Automatic Parallelisation of Rust Programs at Compile Time

Author

Michael Oultram

1428105

Degree

MSci in Computer Science

8th March 2018

Supervisor

Dr Ian Batten

Institution

School of Computer Science

University of Birmingham

Contents

Chapter 1

Preamble

1.1 Abstract

TODO: Write Abstract

All software for this project can be found at `https://github.com/MichaelOultram/FYP`

1.2 Acknowledgements

Rust Compiler (`https://github.com/rust-lang/rust`)

Serde (`https://serde.rs/`): used to convert rust objects into JSON and back again

Chapter 2

Introduction

Processors are being released with more cores. Sequential code cannot take advantage of these new cores. Writing parallel code is more difficult than writing sequential code. Existing sequential code would need to be rewritten to be parallelised. Ideally we want to gain the benefits of parallel code, whilst only having to write easy sequential code.

One solution to this problem is a parallelising compiler. Research done in this field has mostly focused on the C/C++ language although other researches have had success using other languages. Some methods require manual annotation of the source code by the programmer to specify which parts of the program are parallelisable. Others have attempted to automatically detect these areas, but with an unsafe language like C++ it is challenging.

For my project, I focused on the safe language rust. Rust has a unique way of managing memory such that only one thread can access the memory space at once. This is guaranteed at compile time, and should make the process of automatically detecting dependencies much easier. Rust also allows plugins into the compiler (nightly feature only as of writing) which gives modification access to the abstract syntax tree.

2.1 Rust Language Features

Rust is similar to other programming languages such as C++ but it does has some specific features that may not be known to the reader. This section briefly explains features of the language that are used in later sections of the report. If the reader requires more in depth understanding than what is provided, then they should look at the language documentation (*The Rust Book* 2017).

2.1.1 Safety Features

"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety" (*The Rust Programming Language* 2017). To get these safety properties, rust has some unique features. The biggest difference to other programming languages is how variable are handled.

Ownership

In Rust, all variables have an ownership. Only one block can have access to that variable at a time. This is enforced at compile time.

```
1  fn main() {
2      let a = 10;
3      f(&a);
4      g(a);
5      // Cannot access a here anymore
6  }
```

```
7   fn f(a: &u32){} // f borrows a
8   fn g(a: u32){} // g takes ownership of a
```

Listing 1: Borrowing and moving example

In this example, `a` is a local variable in the `main` method.

When `f` is called with parameter `a`, the function borrows that variable. This is similar to call-by-reference from other programming languages.

When `g` is called with parameter `a`, the variable is moved to `g`. This is unlike other programming languages as this is not call-by-value. Instead `g` takes ownership of `a`. When `g` is returned, the main method can no longer use `a`.

Mutability

Variables mutability is declared when the variable is declared. In rust, variables are immutable by default, but if specified they are mutable. When a variable is borrowed, it can either be immutably borrowed or mutably borrowed.

```
1   fn main() {
2       let a = 10;
3       let mut b = 20;
4       f(&a, &b);
5       g(&mut b);
6   }
7   fn f(a: &u32, b: &u32){} // f immutably borrows a and b
8   fn g(b: &mut u32){} // g mutably borrows b
```

Listing 2: Immutable and mutable borrowing

In the `main` method of this example, `a` is an immutable local variable and `b` is a mutable local variable. The `f` function borrows both `a` and `b` immutably. Even though `b` is declared as mutable, it cannot be changed inside `f`. Once `f` returns, `b` becomes mutable again inside the `main` method. The `g` function shows how `b` can be borrowed mutably.

Unsafe Blocks

The programmer may want can turn off some of rust's safety features by using an unsafe block. The most common use of an unsafe block is to modify a mutable static variable but it also allows de-referencing of a raw pointer and calling unsafe functions (i.e. an external c function).

2.1.2 Threads

Moving

Overhead

2.1.3 Crates

Chapter 3

Design

TODO: Explain rust compiler plugin types. Why we compile twice, etc. TODO: Justify all decisions. Explain alternatives considered/implemented and why the design changed

The rust compiler allows for plugins of different types. The two types of plugins used are Syntax Extension plugin and a Linter Plugin. Syntax Extension plugins are run first, and are generally used to convert macros into code. I will use a syntax extension to change the entire body of a function from sequential code into parallelised code. Linter plugins run after all the syntax extension plugins, and are generally used to check code style to produce warnings (like unused variable).

The linter plugin will have the abstract syntax tree of the code with all the macros expanded. This stage has all the information required about dependencies. However, once the rust compiler gets to the linter plugins, I can no longer edit the code (without recompiling the compiler). The solution I decided on was to compile the program twice. On the first compile, the syntax extension does nothing and the linter plugin examines the expanded code. The dependency information gathered is saved into a file for the next compile. On the second compile, the syntax extension plugin reads the file to get all the dependency information. Any parallelisable parts are modified to be run in parallel.

The rust plugin system requires an annotation to be able to access that element of the code. Each function of the sequential source code should be annotated with `#[autoparallelise]`. The AST is described by three types of structs: Block, Stmt and Expr. A block contains a list of statements, and each statement is a combination of expressions. Macros can take arguments and are transformed into code. This transformation happens in the compiler after executing all the syntax extension plugins. Each function is evaluated separately.

3.1 Dependency Analysis

The dependency analysis stage is split over two compiles. The first compile uses the linter plugin, and the second uses the syntax extension plugin.

3.1.1 Linter Plugin

The block of the sequential function is examined statement by statement. Each statement is converted into our representation of the AST so that information can be stored about the dependencies. The extra information includes two environments containing which variables the statement requires and produces.

Once all the statements for a function are converted into our representation, the dependency environment need to be matched up to statement ids. Each converted statement is looked at in turn, starting from the beginning of the function. The algorithm looks backwards from the current point in the function to find what statements produce the variables in the requires

environment. The statement ID relative to the block of all of the dependencies is stored as part of the converted statement.

It is possible for a block to be inside a block by being represented as a Statement. Blocks need to be evaluated using the same method as explained in the previous paragraph. A block statement is represented as a list of converted statements, as well as the blocks own environment and dependency ids. It is also possible for a new block to be part of another expression (i.e. for loops). In this case, this is stored as an ExprBlock with the statement that the block originates from, a subtree containing a Block statement (there could be more than one block) and the environments/dependency ids.

