is continued after

it has terminated.

3.5.2 Experimental Results

SWARM was modelled using Pin to model the 64-core in 3.7. 6 benchmarks were used focussing on graph-based algorithms and database operations and found speed-ups ranging from 51x (on `msf`, a benchmark based on Kruskal's minimum spanning algorithm) to 122x (on `sssp`, a benchmark based on Dijkstra's algorithm) over a software-only parallel implementation.

3.6 Conclusion

This chapter explored the literature surrounding attempts to exploit existing but hidden parallelism in programs. Software implementations such as inspector-executor schemes and Polly for LLVM attempt to schedule loops to improve their parallelism. Hardware implementations such as TLS and SWARM use speculative approaches to find parallelism but are specialised solutions restricted to their hardware architectures.

In the next chapter, we explore the designs and aims of the speculative-parallelisation runtime that this project implements.

Chapter 4

Design and Aims

In this chapter, we provide an overview of how we aim to achieve speculative parallelisation in software, and a holistic view of how we take sequential loops and transform them to produce parallel loop bodies.

4.1 Extracting Loop Bodies

The key to parallelising loops is to extract the loop body to be independently executed across multiple threads. This means that any latches or induction variables must be replaced. The loop bounds are required to understand how many iterations to queue for execution.

This work was done by an LLVM pass, which takes a function, finds its loops, and does the following:

1. Simplify the loop into `LoopSimplify` form.

2. Analyses the loop's body to find uses that need to be replaced from definitions outside the extracted body.

3. Clones the loop's `BasicBlock`s and adds them to a new function.

4. Using the prior loop analysis, replace relevant uses with appropriate values.

5. Before the loop in the original function, introduce a call to the thread library to enqueue the extracted loop body. This library returns a value that tells the program if the parallelisation was successful. If not, we re-run the loop sequentially.

6. Perform verification on the extracted loop body and original loop for syntactic correctness.

This correctly extracts the loop body and calls the thread library to parallelise the loop. If an optimisation level is provided, such as `-O3`, the pass runs after the main optimisations (e.g. dead code elimination and loop invariant code motion), so the benefits from these transformations also benefit the outlined loop.

4.2 Instrumenting Loop Bodies

For runtime data hazard conflict detection, stores and loads in the extracted body need to be instrumented with calls to the thread library. Another LLVM pass was created to achieve this, which is executed during link-time optimisation. The key advantage of this is that during LTO the pass has full access to all function definitions, external and internal, and therefore function calls to external libraries can be instrumented. This was implemented as a Module pass, so the pass runs on the entire LLVM IR module instead of running on each function like the loop extraction pass. LLVM metadata was used to tell the pass which functions should be instrumented.

4.3 Nested Loops

Nested loops need to be handled with care, as the thread library needs to do more work to parallelise them. The LLVM pass to extract the loops earlier was modified to split a loop with nests into 3 functions: the section before the nested loop, the nested loop itself, and the exit of the nested loop. This modification simplifies the logic of the thread library, as this allowed the library to easily handle the entry of a nested loop as a synchronisation point between threads. The pass also produces a separate, sequential version of the loop in case we find a conflict when we execute the nested loop in parallel. If we find a conflict we rollback our writes and sequentially execute the nested loop from the beginning.

4.4 Handling Execution of Loop Bodies

The thread library receives calls to enqueue a loop for parallel execution. Two abstractions were introduced with parallel execution implementation: `Task`s and `Job`s. A `Job` is the state of an entire parallel execution, and includes information about conflict detection, rollback, and contains a queue of `Task`s. Each `Task` is an individual iteration and, to minimise conflicts, `Task`s are dequeued in order. However, this isn't strict, so if one iteration takes longer than others to execute `Task`s continue to be dequeued (rather than waiting). As mentioned earlier, nested loops count as synchronization points, as this simplifies the rollback mechanism; it guarantees no race conditions exist between the nested loop and its parent loop.

4.5 Conflict Detection and Rollback

The thread library is responsible for conflict detection. The implementation of conflict detection took inspiration from SWARM's[5] implementation, where we timestamp each `Task` to detect the different types of conflicts we discussed earlier.

The rollback mechanism is called whenever a conflict is detected. When a conflict is detected the parallel execution of the loop is cancelled and the `ThreadPool` stops dequeuing parallel tasks, waits for all threads to finish their current tasks (this is wasteful but means we can ensure correctness), and then overwrites all values modified by the parallel execution with their value before we entered the parallel job. Nested loops have separate conflict detection and rollback mechanisms from their parent loop, which allows us to cancel a nested loop and run it sequentially without cancelling the parallel execution of the parent loop.

4.6 Conclusion

Now we have outlined the holistic nature of how we parallelise loops, in the next section we provide more detail regarding the implementation of each component.

Chapter 5

LoopExtractionPass and InstrumentFunctionPass

This chapter discusses the implementation of the LLVM passes that analyse a function and extract its loops into a parallelisable function, including how the functions are instrumented afterwards.

5.1 Simplifying Loops into Canonical Forms

`LoopExtractionPass` handles the extraction of a loop body into a new, syntactically valid function that can be executed in parallel. LLVM's `LoopAnalysis` pass provides `LoopExtractionPass` with `LoopInfo`, providing the pass with loops to simplify, along with other important information about the loop such as its induction variable, the `BasicBlock`s comprising it, and its preheader, header, and exits. If the function has no loops, the function is skipped as there is no work to do. To simplify the pass implementation and ensure correctness, the pass uses the invariants provided by the `LoopSimplify` canonical form. If the loop cannot be simplified into `LoopSimplify` form then the loop is skipped and the pass moves on to the next loop in the function.

5.2 Cloning Loop Bodies

Now the loop is in a canonical form, a new function is created for the extracted loop body which will be executed in parallel by the thread library. The function has two parameters: the induction value of the loop body, and an array of pointers for values defined outside of the loop body that are accessed within the loop body. The pass also adds a metadata entry into the LLVM module so that `InstrumentFunctionPass` can find what functions it needs to instrument.

LLVM provides a utility pass which allows us to clone a `BasicBlock` and provides a mapping from the original instructions to the new cloned instructions. Each `BasicBlock` in the loop is cloned and added to the created function, building the value-to-value map on each invocation of the utility pass. At the same time, a mapping of original `BasicBlock`s to cloned `BasicBlock`s is also maintained by the pass.

The end result is a new function containing an extracted loop body and two complete mappings from original to cloned instructions and `BasicBlock`s. However, at this stage, the extracted loop body still has uses of definitions in the original function which, due to its extraction, is outside of its scope and therefore cannot be accessed.

There are two cases of uses that need to be replaced: uses of definitions in the original loop, and uses of definitions external to the loop. Later in the pass, these uses are replaced using the mappings constructed during the cloning.

5.3 Finding External Uses and Defs

Uses of definitions external to the extracted loop body need to be found by the pass. After the function is created and the cloned `BasicBlock`s from the original loop are added to it, the following algorithm is executed:

1. Throughout this algorithm, build up a `std::vector` with any definitions that have uses that need to be replaced later in the pass (i.e. uses of these definitions in the extracted loop.) If it is used more than once in the extracted loop we do not store the definition again, therefore there are no duplicates in this `std::vector`.

2. Iterate through the arguments of the original function and use `Argument::Uses()` to find any uses that are in the loop body. As these are function arguments, they are not defined in the loop body and are therefore external to the loop body.

3. Do the same as we did with function arguments with instructions, except we only add definitions if they are explicitly outside the loop.

Now the pass has a list of external uses that need to be replaced in the extracted body.

5.4 Replacing Internal Uses and Defs

When the loop body is cloned, LLVM still retains the references to the definitions in the original loop, which is why they need to be replaced. Fortunately, the detection and replacement of these values is quite simple.

The following algorithm carries out the replacement of these values:

1. Iterate through each instruction in the original function and retrieve its uses using `Instruction::Uses()`.

2. If a use is in the extracted loop body (i.e. in the new function created by the pass) then replace it with the definition in the extracted loop body. The mapping constructed during the cloning provides this value.

A similar algorithm is carried out with `BasicBlock` uses, such as in `br` and `phi` instructions:

1. Iterate through the keys of the `BasicBlock` mapping, each key is a `BasicBlock` in the original loop body.

2. Using the `BasicBlock::Uses()` function, all uses are retrieved for each `BasicBlock` and if they are in the extracted body then they are replaced with a reference to the newly cloned `BasicBlock` (using the mapping.)

This successfully replaces all outdated uses with new and valid definitions, provided they are already defined in the loop body.

5.5 Replacing External Uses and Defs

Replacing the use of external definitions in extracted loop bodies is not trivial because multiple iterations could be modifying it at the same time, which means simply copying the value of the definition at runtime does not result in correct semantics. Note that multiple loop iterations modifying the same definition do not necessarily imply a conflict, as the definition could span multiple memory addresses and each loop iteration could access disjoint memory locations.

As mentioned earlier, the extracted loop body takes an array of pointers as an argument. The same algorithm is performed as was done for replacing uses in instructions, but we create a load from the array of pointers and then replace the uses with this loaded value, instead of using the mapping. Some types such as integers may require another load to dereference the pointer, and this is detected at compile-time. The array of pointers is created and written to before the parallel execution is queued.

5.6 Adding Calls to the Library

After the pass is completed, the created function is verified for any syntactic errors. If this verification is successful, then the pass has produced a function that is ready to be executed in parallel. A call to the thread library's `__enqueue_task()` is executed before the original loop (which still exists in the code and is unmodified by the pass) is entered. `__enqueue_task()` takes a pointer to the parallel section, the array of pointers for external definitions, and the loop's bounds from the `LoopInfo` returned by the `LoopAnalysis` pass executed at the start of the extraction, as parameters.

`__enqueue_task()` returns a boolean value which corresponds to if the parallel execution is completed conflict-free. The pass adds a conditional branch which either branches to the rest of the function after the loop (on a successful parallel execution) or to the original loop (which is sequential). As the thread library handles rollback before the sequential loop is executed, this preserves the original program's semantics. This is where `LoopExtractionPass` finishes. `InstrumentFunctionPass` executes during LTO and works on the entire LLVM module after linking has taken place, instead of working on each function individually. This gives `InstrumentFunctionPass` access to all function definitions that it requires. This pass finds the metadata that `LoopExtractionPass` created, and retrieves the corresponding function. Each store and load is instrumented with a call to `__check_load_conflict()` and `__check_write_conflict()` respectively. Both functions take the address to conflict-check as a parameter, but `__check_write_conflict()` has an extra parameter for the size of the data type we are storing. This is for later use in the library's rollback mechanism.

To guarantee all possible conflicts are captured, any calls in the parallel loop bodies must also have their callee instrumented. However, to ensure conflict checking is not performed when not required, the callee is cloned and the clone is instrumented. The callee in the instruction is then replaced with this cloned function. A breadth-first search is performed on function calls to capture all calls within the callee. This continues until no further functions need to be instrumented. Any calls to the thread library are ignored and not instrumented.

5.7 Conclusion

This chapter discussed the fundamentals of both the `LoopExtractionPass` and `InstrumentFunctionPass`. In this state, the passes produce parallel-ready loop bodies that are correctly instrumented for conflict-checking. However, due to some complications that nested loops introduce, these passes are still not finished. The next chapter details how the `ThreadPool` and conflict detection are implemented.

Chapter 6

Conflict Detection and Task Execution

In this chapter, we discuss the implementation of the `ThreadPool` and conflict detection mechanisms of the thread library.

6.1 Library Functions

The library exposes 4 functions to the program calling it:

- `__enqueue_task()` - this function is called to start a parallel execution. It handles initialisation of the `ThreadPool` and adds tasks to the `ThreadPool` to be executed in parallel. This function also handles any arguments that are passed into the parallel tasks.

- `__check_load_conflict()` - this function is called before a value is loaded. It tells the conflict detection mechanism to check for write-after-read violations (i.e. when a later iteration has written to an address that an earlier iteration is about to load).