Once all functions have been analysed, the DependencyTree is converted into an EncodedDependencyTree. The code part of the converted statement is converted into a statement id. The statement ID is represented as a pair of numbers `(span.lo().0, span.hi().0)`, which relate to the byte location of the source code. This will remain consistent between compile runs, whereas the NodeID does not. All EncodedDependencyTrees and function meta data is stored into a JSON file using `serde_json`.

3.1.2 Syntax Extension Plugin

The plugin detects the JSON file and loads it. This is stored as a shared state between different functions.

The first part of the dependency analysis is repeated for the syntax extension plugin this time. In later section of the design, we need access to the pure AST. There is no (easy) way for the AST to be store into a JSON file and recreated into structs that I could find. The reason that we use the linter plugin is so that we can see the dependencies hidden inside macros.

The dependencies are merged function by function from the shared state so that unexpanded macros get the missing dependencies. The dependencies of Statements that have the same StmtID are merged together.

3.2 Scheduler

Once the dependency analysis is complete, the scheduler takes the dependency tree and works out a schedule. All relative dependency ids are converted into StmtIDs. The idea around the scheduling algorithm is Maximum Spanning Trees. All statements that have no dependencies can be started at the very beginning. All remaining statements wait for all their dependencies to be put into the schedule. Once all the dependencies for a statement are added, this statement is selected as the next one to add to the schedule. As each statement requires all of its dependencies before it can be executed, it should be scheduled to run after the slowest dependency. This should minimise the amount of time that the statement has to wait for its dependencies. Synclines are created for all the remaining dependencies so that all dependencies are met. Each block gets its own schedule.

3.3 Reconstructor

All parts of the reconstructor algorithm takes part in the second compile.

Chapter 4

Implementation

4.1 Analysis Stage

Here is some text above

rust/src/libsyntax/visit.rs

```
34  pub enum FnKind<'a> {
35      /// fn foo() or extern "Abi" fn foo()
36      ItemFn(Ident, Unsafety, Spanned<Constness>, Abi, &'a Visibility, &'a Block),
37
38      /// fn foo(&self)
39      Method(Ident, &'a MethodSig, Option<&'a Visibility>, &'a Block),
40
41      /// |x, y| body
42      Closure(&'a Expr),
43  }
```

rust/src/libsyntax/ast.rs

```
489 pub struct Block {
490     /// Statements in a block
491     pub stmts: Vec<Stmt>,
492     pub id: NodeId,
493     /// Distinguishes between `unsafe { ... }` and `{ ... }`
494     pub rules: BlockCheckMode,
495     pub span: Span,
496     pub recovered: bool,
497 }
```

rust/src/libsyntax/ast.rs

```
781 pub struct Stmt {
782     pub id: NodeId,
783     pub node: StmtKind,
784     pub span: Span,
785 }
```

rust/src/libsyntax/ast.rs

```
815 pub enum StmtKind {
816     /// A local (let) binding.
817     Local(P<Local>),
818
819     /// An item definition.
820     Item(P<Item>),
821
822     /// Expr without trailing semi-colon.
823     Expr(P<Expr>),
824     /// Expr with a trailing semi-colon.
825     Semi(P<Expr>),
826     /// Macro.
827     Mac(P<(Mac, MacStmtStyle, ThinVec<Attribute>)>),
828 }
```

```
let a; // Local without init
a = {
    let b = vec![1,2,3]; // Local with init
    println!("{:?}", b); // Mac
    b.len() // Expr
}; // Semi
```

Listing 3: Example showing different StmtKinds

rust/src/libsyntax/ast.rs

```
pub enum ExprKind {
    /// A `box x` expression.
    Box(P<Expr>),
    /// First expr is the place; second expr is the value.
    InPlace(P<Expr>, P<Expr>),
    /// An array (`[a, b, c, d]`)
    Array(Vec<P<Expr>>),
    /// A function call
    ///
    /// The first field resolves to the function itself,
    /// and the second field is the list of arguments.
    /// This also represents calling the constructor of
    /// tuple-like ADTs such as tuple structs and enum variants.
    Call(P<Expr>, Vec<P<Expr>>),
    /// A method call (`x.foo::<'static, Bar, Baz>(a, b, c, d)`)
    ///
    /// The `PathSegment` represents the method name and its generic arguments
    /// (within the angle brackets).
    /// The first element of the vector of `Expr`s is the expression that evaluates
    /// to the object on which the method is being called on (the receiver),
    /// and the remaining elements are the rest of the arguments.
    /// Thus, `x.foo::<Bar, Baz>(a, b, c, d)` is represented as
    /// `ExprKind::MethodCall(PathSegment { foo, [Bar, Baz] }, [x, a, b, c, d])`.
    MethodCall(PathSegment, Vec<P<Expr>>),
    /// A tuple (`(a, b, c ,d)`)
    Tup(Vec<P<Expr>>),
    /// A binary operation (For example: `a + b`, `a * b`)
    Binary(BinOp, P<Expr>, P<Expr>),
    /// A unary operation (For example: `!x`, `*x`)
    Unary(UnOp, P<Expr>),
    /// A literal (For example: `1`, `"foo"`)
    Lit(P<Lit>),
    /// A cast (`foo as f64`)
    Cast(P<Expr>, P<Ty>),
    Type(P<Expr>, P<Ty>),
    /// An `if` block, with an optional else block
    ///
    /// `if expr { block } else { expr }`
    If(P<Expr>, P<Block>, Option<P<Expr>>),
    /// An `if let` expression with an optional else block
    ///
    /// `if let pat = expr { block } else { expr }`
    ///
    /// This is desugared to a `match` expression.
    IfLet(P<Pat>, P<Expr>, P<Block>, Option<P<Expr>>),
    /// A while loop, with an optional label
    ///
    /// `'label: while expr { block }`
    While(P<Expr>, P<Block>, Option<SpannedIdent>),
    /// A while-let loop, with an optional label
    ///
    /// `'label: while let pat = expr { block }`
    ///
    /// This is desugared to a combination of `loop` and `match` expressions.
    WhileLet(P<Pat>, P<Expr>, P<Block>, Option<SpannedIdent>),
    /// A for loop, with an optional label
    ///
    /// `'label: for pat in expr { block }`
    ///
    /// This is desugared to a combination of `loop` and `match` expressions.
    ForLoop(P<Pat>, P<Expr>, P<Block>, Option<SpannedIdent>),
    /// Conditionless loop (can be exited with break, continue, or return)
```

```
1049        ///
1050        /// `'label: loop { block }`
1051        Loop(P<Block>, Option<SpannedIdent>),
1052        /// A `match` block.
1053        Match(P<Expr>, Vec<Arm>),
1054        /// A closure (for example, `move |a, b, c| a + b + c`)
1055        ///
1056        /// The final span is the span of the argument block `|...|`
1057        Closure(CaptureBy, P<FnDecl>, P<Expr>, Span),
1058        /// A block (`{ ... }`)
1059        Block(P<Block>),
1060        /// A catch block (`catch { ... }`)
1061        Catch(P<Block>),
1062
1063        /// An assignment (`a = foo()`)
1064        Assign(P<Expr>, P<Expr>),
1065        /// An assignment with an operator
1066        ///
1067        /// For example, `a += 1`.
1068        AssignOp(BinOp, P<Expr>, P<Expr>),
1069        /// Access of a named struct field (`obj.foo`)
1070        Field(P<Expr>, SpannedIdent),
1071        /// Access of an unnamed field of a struct or tuple-struct
1072        ///
1073        /// For example, `foo.0`.
1074        TupField(P<Expr>, Spanned<usize>),
1075        /// An indexing operation (`foo[2]`)
1076        Index(P<Expr>, P<Expr>),
1077        /// A range (`1..2`, `1..`, `..2`, `1...2`, `1...`, `...2`)
1078        Range(Option<P<Expr>>, Option<P<Expr>>, RangeLimits),
1079
1080        /// Variable reference, possibly containing `::` and/or type
1081        /// parameters, e.g. foo::bar::<baz>.
1082        ///
1083        /// Optionally "qualified",
1084        /// E.g. `<Vec<T> as SomeTrait>::SomeType`.
1085        Path(Option<QSelf>, Path),
1086
1087        /// A referencing operation (`&a` or `&mut a`)
1088        AddrOf(Mutability, P<Expr>),
1089        /// A `break`, with an optional label to break, and an optional expression
1090        Break(Option<SpannedIdent>, Option<P<Expr>>),
1091        /// A `continue`, with an optional label
1092        Continue(Option<SpannedIdent>),
1093        /// A `return`, with an optional value to be returned
1094        Ret(Option<P<Expr>>),
1095
1096        /// Output of the `asm!()` macro
1097        InlineAsm(P<InlineAsm>),
1098
1099        /// A macro invocation; pre-expansion
1100        Mac(Mac),
1101
1102        /// A struct literal expression.
1103        ///
1104        /// For example, `Foo {x: 1, y: 2}`, or
1105        /// `Foo {x: 1, .. base}`, where `base` is the `Option<Expr>`.
1106        Struct(Path, Vec<Field>, Option<P<Expr>>),
1107
1108        /// An array literal constructed from one repeated element.
1109        ///
1110        /// For example, `[1; 5]`. The first expression is the element
1111        /// to be repeated; the second is the number of times to repeat it.
1112        Repeat(P<Expr>, P<Expr>),
1113
1114        /// No-op: used solely so we can pretty-print faithfully
1115        Paren(P<Expr>),
1116
1117        /// `expr?`
1118        Try(P<Expr>),
1119
1120        /// A `yield`, with an optional value to be yielded
1121        Yield(Option<P<Expr>>),
```

```
1122    }
```

rust/src/libsyntax_pos/lib.rs

```
143    pub struct SpanData {
144        pub lo: BytePos,
145        pub hi: BytePos,
146        /// Information about where the macro came from, if this piece of
147        /// code was created by a macro expansion.
148        pub ctxt: SyntaxContext,
149    }
```

Here is some text below **??**

4.2 Modification Stage

Chapter 5

Evaluation

Chapter 6

Discussion

TODO: Write Discussion

Chapter 7

Conclusion

TODO: Write conclusion

Chapter 8

Appendix

8.1 Submission File Structure

TODO: Write section

8.2 Running the Code

Install rustup:
`curl https://sh.rustup.rs -sSf | sh`

Install rustc 1.25.0-nightly (0c6091fbd 2018-02-04) compiler:
`rustup install nightly-2018-02-04`
`rustup default nightly-2018-02-04`

Verify the version is correct using:
`rustc --version`

Create a new crate and write sequential code:
`cargo init`

Under `[dependencies]` in `Cargo.toml`, add:
`auto_parallelise = { version = "0.1.0", git = "https://github.com/MichaelOultram/FYP/", branch = "release" }`

At the top of your `lib.rs` or `main.rs` file, add:
```
#![feature(plugin)]
#![plugin(auto_parallelise)]
```

At the top of every function, add:
`#[autoparallelise]`

Compile the code twice using (the first time will fail):
`cargo build --release`

Run the parallelised code using:
`cargo run --release`

8.3 References

The Rust Book (2017). URL: `https://doc.rust-lang.org/book/` (visited on 12/11/2017).
The Rust Programming Language (2017). URL: `https://www.rust-lang.org` (visited on 12/11/2017).

Automatic Parallelisation of Rust Programs at Compile Time

Project Proposal

1 Problem

Kish (2002) estimated the end of Moore's Law of miniaturization within 6-8 years or earlier (based on their publication date) and as such, manufacturers have been increasing processors' core count to increase processor performance (Geer 2005). Writing parallelised programs to take advantage of these additional cores has some difficulty and often requires significant changes to the source code. Is it possible to automate these changes to convert sequential code into parallelised code? Previous attempts at solving this problem include D'Hollander, Zhang and Wang (1998) where they automated parallelisation of sequential FORTRAN code and Baskaran, Ramanujam and Sadayappan (2010) where they automated conversion of sequential C into CUDA code. Both of these approaches use unsafe programming languages significant complexity to their solutions. Instead, can this problem be solved more easily with a safe programming language like rust (*The Rust Programming Language* 2017)?

2 Approach

My approach involves writing a plugin for the rust compiler. The rust compiler plugin system allows for different types of plugin, which are run at different stages of the compilation. A syntax extension plugin can manipulate the abstract syntax tree of any annotated function, one function at a time. A linter plugin can see the abstract syntax tree of every function without annotations but it cannot manipulate anything.