- `__check_write_conflict()` - this function is called before a value is stored. It tells the conflict detection mechanism to check for read-after-write violations and write-after-write violations. This function takes the size of the data type we are storing, which is required for rollbacks.

- `__malloc()` - this is a helper function that calls `malloc` but also handles the freeing of allocated memory addresses to avoid memory leaks.

6.2 Jobs, Tasks, and Timestamps

Two abstractions describe the state of a parallel execution: a `Job` and a `Task`.

- A `Task` is a single iteration of the loop body. It has a `Timestamp` for conflict detection.

- A `Job` is the extracted loop body that is being executed. This class contains a `std::priority_queue` of `Task`s to be executed, ordered by timestamp. A `Job` also has a `JobState` which is the class responsible for conflict-checking and rollback. Multiple `Job`s can share a `JobState` for conflict detection across different jobs, this is crucial for handling nested loops.

A `Timestamp` is a `std::vector` of integers, which is lexicographically compared to determine which of the two is executed first in the loop's sequential execution. This is used in the `ThreadPool` and `Job`s to execute `Task`s in parallel as close to the sequential order of execution as possible, helping to minimise conflicts.

6.3 ThreadPool and Task Execution

The ThreadPool handles how Jobs are executed and provides utility functions for JobState. A parallel execution of a loop, starting with __enqueue_task(), proceeds as follows:

1. A new ThreadPool is initialised with a number of threads specified in a macro called THREADS. ThreadPool::addTask() is called to create a new Job (if one does not exist already for the function) and a collection of Tasks which are then added to the created Job. The ThreadPool handles the creation of Tasks to simplify memory management.
2. The main thread blocks and waits for the parallel execution to finish or be cancelled (due to a conflict).
3. The ThreadPool initialises its std::threads, which all begin executing ThreadPool::dequeueTask().
4. Each thread gets the current Job and calls Job::popTask() to get the next Task to execute and then calls Task::exec() to execute the Task. Each thread repeats this until the job is cancelled or is completed.
5. On completion or cancellation, the main thread wakes up and retrieves a boolean indicating whether or not the execution was a success. This is then returned to the original program. Some clean-up also takes place, such as freeing the Jobs and Tasks created by the ThreadPool.

If a conflict is detected and the Job is cancelled, or if the Job runs out of Tasks (i.e. is completed) then all active ThreadPool threads block in Job::popTask(), where they wait for other Tasks in progress to complete. The final thread to finish executing its Task tells the ThreadPool that the Job is finished and then wakes the other threads that are blocked. An std::future is set with a boolean value to represent if the Job was cancelled or not. Setting this std::future wakes the main thread.

6.4 Rollback

To avoid tracking unnecessary memory addresses, __check_write_conflict() adds entries to the rollback register whenever the program writes to a memory address that the conflict detection mechanism hasn't seen. The rollback register is a map of memory addresses to a struct containing information regarding the size of the data stored at the address and a copy of the data before it was written to.

When a new memory address is encountered, a new block of memory is allocated by malloc according to the size received by __check_write_conflict(). memcpy is then used to store the original data in the new block of memory we have allocated, saving a copy of it. The data to be copied can be of any size, so memcpy is used as it can be provided with the number of bytes to copy. In the event of a conflict, the library waits for all threads to finish their current iteration and iterates through the rollback register, calling memcpy to copy the saved data back into its original address. This guarantees that no writes from the cancelled parallel execution propagate into the sequential execution of the loop.

6.5 Conflict Detection

A JobState maps a memory address to an AddrHistory struct, which contains two std::sets of Timestamps, one for reads and one for writes. This allows JobState to have a complete history of all reads and writes to a memory address, which is required for conflict detection. Both std::sets store the Timestamps in reverse order, which allows the use of std::set::upper_bound to check for Timestamps lower than one provided. An entry is added whenever __check_load_conflict() and __check_write_conflict() are called.

JobState has two functions which handle conflict-checking:

- doesLoadConflict() takes the memory address to load from as a parameter and uses the address's AddrHistory to find its write history. The Timestamp for the calling Task is

retrieved by calling `ThreadPool::getTimestampForCurrentThread()` and is passed into `std::set::upper_bound` to find any later `Timestamps` that may have written to the address that is about to be loaded. If there is, then this indicates a write-after-read conflict that blocks parallelisation.

- `doesStoreConflict()` takes the memory address to store to. This function needs to detect read-after-write violations and write-after-write violations. From `AddrHistory` the function gets the read history for the given address and checks if a later `Timestamp` has read the address the program is about to write to, if one exists then this shows a read-after-write conflict, which blocks parallelisation. For write-after-write violations the function checks if a later `Timestamp` has already written to the address, if a later `Timestamp` exists then this is a violation and would block parallelisation.

Similar to SWARM[5], because writes are immediately performed rather than buffered, later iterations of a loop can directly access up-to-date data without the need for extra logic. This is the main reason why executing `Task`s in as close to sequential order as possible is important, as it minimises the chance of a read-after-write violation (e.g. if the later `Task` reads the memory that the earlier `Task` writes to).

6.6 Conclusion

This chapter discussed the fundamentals of task execution and conflict detection in the thread library, whilst also covering important concepts and establishing how rollback and conflicts are handled correctly. Similar to the previous chapter, nested loops require more care. In the next chapter, the complications presented by nested loops are detailed and the modifications and additions made to both the LLVM passes and thread library to handle them are described in-depth.

Chapter 7

Handling Nested Loops

In this chapter, we cover the complications introduced by nested loops and how we modify our passes and library to account for them.

7.1 Challenges Introduced by Nested Loops

The main challenge with nested loops is how to handle them in a parallel context. They require their own `JobState` (and hence have their own rollback and `AddrHistory` map) so rollbacks only cancel the nested parallel section. Nested loops also introduce the following complications:

- How do nested loops affect execution order?
- When creating rollback entries for the nested section, how can we guarantee we have the most up-to-date values from the outer loop?
- What happens when we cancel a parallel section with a nested loop and the nested loop has already started? (since nested loops are their own `Job`, the library might need to keep track of child `Jobs`.)