This problem requires read access to the entire abstract syntax tree, and then modify access to the parallelisable parts. Due to the order of execution, the syntax extension plugin would be executed first and then the linter plugin but I would require them to be executed in the opposite order. To solve this problem, I propose that the program is compiled twice.

On the first compilation, the syntax extension plugin would do nothing. The linter plugin would view the entire abstract data tree and analyse what each statement depends on and modifies. This would create a dependency tree, where any two statements that are independent can be run in parallel. The linter plugin would use this dependency tree to determine which parts should be parallelised, and save this information to a file.

On the second compilation, the syntax extension would be able to read the file the linter plugin created on the previous compilation. This lists all the changes required and the syntax plugin can apply those changes to the abstract syntax tree function by function. The linter plugin would also be able to view this file, and it could produce compiler warnings for any function that could be parallelised that is missing an annotation.

2.1 Requirements

I believe that I have access to all the required software and hardware for this project. Below is listed most of what I am planning to use:

- A computer to program on
- Source code of some rust programs to parallelise
- Rust compiler (and it's source code)
- A text editor to write code in (Atom)

2.2 Estimated Timeline

Estimated End Date	**Milestone**
30/10/2017	Create two plugins, a linter and a syntax extension, and get them to communicate via a file
13/11/2017	Get access to the abstract syntax tree inside the linter plugin
27/11/2017	In the linter plugin, analyse each statement to see what variables they depends on and what variables are modified (if any)
11/12/2017	Produce a dependency tree for the entire program based on this analysis
25/12/2017	Look for areas in the tree which do not depend on one another, these areas could be run in parallel
08/01/2018	Save these areas to the shared file used for communication. Make sure that the syntax extension plugin can read this file correctly
22/01/2018	In the syntax extension plugin, first try one statement in parallel to test it works and then try to run everything in parallel
29/01/2018	Run tests against rust programs for program correctness and calculate speedup/slowdown against the sequential version
12/02/2018	In the linter plugin, try to analyse the speed of each parallelisable part. Threads have an overhead so it may be faster to run in sequential for some cases. Record this into the shared file
26/03/2018	In the syntax extension plugin, take into consideration the time information and only run parts in parallel if it would be faster to run in parallel
05/03/2018	Run more tests against rust programs for speedup/slowdown

2.3 Possible Extensions

If I'm ahead of schedule with the coding section of this project, I could look into parallelising branches which have a slow condition (i.e. 'if' statements). Each branch would run isolated in separate thread using cloned data. When the condition is finally calculated the incorrect branches would be terminated. This kind of parallelisation is different from the project plan as some threads are "thrown away".

Another extension upon the previous extension would be to utilise the GPU in cases where the same isolatable task is repeated a large number of times but on different data such that the overhead of moving this task to the GPU is worthwhile. It is very unlikely that I would attempt this extension due to the amount of work required, and I'm unsure about how much real world code exists that could be automatically run on the GPU efficiently.

3 Evaluation

To evaluate whether the chosen approach is the best option to solve the problem, it is worth looking at the downsides to this approach, and look other possible approaches. I will also describe how I will evaluate the final solution to see whether it solves the original problem.

3.1 Disadvantages to the Chosen Approach

The first downside is that the target program would need to be compiled twice due to the limitation of rust compiler plugins as explained section 2. It might be possible to edit the rust compiler to make it so that the program needs to be compiled once, but this has its own downsides as explained in 3.2.1.

The second downside is installing the compiler plugins. To take an existing sequential rust program and compile it so that it runs in parallel would require a few steps. The compiler plugin crate would need to be imported, and every function would need to be annotated so that the syntax extension plugin has edit access to the abstract syntax tree for each function.

3.2 Other Approaches

Below are a few other approaches to the problem that I considered before reaching the chosen approach. Each alternative approach looks at the potential benefits over the chosen one and then explains why I did not choose that alternate approach.

3.2.1 Make a custom rust compiler

One of the downsides of the chosen approach is that the program has to be compiled twice. I may be able to bypass this limitation by making a fork of the rust compiler instead of using compiler plugins. It would then be possible to examine the entire abstract syntax tree and then edit it afterwards. However, this approach seems infeasible for me as the rust compiler code complexity would add too much time to the project.

3.2.2 Use a different language from rust

There are many programs out there, written in many different languages. So why rust? Rust is not in the top 25 of most popular programming languages, according to StackOverflow (2017), so the number of existing programs may smaller than a more popular language. Rust is not unpopular however making it the most loved language and the 10th language on the most want to lean list (StackOverflow 2017). If there are not enough existing programs to justify specialising the tool just for rust (which there probably is), the number programs should increase exponentially in the future. The main reason I chose rust is because it is a safe language with guaranteed memory safety and threads without race conditions which would help significantly with the task.

3.2.3 Manual annotation of the parts of the program that are parallelisable

Instead of analysing the program for parallelisable parts, let the programmer use annotations to show which functions can be run in parallel, similar to OpenMP developed for C++ (Dagum and Menon 1998). This would be significantly simpler than the chosen approach as the analysation and dependency tree steps of the compilation could essentially be removed. The programmer is responsible for telling the compiler what can and cannot be parallelised. If the programmer accidentally makes a function parallel that should not be run in parallel, then the compiler would either need to spot this error (using the analysis step that was removed) or it would create an incorrect program. This approach would also not allow for automatic translation from sequential to parallel code and so it is not that much better than using threads manually.

3.2.4 Just run the sequential code

All of this analysis at compile time (and the fact that it would compile twice in the chosen approach) would increase compile times and running code in multiple threads has some runtime overhead. Is it worth the development time of automating the conversion of sequential code into parallelisable code? This is a valid argument if the number of sequential programs that could benefit from this software is small enough such that it would be faster and easier to redevelop these programs. It must also be true that developing future programs as parallel programs would have the same difficulty as developing it as a sequential program for this argument to hold. Since developing parallel programs is more difficult than developing sequential programs, and there are probably enough programs that could benefit from this software, it is worth the development time to create this piece of software.

3.3 Measuring Project Success

I can look at existing programs written in rust to see if there is any real world impact by measuring the speedup of the parallelised program vs the original sequential code. I can also calculate what percentage of the static sequential code has been parallelised, and what percentage of the runtime is using the parallel code. From these statistics, I can evaluate how well my solution solves the problem.