Treating the entry to each nested section as a synchronisation point (akin to `#pragma omp barrier` in OpenMP) solves these complications. If the outer loop has finished all its iterations before we enter the nested section we can be sure that the rollback entries are the most up-to-date, and if we cancel the outer parallel section we can also cancel the execution of the nested loop (and hence cancel early in the iteration, rather than waiting for the entire iteration to finish.)

7.2 Handling Nested Loops in `LoopExtractionPass`

By building a list of functions that `LoopExtractionPass` has generated, the pass can detect if a loop is nested. It checks the current function and if it's in the list and contains a loop then this loop must be nested. To achieve the synchronisation behaviour required by the thread library, the pass creates 3 new functions instead of 1 (with the same signature as before):

- The parallel extracted loop body, the process for constructing this is the same as before but with some small modifications.
- The sequential version of the loop is extracted to a new function and is removed from the original function unlike with non-nested loops where they remain in the function.
- The `BasicBlock` successors to the nested-loop exits are extracted to a new function.

The entry to the nested loop in the original function is replaced with a call to `__enqueue_task()` followed by a `ret` instruction. Figure 7.1 Demonstrates the end result of `LoopExtractionPass` on a loop with a nested loop.

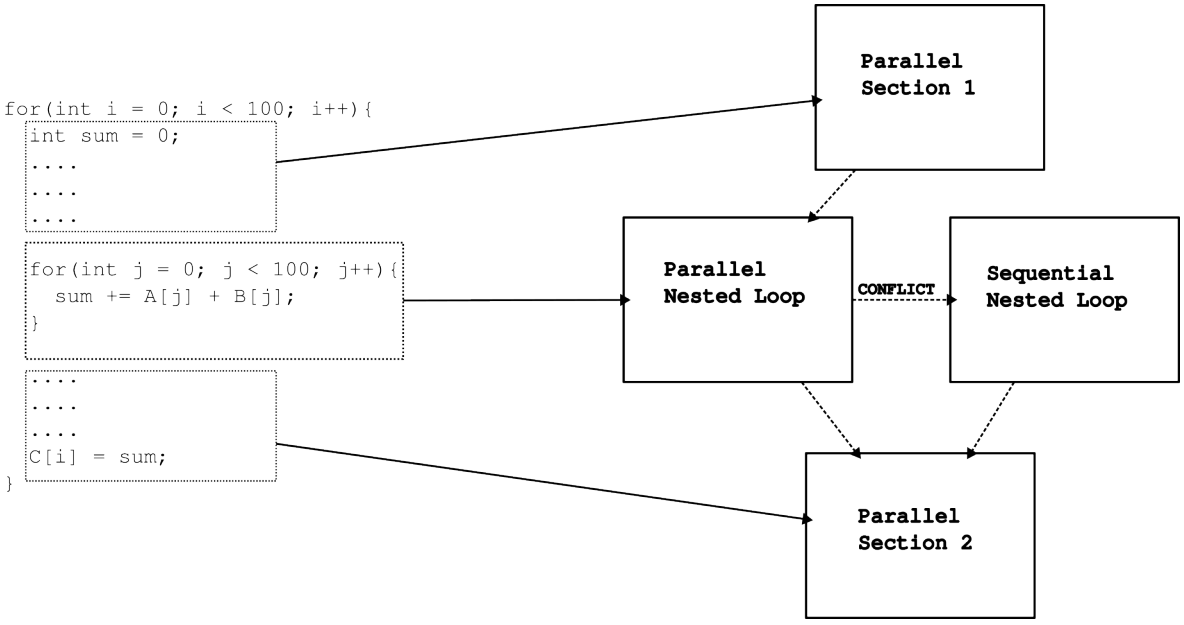


Figure 7.1: How `LoopExtractionPass` transforms a loop with a nested loop. The dotted arrows show the control flow of the program on execution. Each solid box is a function.

7.3 Executing Nested Loops and Synchronising Jobs

When the library receives a call to `__enqueue_task()`, a check is performed to detect if the enqueued parallel section is a nested loop or not. For non-nested loops, the calling thread sleeps while it waits for the parallel section to return a value, but for nested loops, the calling thread doesn't wait, so it can continue its dequeue cycle. This motivates transforming the loop into multiple functions, as we avoid idle threads.

To correctly handle nested loops, the following parameters were added to `__enqueue_task()`:

- A function pointer to the sequential version of the loop. This is called after rollback by the thread library when the current `Job` is cancelled due to a conflict.
- A function pointer to the successors of the nested loop's exit. The thread library calls this function after the parallel section has been successfully executed or after the sequential version of the nested loop has been executed successfully.

This correctly handles the control flow of the original function. A `Job` now changes to a slightly weaker definition, instead of being a parallel execution of a complete loop body it's a parallel execution of a section of a loop. Combined with the fact that a nested-loop body is treated as a new `Job`, this implicitly us with the synchronisation behaviour we wanted. As a given thread waits for the current `Job` to be cancelled or completed, the nested loop's `Job` isn't started until the section before its call is fully finished. This effectively treats the entry to the nested section as a barrier.

When a `Job` calls `__enqueue_task()`, the `ThreadPool` keeps track of all the `Task`s that produced the call. When the nested loop execution finishes, the loop's successor function is called by creating new `Task`s with the same properties as the `Task`s that called the nested loop (e.g. their induction variable.) This is so we replicate the behaviour of resuming the outer loop rather than treating it as a new function body.

Rollback and conflict detection for nested loops is quite trivial as a `JobState` can be shared between `Job`s. `JobState`s handle the conflict detection and rollback for a specified scope (but not including nested scopes.) This means that the nested loop has its own `JobState` and the outer loop has a separate `JobState`. When the execution of a parallel section finishes successfully, or sequential execution of a nested loop begins, the `Job` is provided with the outer loop's `JobState`, which preserves all read and write history from before the nested section and includes the rollback

information in case a conflict exists.