4 References

Baskaran, Muthu, Jj Ramanujam and P Sadayappan (2010). "Automatic C-to-CUDA code generation for affine programs". In: *Compiler Construction*. Springer, pp. 244–263.

Dagum, Leonardo and Ramesh Menon (1998). "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1, pp. 46–55.

D'Hollander, Erik H, Fubo Zhang and Qi Wang (1998). "The fortran parallel transformer and its programming environment". In: *Information sciences* 106.3-4, pp. 293–317.

Geer, David (2005). "Chip makers turn to multicore processors". In: *Computer* 38.5, pp. 11–13.

Kish, Laszlo B (2002). "End of Moore's law: thermal (noise) death of integration in micro and nano electronics". In: *Physics Letters A* 305.3, pp. 144–149.

StackOverflow (2017). *Stack Overflow Developer Survey 2017*. URL: `https://insights.stackoverflow.com/survey/2017` (visited on 09/10/2017).

The Rust Programming Language (2017). URL: `https://www.rust-lang.org/en-US/` (visited on 09/10/2017).

Automatic Parallelisation of Rust Programs at Compile Time

Michael Oultram
Student ID: 1428105

Dr Ian Batten
Project Supervisor

***Abstract*—Processors have been gaining more multi-core performance which sequential code cannot take advantage of. Many solutions exist to this problem including automatic parallelisation at the binary level, automatic parallelisation at the compiler and manually parallelising using annotations. The potential benefits of solving this problem are increased performance for existing programs, as well as making development of new software easier (as programmers do not need to worry about writing parallelised code).**

This paper focuses on automatically converting sequential source code into a parallelised program. The literature is explored and presenting into two main areas: theoretical models of automatic parallelisation and existing real-world parallelising compilers. The author presents a design for a new parallelising compiler for the Rust programming language. To demonstrate the design, three sequential elements of Rust programs are manually converted by this paper: function calls, for-loops and branches.

I. Introduction

Kish (2002) estimated the end of Moore's Law of miniaturization within 6-8 years or earlier (based on their publication date) and as such, manufacturers have been increasing processors' core count to increase processor performance (Geer 2005). Writing parallelised programs to take advantage of these additional cores has some difficulty and often requires significant changes to the source code. One solution to this problem, which is the focus of this paper, is to automatically transform sequential source code into parallelised code. This solution, if achieved, would allow for existing uncompiled sequential programs to take advantage of the new hardware. Developing new programs would be easier for programmers as they can just write the code sequentially and let the compiler parallelise it.

Section II examines the literature to finds that most existing solutions focus on unsafe languages such as C++ and FORTRAN. Since a significant proportion of existing programs are written in these languages, it makes sense for other authors to focus on these languages. There is a downsides to focusing on these unsafe languages, dependency analysis becomes much more difficult. To avoid this downside, the design this paper introduces in section IV uses the safe programming language Rust.

"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety" (*The Rust Programming Language* 2017). This brief introduction to some of Rust's features will explain the elements of the language necessary for the reader to understand for later sections of this paper. For further understanding of the language, it is recommended that the reader looks at *The Rust Book* (2017).

In C++ (or other unsafe languages), a variable can be accessed and modified if it is in scope. In Rust, each variable is immutable by default and if a variable is passed to another function, then that function takes 'ownership' of the variable. The code below would not compile in Rust, but similar code would compile in C++. When `f` is called, it takes ownership of the variable `i`. This essentially moves it out of scope, and so it cannot be moved into `g`.

```
fn main() {
    let mut i = vec![1,2,3];
    f(i);
    g(i);
}
```

To make the code above compile, we have two options. The first option is to let `f` 'borrow' the variable instead of moving the ownership of `i` to `f`. In the example below, variable `i` is 'mutably borrowed' by `f`, and then 'immutably borrowed' by `g`. The syntax of Rust makes it explicit that `f` may modify this variable and `g` cannot. This information expressed is available to the compiler.

```
fn main() {
    let mut i = vec![1,2,3];
    f(&mut i);
    g(&i);
}
```

The second option is to clone `i` instead. This would keep the types of `f` and `g` the same as the first example. In the example below, we make a copy of `i` before moving it to `f`. Any changes that `f` makes to `i` will not be transferred to `g`, unlike the previous example.

```
fn main() {
    let mut i = vec![1,2,3];
    let j = i.clone();
    f(i);
    g(j);
}
```

Note: `i` must be cloned before moving `i` to `f`, or else we would not have access to `i` to clone it.

II. Related Work

A. Parallelisation Models

In this section, we look at theoretical models of automatic parallelism. The static parallelism subsection shows related work where the schedule is fixed and calculated at 'compile' time. It is shown how rearranging loop iterations and optimising memory access patterns for multiple threads can increase performance. The speculative parallelism subsection shows related work where the schedule is more flexible. This kind of parallelism tries to run dependent tasks in parallel and detecting when there is a conflict. When a conflict occurs, some parallel thread is 'undone' and rerun.

1) Static parallelism: Feautrier (1992) describes one model of a parallel program as a set of operations Ω on an initial store, and a partial ordering binary relation Γ also known as a dependency tree. It is shown that this basic model of a parallel is equivalent to affine scheduling, where Ω and Γ are described as linear inequalities. Finding a solution where these linear inequalities hold produces a schedule for the program where dependent statements are executed in order. There are some programs where no affine schedule exists. Bondhugula et al. (2008) uses the affine scheduling model on perfectly, and imperfectly nested loops. They describe the transformations needed to minimise the communication between threads, further increasing the performance of the parallelised code.

An alternative method to affine scheduling is iteration space slicing introduced by Pugh and Rosser (1997). "Iteration space slicing takes dependency information as input to find all statement instances from a given loop nest which must be executed to produce the correct result". Pugh and Rosser (1997) shows how this information can be used to transform loops on example programs to produce a real world speedup. Beletska et al. (2011) shows that iteration space slicing extracts more coarse-grained parallelism than affine scheduling.

2) Speculative parallelism: Zhong et al. (2008) shows that there is some parallelisable parts hidden in loops that affine scheduling and iteration space splicing cannot find. They propose a method that runs future loop iterations in parallel with past loop iterations. If a future loop iteration accesses some shared memory space, and then a past iteration modifies that same location, the future loop iteration is 'undone' and restarted. It is shown that this method increases the amount of the program that is parallelised.

Prabhu, Ramalingam and Vaswani (2010) introduce two new language constructs for C# to allow the programmer to manually specify areas of the program that can be speculatively parallelised. Yiapanis, Brown and Luján (2016) designs a parallelising compiler which can automatically take advantage of speculative parallelism.

B. Parallelisation Implementations

In this section we look at: parallelising compilers which focus on parallelising FORTRAN programs; OpenMP which is an model for shared memory programming and parallelising compilers which convert sequential CPU code into parallelised GPU code. Some of these parallelising compilers are based off of models described in subsection II-A.

Eigenmann, Hoeflinger and D. Padua (1998) manually parallelises the PERFECT benchmarks for FORTRAN which are compared with the original versions to calculate the potential speedup of an automatic parallelising compiler. D'Hollander, Zhang and Wang (1998) developed a FORTRAN transformer which reconstructs code using `GOTO` statements so that more parallelisms can be detected. It performs dependency analysis and automatically parallelised loops by splitting the task into jobs. These jobs can be split between networked machines to run more jobs concurrently. Rauchwerger and D. A. Padua (1999) introduce a new language construct for FORTRAN programs which allows for run-time speculative parallelism on for loops. Their implementation parallelises some parts of the PERFECT benchmarks which existing parallelising compilers of the time could not find.

Quiñones et al. (2005) introduce the Mitosis compiler which combines speculation with iteration space slicing. There is always only one non-speculative thread which is seen as the base execution; all other threads are speculative. The Mitosis compiler computes the probability of two iterations conflicting. If this probability is low, and there is a spare thread unit, then the loop iteration is executed in parallel. The non-speculative thread detects any conflicts as it is the only thread that can commit results.

Dagum and Menon (1998) introduces a programming interface for shared memory multiprocessors called OpenMP targeted at FORTRAN, C and C++. The programmer annotates the elements of the program that are parallelisable, which the compiler recognises and performs the optimisation. OpenMP is compared to alterative parallel programming models. Kim et al. (2000) introduces the ICP-PFC compiler for FORTRAN which uses the OpenMP model. All loops in the source code are analysed by calculated a dependency matrix. The compiler automatically adds the relevant OpenMP annotations to the loop. Lam (2011) extends OpenMP using machine learning to automate the parallelisation. The system is trained using a set containing programs already parallelised using OpenMP. The knowledge learned is applied to sequential programs to produce parallelised programs.

A CPUs architecture is typically optimised for latency whereas a GPUs architecture is typically optimised for throughput. This can make GPUs perform much better that CPUs for a certain type of task. Baskaran, Ramanujam and Sadayappan (2010) uses the affine transformation model to convert sequential C code into parallelised CUDA code. `For` loops are tiled for efficient execution on the GPU.

III. Problem Details

The literature focuses on unsafe languages such as C/C++ and FORTRAN. As a result, most of their methods revolve around understanding conflicts between statements/loop iterations of complex code. Since memory in Rust has 'ownership', this dependency information is more readily available.

The Rust compiler allows for plugins of different types but there are two types that are of interest to this problem. Written in the order of execution, they are:

- Syntax Extension: can modify the abstract syntax tree of any annotated function.
- Early Lint Pass: can see abstract syntax tree of each uncompiled function, with macros expanded, without annotations, but it cannot edit them.

Before we can automate the parallelisation of Rust programs, we must look at some example programs to fully understand the problem. The examples were written sequentially and then manually parallelised. The pattern of parallelisation could be applied to different examples, and a design is proposed to automate this in section IV. In these examples, threads are used as if they have no overhead, to simplify the explanation of how a sequential program could be parallelised. In reality this is not the case and so in section IV `thread::spawn` is replaced with something more optimal.

A. Parallel Function Optimisations

In some cases, the function arguments are known before the result of the function is required. When the program is sequential, the program must wait for the result of that function call. If the program was parallelised, this function could be started in the background as soon as the arguments are decided. This would allow the result to be ready for when it is requested or at least closer to ready than the pure sequential version.

Example: Algorithm 1 is a program that calculates Fibonacci numbers in a very inefficient way. The main method calculates `i` on the second line, and since it is not mutable it cannot be changed. The `slow_method` is executed before the `fibonacci` function, even though the two functions are independent. Just running the `fibonacci` function earlier would not improve performance as the `slow_method` would now have to wait for `fibonacci` function to finish. Ideally, `fibonacci` should start as soon as `i` is calculated, and at the same time `slow_method` should be executed.

Algorithm 2 shows one way of converting Algorithm 1 into a more parallelised version. These changes allow for `fibonacci_parallel` to be called as soon as `i` is decided. `fibonacci_parallel` does not block, and will return almost immediately with a `JoinHandle`. The inner thread is executed in the background, allowing `slow_method` to be executed at the same time. Inside the background thread `fibonacci_parallel` is called again which spawns another thread. These changes also allow for `n1` and `n2` to be executed in parallel. Calling `join()` on a `JoinHandle` will block until the thread is terminated, and returns the result of the thread. The `fibonacci` function is changed so that any external code that uses this function will still work (although sequentially).

Algorithm 1 Sequential Fibonacci Function

```
fn main() {
    // Set i = first program argument
    let program_args = std::env::args();
    let i = program_args[0].trim().parse::<u32>();
    slow_method();
    println!("{}", fibonacci(i));
}

fn fibonacci(n: u32) -> u64 {
    match n {
        0    => 0,
        1, 2 => 1,
        _    => fibonacci(n-1) + fibonacci(n-2),
    }
}
```

Algorithm 2 Parallel Fibonacci Function

```
fn main() {
    // Set i = first program argument
    let program_args = std::env::args();
    let i = program_args[0].trim().parse::<u32>();
    let fib = fibonacci_parallel(i);
    slow_method();
    println!("{}", fib.join());
}

fn fibonacci_parallel(n: u32) -> JoinHandle<u64> {
    thread::spawn(move || {
        match n {
            0    => 0,
            1, 2 => 1,
            _    => {
                let n1 = fibonacci_parallel(n-1);
                let n2 = fibonacci_parallel(n-2);
                n1.join() + n2.join();
            },
        }
    })
}

fn fibonacci(n: u32) -> u64 {
    let fib = fibonacci_parallel(n);
    fib.join()
}
```

B. For-Loop Optimisations

Loop iterations that are independent of each other could be run at the same time to increase performance. It is not always clear how many iteration of a loop may occur, especially while-loops. For-loops are more explicit about how many iterations will occur, but they can still terminate early with a `break` or a `return` statement.