To illustrate how this works, here is a small example loop:

```
for( int i = 0; i < 100; i++ ){
  int sum = 0; // This runs with JobState A

  for(int j = 0; j < 100; j++){
    sum += A[j] + B[j]; // This runs with JobState B
  }

  C[i] = sum; // This runs with JobState A

  for(int j = 0; j < 100; j++){
    sum += D[j] + E[j] // This runs with JobState C
  }

  F[i] = sum; // This runs with JobState A
}
```

Each time a new scope is entered, a new `JobState` is created, and when the program returns to the previous scope the `JobState` of the previous scope is used instead.

7.4 Preserving Scopes

One issue introduced by this approach is the destruction of stack-allocated variables upon executing `ret`, and these values need to be preserved to be used by both the nested loop and the rest of the loop body after the nested loop. These values are found in the same way as finding external uses for instructions and arguments. Values are preserved by creating a `__malloc()` call to the thread library, telling the library how much memory to allocate for the value. This address is then passed into the nested loop through the list of pointers given to `__enqueue_task()`. Freeing this memory is handled by the thread library at the end of a parallel execution.

Another consequence of this approach is that two lists of arguments are required: one for the nested loop, but another for the scope of the nested loop's successor (as the start of the parallel section could declare variables used after the nested loop.) This is added as another parameter to the call to `__enqueue_task()`.

7.5 Conclusion

This chapter discussed the complications introduced by nested loops and how the thread library and LLVM pass have successfully handled them. The next chapter will investigate the impact of the overall implementation on a program's execution.

Chapter 8

Evaluation

This chapter evaluates the performance impact that the implementation of the thread library has on a program's execution time.

8.1 Theoretical Impact on Performance

The maximum theoretical speed-up of a program with a fixed amount of work is governed by Amdahl's law:

$$S = \frac{1}{(1 - P) + \frac{P}{N}} \tag{8.1}$$

where:

S is the speedup of the program using N processors
P is the proportion of the program that can be made parallel
N is the number of processors

In this equation, P is the proportion of the program spent in loops that `LoopExtractionPass` detects, simplifies, and extracts for the thread library to execute concurrently. Figure 8.1 demonstrates how the speed-up of a program is ultimately limited according to how much of the program can be parallelised.

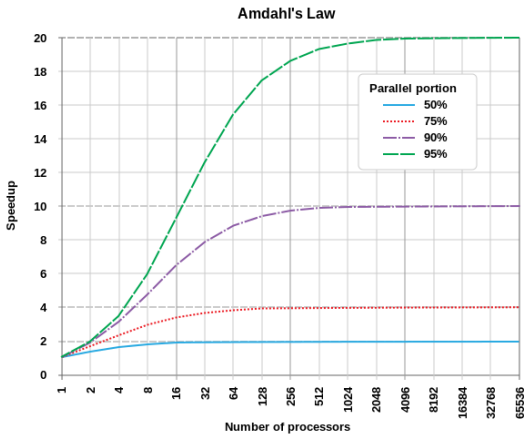


Figure 8.1: The speed-up of a program calculated using a variable number of processors according to the parallel portion of the program, provided by [6]

8.2 Micro-benchmarks

8.2.1 Evaluation Methodology

To evaluate the performance of the library, deterministic micro-benchmarks are compiled into object files with `-O3` using `Clang v18.1.2` and a modified version of `Clang v18.1.2` which has `LoopExtractionPass` and `InstrumentFunctionPass` statically linked into the pipeline as plugin passes. As the same version of `Clang` is used, the only difference between the speculative-parallel version of the benchmark and the non-speculative version is the addition of the passes. This produces two object files, the one produced by the modified `Clang` gets dynamically linked with our thread library. The `volatile` keyword is sometimes used in the benchmarks to prevent the compiler from performing certain optimisations, such as constant propagation and dead code elimination.

This provides us with two binaries executed and timed using a Python script (generated by ChatGPT). The script runs each binary 10 times and measures the execution time. This is then averaged and the standard deviation for each case is calculated. The thread library is set to use 4 threads for the parallel execution, as this lowers the thread contention in the thread library. Whilst this is an important aspect of the library that can impact execution time, runtime conflict detection dominates the execution time and thread contention is negligible.

8.2.2 Verifying Correctness

Deterministic benchmarks provide a means of differential testing; the correctness of the implementation can be verified by comparing the final output of the speculative-parallel version of the program and the non-speculative version, which is important in all benchmarks (even without conflicts). Verifying benchmarks with conflicts ensures that the conflict detection has no false negatives and that the rollback mechanism works as intended.

False positives, while they shouldn't exist, are not tested for, as in the worst-case they impact performance (since the rollback mechanism is activated when it shouldn't be) but do not affect correctness as conflict benchmarks verify that the rollback works correctly.

8.2.3 Non-conflict Benchmark Results

Benchmark	Spec-Parallel Execution Time (s)	Standard Deviation (s)	Non-Speculative Execution Time (s)	Standard Deviation (s)
no-conflict	0.0031	0.0002	0.0006	0.0000
no-conflict-2	0.0047	0.0001	0.0006	0.0000
no-conflict-3	0.0046	0.0002	0.0006	0.0000
no-conflict-4	0.0014	0.0001	0.0006	0.0000
no-conflict-5	0.0322	0.0006	0.0006	0.0000
no-conflict-6	0.0494	0.0005	0.0006	0.0001
no-conflict-7	0.0316	0.0003	0.0009	0.0001
call	0.0052	0.0002	0.0006	0.0000
matrix	2.7614	0.0529	0.0020	0.0000

Table 8.1: Average Execution Time and Standard Deviation for Benchmarks without Conflicts

Table 8.1 demonstrates that speculative-parallel benchmarks are significantly slower than the unmodified benchmarks. The extent of the slow-down varies widely according to the nature of the benchmark. Benchmarks that are very write-heavy are significantly slower than other benchmarks.

In the worst-case, `matrix`, which is simple matrix multiplication, is more than 1000x slower than the unmodified version, due to stores and loads dominating the workload. In other benchmarks where there are fewer stores and loads, such as `no-conflict-5`, `no-conflict-6`, and `no-conflict-7`, the speculative-parallel version is only a 50x slowdown. However, this is still significant. This observation is further testified by benchmarks with few stores and loads, `no-conflict`, `no-conflict-2`,

and `no-conflict-3`, which result in around a 5-8x slowdown.