Example: Algorithm 3 is a real world example with a two level for-loop combined with an if-statement which returns the password for the first valid hash. The inner for-loop is

not parallelisable as each iteration reads and modifies the `hash_word` variable. The outer for-loop is parallelisable, but if we were to naively put the contents into separate threads, as shown in Algorithm 4, then we may end up with the wrong result. In the sequential version, the first password in the list to match the hash would be returned but in the naive parallel version, the password returned depends on the order the threads are run. In this example program at most only one iteration could possible enter the if-branch, terminating the for-loop early, but in other examples there could be more than one. Algorithm 5 is the final complete parallelisation of Algorithm 3. Each word is hashed in it's own thread, compared to `hash_password` and returned to be stored in the initial thread. The function does not return a result until all previous iteration threads have also returned.

Algorithm 3 Sequential Password Cracker

```
fn crack_password(dictionary: &Vec<String>,
                  hash_password: String)
                  -> Option<String> {
    for word in dictionary {
        // Hash word using Sha256
        let mut hash_word = word;
        for _ in 0..40 {
            let mut sha = Sha256::new();
            sha.input_str(word);
            hash_word = sha.result_str();
        }
        // Check if hash matches
        if hash_password == hash_word {
            return Some(word.clone());
        }
    }
    // No hash matched
    None
}
```

C. Branch Optimisations

In the previous optimisations, all the code that is run in parallel would have also been run in sequential normally. This optimisation more speculative than the other optimisations described as some threads are going to be disregarded. Each side of the branch for an if-statement could be run in parallel whilst waiting for the condition. When the condition is finally calculated, the result of the correct branch is kept. Algorithm 7 shows this optimisation applied to Algorithm 6. Ideally, when the condition is calculated and the incorrect branch is still being calculated in a thread, then this thread should be cancelled. This is not easily achievable in Rust so this optimisation is left as a possible extension of the design.

IV. Design Overview

A typical parallelising compiler, such as those described in section II, have a few stages. First the compiler looks at each statement of the source code, and performs dependency analysis to calculate the critical path. Any independent statements can be run in parallel. A scheduling algorithm calculates which order the statements should be executed, grouping statements that are dependent on each other into

Algorithm 4 Naive Parallel Password Cracker

```
fn crack_password(dictionary: &Vec<String>,
                  hash_password: String)
                  -> Option<String> {
    // Create a communication channel
    let (tx, rx) = mpsc::channel();
    // Start a thread for each dictionary entry
    for i in 0..dictionary.len() {
        let word = dictionary[i].clone();
        let tx = tx.clone();
        thread::spawn(move || {
            let result = {
                // Hash word using Sha256
                let mut hash_word = word;
                for _ in 0..40 {
                    let mut sha = Sha256::new();
                    sha.input_str(word);
                    hash_word = sha.result_str();
                }
                // Send result via channel
                if hash_password == hash_word {
                    Some(word)
                } else {
                    None
                }
            };
            tx.send(result);
        });
    }
    // Receive up to dictionary.len() results
    for _ in 0..dictionary.len() {
        if let Some(result) = rx.receive() {
            return result;
        }
    }
    // No hash matched
    None
}
```

the same task. A compile time performance metric is used on each task to estimate the potential speedup of the parallel version over the sequential version. Due to the overhead of threads, this potential speedup would not actually be achieved. The parallelising compiler takes this into account and will only use the parallel version if it predicts it will really be faster.

This papers design takes elements from a typical parallelising compiler, dividing the task into two main stages, an analysis stage and a modification stage. The analysis stage is run by a linter plugin and the modification stage is run by a syntax extension plugin. Analysis stage must come before the modification stage, so compiling must be done twice (once for each stage). When the plugin is loaded, it determines which stage it is by looking for a shared JSON file. If this file does not exist, then it is the analysis stage, otherwise the file's content is loaded and the stage is updated to be the modification stage.

A. Analysis stage

On the first compilation, the syntax extension plugin would do nothing. The linter plugin would view the entire abstract data tree and analyse what each statement depends on and modifies. This would create a dependency tree, where

Algorithm 5 Parallel Password Cracker

```
fn crack_password(dictionary: &Vec<String>,
                  hash_password: String)
                  -> Option<String> {
    let (tx, rx) = mpsc::channel();
    for i in 0..dictionary.len() {
        let word = dictionary[i].clone();
        let tx = tx.clone();
        thread::spawn(move || {
            let result = {
                // Hash word using Sha256
                let mut hash_word = word;
                for _ in 0..40 {
                    let mut sha = Sha256::new();
                    sha.input_str(word);
                    hash_word = sha.result_str();
                }
                // Check if hash matches
                if hash_password == hash_word {
                    Some(word)
                } else {
                    None
                }
            };
            tx.send((i, result));
        });
    }
    // Receive all the results
    let mut results = vec![None; list.len()];
    let mut verified_upto = -1;
    for _ in 0..results.len() {
        // Receive result and store in location
        let (i, result) = rx.receive();
        results[i] = Some(result);
        // Check for final result
        for i in 0..results.len() {
            if let Some(result) = results[i] {
                if let Some(word) = result {
                    return word;
                }
            } else {
                // Have not received i result yet
                break;
            }
        }
    }
    // No hash matched
    None
}
```

Algorithm 6 Sequential Slow If

```
fn f(a: u32, b: u32) -> u32 {
    if slow_condition(a, b) {
        (a * (b - a)) + 5
    } else {
        ((a * b) + 19) ^ 2
    }
}
```

any two statements that are independent can be run in parallel. The linter plugin would use this dependency tree to determine which parts should be parallelised, and save this information to a file. Some parts cannot be parallelised.