`no-conflict-4` is unique as it's a benchmark with no stores or loads, only a single call to `printf` each iteration. Here only a 2x slowdown is encountered. This is a result of the overhead from the `ThreadPool` implementation including creating and adding `Task`s and `Job`s to the execution, as well as clean-up operations and minor thread contention.

8.2.4 Conflict Benchmark Results

Benchmark	Spec-Parallel Execution Time (s)	Standard Deviation (s)	Non-Speculative Execution Time (s)	Standard Deviation (s)
conflict	0.0016	0.0002	0.0005	0.0000
conflict-2	0.0014	0.0001	0.0006	0.0000
conflict-3	0.0024	0.0002	0.0007	0.0001
conflict-4	0.1262	0.0026	0.0007	0.0000
conflict-5	0.1255	0.0013	0.0007	0.0000
conflict-6	0.0050	0.0002	0.0005	0.0001
conflict-7	0.0191	0.0002	0.0006	0.0000

Table 8.2: Average Execution Time and Standard Deviation for Benchmarks with Conflicts Over 10 Runs

Table 8.2 shows the results of the benchmarks with conflicts, and further consolidates the conclusions found in the non-conflict tests; conflict detection is the leading cause of the slow-down.

`conflict`, `conflict-2`, and `conflict-3` are benchmarks where the dependency conflicts can be found very early in the parallel execution. Therefore the conflict-detection cancels the `Job` before too much work can be wanted by the rollback. This leads to the program re-running the parallel sequentially, the program execution is within the range of a 2-3x slow-down of the non-speculative version. This slow-down is accounted for by the roll-back mechanism, the conflict checking leading up to the violation, and the previously mentioned overheads with the `ThreadPool` itself.

`conflict-4` and `conflict-5` are benchmarks where violations are detected later in the parallel execution, due to them existing within a nested loop. This means that, before the violation is detected, the execution of the outer loop up to the point of the nested loop has fully completed due to the nested loop entry acting as a barrier. This is why these benchmarks have slowdowns over 100x, as the thread library fully executes a `Job`, discovers the violation in the nest, rolls back, and executes the loop sequentially. The same behaviour is observed from `conflict-6` and `conflict-7`.

8.3 SciMark 2.0

SciMark 2.0 is a benchmark suite focussing on calculations and algorithms in scientific and numerical workloads such as matrix multiplication and Monte Carlo simulations. Typically, these workloads are parallel and provide an opportunity for speculative parallelisation.

8.3.1 Evaluation Methodology

Similar to the micro-benchmarks, two binaries are compiled: one with unmodified `Clang` and the other with `Clang` where our passes are statically linked into the pipeline. `-parallel` is removed and `-par-num-threads` is set to 1 for both binaries. Although this impacts the score of the benchmark, the relative performance of a single-threaded implementation compared to running the implementation with speculative parallelization is what is under investigation. This simulates the scenario mentioned in the introduction, where a developer isn't aware of a parallel implementation, and represents a realistic use-case for speculative parallelisation. Using a Python script (generated by ChatGPT), `scimark` is executed with `-large` option 10 times for each binary, and the average results for each benchmark are calculated. `scimark` and its source code are not modified to support speculative parallelisation, only the compile flags are changed.

8.3.2 Verifying Correctness

`scimark` provides a checksum that can be compared between separate runs to verify correctness. If the checksums of the non-speculative execution and the speculative-parallel executions are the same, then the execution of the speculative-parallel version is correct. This is checked and verified at the end of all runs.

8.3.3 SciMark Results

Benchmark	Average MFLOPS (Spec-Parallel)	Average MFLOPS (Default)
FFT	37.783	590.203
SOR	2350.335	2352.427
Monte Carlo	638.202	640.294
Sparse matrix multiply	5233.171	5808.228
LU	1225.238	9768.093
Composite	1896.946	3831.849

Table 8.3: Average SciMark 4 Benchmark results over 10 runs

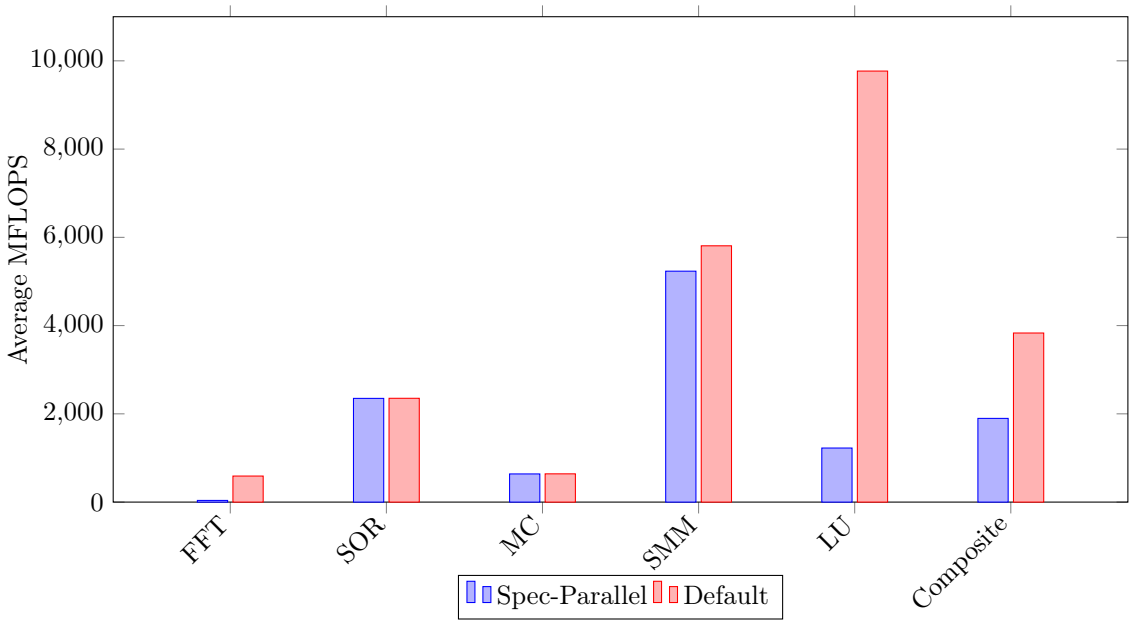


Figure 8.2: Average SciMark 4 Benchmark results over 10 runs

Some benchmarks are close to the unmodified binary; Jacobi Successive Over-relaxation (SOR) and Monte Carlo are within margin-of-error. The SOR implementation contains a loop with a loop-carried dependency, so the loop is executed sequentially, explaining why the two results are so close. This also demonstrates that conflict detection detects loop dependencies very quickly when they exist. SciMark states that Monte Carlo "exercises random-number generators, synchronized function calls, and function inlining"[14], suggesting that conflicts exist for this benchmark in non-synchronized sections. Hence, the unmodified and speculative-parallel binaries are similar in performance.

Sparse matrix multiply (SMM) also isn't too far off from the unmodified binary. This is likely because of the sparseness of the matrix, so we don't perform writes as often, which reduces the impact of conflict detection in two ways: fewer writes are recorded in `AddrHistory` so WAR violations are less expensive to detect (as this checks the history of stores for each load), and fewer stores means fewer calls to conflict-detection.

This leaves two benchmarks with significantly reduced performance: Fast Fourier Transform (FFT)

and dense LU matrix factorization (LU). A conflict is not reached here and so the entire workload is performed as a parallel execution, demonstrating the performance impact of frequent run-time conflict detection when implemented in software.

The composite score is aggregated from the results of all benchmarks for an overall score. The speculative-parallel composite score is almost half the composite score of the modified binary.

8.4 Conclusion

In conclusion, the evaluation demonstrates that run-time conflict detection is costly and impractical to implement in software to achieve speculative parallelisation. However, from the standard deviation results in the conflict benchmarks, the library detects conflicts consistently early in the parallel execution. The success of the conflict-detection mechanism is also shown in the SOR and Monte Carlo benchmarks in SciMark 4.0, where dependency violations are detected so early that the overall impact on performance is minimal.

This result demonstrates that if the conflict detection can be accelerated in some way by hardware then the slow-down could be significantly reduced. If the `ThreadPool` implementation could also be expressed in hardware, perhaps through a clustered processor, then this would also reduce the overheads from the thread library. The software-only implementation only produces a 2x slow-down on average in scientific tasks, suggesting that a hardware-accelerated implementation would improve performance rather than hinder it.

In hardware, memory addresses could be marked by a timestamp when loaded and unmarked when stored, which allows us to implement conflict detection when un-marking the memory address. However, the hardware would also have to store the logs of stores and loads to each address in some way. The hardware could only look at some bits from an address to reduce space requirements, but this reduces the precision of the conflict detection as multiple memory addresses could share a log throughout a parallel execution. This could be a performance concern depending on how much information is lost using this method because of more frequent, and unnecessary, rollbacks. Correctness is not affected as it only introduces false positives.

This project can inform the future design of speculative-parallel capable processors, and the design of dedicated hardware to accelerate conflict detection at runtime.

Chapter 9

Conclusion

This project successfully implements a run-time conflict-detection and rollback mechanism by creating LLVM passes to outline loops and instrumenting them with calls to an external library. This project preserves program correctness in the case of a conflict by successfully overwriting modified memory addresses with their original value from before we started the program execution. Conflicts are consistently detected early in the outlined loops if they exist, preventing too much work from being wasted by the processor before we rollback. In cases where we do rollback in the benchmarks, particularly in SOR and Monte Carlo simulations, the cost of the rollback is dominated by the workload itself, resulting in minimal impact on the final results of the benchmark.

Although only a 2x slow-down is experienced on average across scientific workloads, the software implementation of this system results in significant overheads with slow-downs of 1000x in the worst case. An investigation into this system's hardware or hardware-accelerated implementation could overcome these issues and provide a practical solution to utilise parallelism that modern compilers cannot detect. This project lays the groundwork for future improvements to the library that, whilst it's unlikely that all slow-down can be eliminated, provide opportunities to experiment with different conflict detection techniques that could result in less overhead and translate to realised benefits in a potential hardware implementation.

9.1 Limitations

The wide scope of this project and its restricted timeline lead to several limitations, particularly in the LLVM pass, with the main impact of these limitations resulting in fewer loops being executed in parallel. This directly affects Amdahl's law and lowers the theoretical parallelism limit.

9.1.1 Unbounded Loops

The LLVM pass currently checks for bounds to each loop in `LoopSimplify` form, if the bounds cannot be determined at compile-time then the loop is skipped. This is because the generated call to `__enqueue_task()` contains parameters that are directly inserted and generated during `LoopExtractionPass`, therefore if we don't have the bounds at this stage we cannot create the call.

9.1.2 Loops With Multiple Exits and Exiting Blocks

Multiple exits complicate the control flow after a loop has finished a parallel execution, as information has to be relayed back to the original function through a return value from `__enqueue_task()` to tell it which exit block to branch to. Whilst this isn't impossible to solve, it significantly complicates the transformations required for the original function. For example, a switch statement could determine which `BasicBlock` to branch to after the parallel execution.

Multiple exiting blocks significantly complicate the thread library implementation. For example, take the following parallel execution:

1. A new `Job` is created to execute a parallel section with multiple exiting blocks.

2. One thread begins execution of an iteration that breaks out of the loop, whilst another thread begins execution of an iteration that is timestamped after the one that breaks.

3. Although the loop is supposed to break, the execution of the later iteration still occurs, along with all of its writes.

4. The parallel execution is finished, but the state resulting from it is invalid due to the extra iteration.

This could be solved by keeping track of the writes in a particular iteration, similar to how rollback works, so the `ThreadPool` can revert them in the case of a break in an earlier iteration. However, this is hard to do efficiently and otherwise would result in significant overhead.

9.2 Future Work and Optimisations

The project lays the groundwork for future improvements and optimisations, particularly to the thread library, but also to the LLVM pass.