1) Side Effects: A pure function will return the same output if it is given the same input. If the function can return something different when given the same input, then it is not pure. The element of the function that changes the output is known as a side effect. The main side effect found in

Algorithm 7 Parallel Slow If

```
fn f(a: u32, b: u32) -> u32 {
    let true_branch = {
        let a = a.clone();
        let b = b.clone();
        thread::spawn(move || (a * (b - a)) + 5)
    };
    let false_branch = {
        let a = a.clone();
        let b = b.clone();
        thread::spawn(move || ((a * b) + 19) ^ 2)
    };
    if slow_condition(a, b) {
        true_branch.join()
    } else {
        false_branch.join()
    }
}
```

programs is IO. For example, a function takes a filename as input and outputs the contents of that file. If the contents of that file changes, the function would outputs something different even though its parameters are the same. Printing to standard out is also considered a side effect. Side effects generally cannot be parallelised, as the order of execution must be kept the same (we don't want to print in the wrong order). Additionally, everything that occurs before the side effect in the sequential version should also be run before the side effect in the parallel version.

B. Modification stage

On the second compilation, the syntax extension reads the file the linter plugin created on the previous compilation. This lists all the changes required and the syntax plugin can apply those changes to the abstract syntax tree function by function. The modifications applied would be similar to the sample programs shown in section III. Those modifications assumed that threads have no overhead and were used as a method of describing how a sequential program could be run in parallel. In the real world, these parallelised sample programs produce an unreasonable amount of threads and most of the threads are just waiting for other threads to return. This synchronisation can cause the parallel version to perform a lot worse than sequential code. The following subsections describe some of the problems with the parallelised sample programs, and provides some solutions.

1) Infinite Thread Problem: The initial solution for too many threads is to use a thread-pool so only a fixed number of threads are executed at the same time. This solution would not work in this case as if all the threads are waiting on a future task, which is also waiting for a thread to be free then we get stuck in a deadlock. By tweaking the design of the thread-pool slightly, we can have a fixed the number of tasks and prevent this deadlock.

As the task queue is the main cause of this deadlock, it is removed from this new proposed design which will be referred to as a no-queue thread-pool. Whenever a task is created, it looks for an available thread and that thread starts

executing the task in the background. If there is no thread available, then the current thread should execute the task. Also, if the the current thread is waiting for a task to return, then it could execute another task whilst waiting.

To modify the sample programs to use the no-queue thread-pool would be of minimal work; it would require a shared memory space (to gain access to the threads) and instead of calling `thread::spawn`, it would call another function.

2) Performance Analysis: Parallelising small tasks, or tasks that would require a lot of synchronisation when run in parallel actually run significantly slower than the sequential code. In these cases, the code should not be parallelised. There is a problem though, how does the compiler know if it is faster parallelised or sequential.

One possible solution is to run the sequential version in one thread, and the parallel version in the other threads. Whichever version finishes first is used, and the other version is cancelled. This approach leaves the performance approach until run-time, but it is a very inefficient solution as the parallel version now has fewer threads.

Another solution is to use a performance metric at compile time to estimate which version will be faster. If we assume each statement takes one unit of execution time, we can attempt to calculate how many time units the sequential and parallel version will take. Since we know that there is an overhead to the parallel version, we should add on some extra time units. The faster version would be the version that is compiled. This solution would not really work as described as the execution time may be dependent on how many iterations of a loop (which isn't necessarily known at compile time). It is also dependent on the number of cores the machine has, also not known at compile time.

The best solution is a combination of the previous two ideas. The compiler should create an function which estimates both the sequential and the parallel version execution time. This function can take in any run-time conditions i.e. the number of iterations of a loop or the number of cores the machine has. This function should be fast such that the overhead of calculating which version is faster should be negligible. At run-time this function predicts the faster version using the run-time variables, and runs only that version.

V. References

Baskaran, Muthu, Jj Ramanujam and P Sadayappan (2010). "Automatic C-to-CUDA code generation for affine programs". In: *Compiler Construction*, pp. 244–263.

Beletska, Anna et al. (2011). "Coarse-grained loop parallelization: Iteration space slicing vs affine transformations". In: *Parallel Computing* 37.8, pp. 479–497.

Bondhugula, Uday et al. (2008). "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model". In: *International Conference on Compiler Construction*, pp. 132–146.

D'Hollander, Erik H, Fubo Zhang and Qi Wang (1998). "The fortran parallel transformer and its programming environment". In: *Information sciences* 106.3-4, pp. 293–317.

Dagum, Leonardo and Ramesh Menon (1998). "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1, pp. 46–55.

Eigenmann, Rudolf, Jay Hoeflinger and David Padua (1998). "On the automatic parallelization of the Perfect Benchmarks (R)". In: *IEEE Transactions on Parallel and Distributed Systems* 9.1, pp. 5–23.

Feautrier, Paul (1992). "Some efficient solutions to the affine scheduling problem. I. One-dimensional time". In: *International journal of parallel programming* 21.5, pp. 313–347.

Geer, David (2005). "Chip makers turn to multicore processors". In: *Computer* 38.5, pp. 11–13.

Kim, Hong Soog et al. (2000). "ICU-PFC: An automatic parallelizing compiler". In: *Proceedings - 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, HPC-Asia 2000*, pp. 243–246.

Kish, Laszlo B (2002). "End of Moore's law: thermal (noise) death of integration in micro and nano electronics". In: *Physics Letters A* 305.3, pp. 144–149.

Lam, Nam Quang (2011). "A Machine Learning and Compiler-based Approach to Automatically Parallelize Serial Programs Using OpenMP". In: *Master's Projects* 210.

Prabhu, Prakash, Ganesan Ramalingam and Kapil Vaswani (2010). "Safe programmable speculative parallelism". In: *ACM Sigplan Notices* 45.6, pp. 50–61.

Pugh, William and Evan Rosser (1997). "Iteration space slicing and its application to communication optimization". In: *Proceedings of the 11th international conference on Supercomputing*. ACM, pp. 221–228.

Quiñones, Carlos García et al. (2005). "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices". In: *ACM Sigplan Notices* 40.6, pp. 269–279.

Rauchwerger, Lawrence and David A. Padua (1999). "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization". In: *IEEE Transactions on Parallel and Distributed Systems* 10.2, pp. 160–180.

The Rust Book (2017). URL: https://doc.rust-lang.org/book/ (visited on 12/11/2017).

The Rust Programming Language (2017). URL: https://www.rust-lang.org (visited on 12/11/2017).

Yiapanis, Paraskevas, Gavin Brown and Mikel Luján (2016). "Compiler-Driven Software Speculation for Thread-Level Parallelism". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38.2, pp. 1–45.

Zhong, Hongtao et al. (2008). "Uncovering hidden loop level parallelism in sequential applications". In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 290–301.