9.2.1 Loop Strip Mining and Checkpointing

If a conflict is detected the entire parallel section and all its work is overwritten and the original program state is restored. However, by implementing loop strip mining the parallel execution can be checkpointed at the end of each strip. Take the following loop:

```
for( int i = 0; i < 100; i++ ){
  // Some instruction that causes conflicts between i=85 and i=89
}
```

With loop strip mining, this loop could be transformed by the LLVM pass to be turned into:

```
for (int i = 0; i < 100; i += 5) {
    for (int j = i; j < i + 5; j++) {
        // Some instruction that causes conflicts between i=85 and i=89
    }
}
```

In the first version of the loop, when the conflict is reached all of the work done by the loop is discarded when the `ThreadPool` calls the rollback mechanism. However, with the strip-mined loop, the rollback mechanism is triggered on the nested loop, which is at most 5 iterations of work in this example. Conflicts don't tend to appear so late in a loop, so it's uncertain how much of an improvement this would make in general, but in specific cases, this could significantly improve execution time.

Although the previous benefit only concerns loops with conflicts, another consequence of this is faster conflict detection; as only the last 5 iterations need to be checked for conflicts, this significantly reduces the number of `Timestamps` in the store and load history of `AddrHistory`. Therefore loop strip mining would improve the performance of all parallel executions, not just those with conflicts.

9.2.2 Asynchronous Conflict Detection

The current conflict-detection mechanism is executed on the same thread as the execution of a `Task`, which means that the parallel execution "blocks" until conflict detection is complete. This also means that thread contention from conflict detection also leaks into the program's execution time. However, conflict detection can run in parallel to the parallel execution itself, but for this to work two conditions must hold: conflict detection is performed in the same order as the stores/loads occur in the parallel execution, and all conflict detection jobs must be finished before the original execution of the program resume. This could lead to a considerable increase in performance.

9.2.3 Counting Conflict Checks and Measuring Rollback Impact

To further investigate the performance impact of conflict detection, the number of conflict checks can be counted per execution. This opens multiple avenues of investigation: investigating the difference between the cost of a load and store conflict check, finding ways to minimise unnecessary conflict checks, and finding concrete correlations between conflict checking frequency and software performance. As shown by the results in the previous chapter, the rollback mechanism is fast and doesn't result in an unfavourable performance impact, but its performance characteristics could be investigated further to find what its cost depends on.

A compiler could use these metrics to evaluate if a loop would benefit from speculative parallelisation before the software is executed using a cost-benefit formula (similar to what LLVM already does for vectorization.)

9.2.4 Hardware-Accelerated/Implemented Conflict Detection

Conflict detection is the main contributor to the slowdowns experienced in our benchmarks, so accelerating this with hardware, or directly implementing it in hardware, is key to achieving performance benefits from speculative parallelisation. A CPU with conflict-detection integrated into its hardware, and with some logic for queueing the tasks based on `Timestamps` like the `ThreadPool` does, can avoid the overheads that the thread library in software produces such as thread lock contention. Significantly faster conflict-detection could be achieved by implementing its data structures in hardware (e.g. by using bloom filters rather than `std::set`). Clustered processors with multiple threads per core could further improve performance, as we can spawn many threads for a single execution without the overhead of a software thread (such as OS scheduling delays).

Loop Heuristics

A hardware implementation of the library could also include heuristics that keep track of a parallel execution's previous successes and failures in the program. This could help the CPU avoid wasting cycles on a loop that always contains a conflict and will cancel at some point during its parallel execution. However, the CPU could also collect information about other loop patterns at runtime, such as strides in memory accesses, to evaluate whether a loop should be executed with speculative parallelisation even if it has failed previously. While this would introduce overhead in a software implementation, this could be done efficiently in hardware.

9.3 Conclusion

With future work built on the implementation detailed by this report, speculative parallelisation can be implemented by hardware to avoid the overheads that hinder the software implementation. This could result in performance gains from speculative parallelisation, with no developer interaction required.

Bibliography

[1] TutorialsPoint. *Compiler Design - Phases of Compiler*. URL https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm. [Accessed 16th January 2024].

[2] Shun-Tak Leung and John Zahorjan. Improving the performance of runtime parallelization. *SIGPLAN Not.*, 28(7):83–91, 1993. doi: 10.1145/173284.155341.

[3] Wikipedia. *Polytope model - Wikipedia*. URL https://en.wikipedia.org/wiki/Polytope_model. [Accessed 23rd January 2024].

[4] L. Rauchwerger and D.A. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999. doi: 10.1109/71.752782.

[5] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism In:. In *Proceedings of the 48th International Symposium on Microarchitecture*, page 228–241, New York, NY, USA, 2015. doi: 10.1145/2830772.2830777.

[6] Daniels220. *Amdahl's Law - Wikipedia*. URL https://en.wikipedia.org/wiki/Amdahl%27s_law. [Accessed 10th June 2024].

[7] LLVM Project. *LLVM Language Reference Manual*, 2024. URL https://llvm.org/docs/LangRef.html. [Accessed 18th January 2024].

[8] LLVM Project. *LLVM Loop Terminology (and Canonical Forms) – LLVM 18.0.0git documentation*, 2024. URL https://llvm.org/docs/LoopTerminology.html. [Accessed 3rd June 2024].

[9] J.H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991. doi: 10.1109/12.88484.

[10] Manuel Arenaz, Juan Touriño, and Ramón Doallo. In Jiannong Cao, Laurence T. Yang, Minyi Guo, and Francis Lau, editors, *An Inspector-Executor Algorithm for Irregular Assignment Parallelization In:*, pages 4–15, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[11] Bruno Chinelato Honorio, João P. L. de Carvalho, Munir Skaf, and Guido Araujo. Using OpenMP to Detect and Speculate Dynamic DOALL Loops In:. In Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg, editors, *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, pages 231–246, Cham, 2020. Springer International Publishing.

[12] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-Polyhedral optimization in LLVM in:. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

[13] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. *SIGARCH Comput. Archit. News*, 28(2):1–12, 2000. doi: 10.1145/342001.339650.

[14] Roldan Pozo and Bruce R Miller. *Java SciMark 2.0 (About)*. URL https://math.nist.gov/scimark2/about.html. [Accessed 13th June 2024].