Verification of Loop Parallelisations *

S.C.C. Blom, S. Darabi, and M. Huisman

University of Twente, the Netherlands

Keywords: Formal verification, Hoare Logic, Permission-based separation logic, Parallel loops, Parallelising compilers.

Writing correct parallel programs becomes more and more difficult as the complexity and heterogeneity of processors increase. To address this issue, parallelising compilers aim to detect loops that can be executed in parallel. However, this detection is not perfect. Therefore developers can typically also add compiler directives to declare that a loop is parallel. Any loop annotated with such a compiler directive will be assumed to be parallel by the compiler.

This work addresses the correctness of such compiler directives for loop parallelisation. This is achieved by adding specifications to the program that when verified guarantee that the program can be parallelised without changing its behaviour. Our specifications stem from permission-based separation logic [3, 4], an extension of Hoare logic. This has the advantage that we can easily combine the specifications related to loop parallelisation with functional correctness properties.

Concretely, for each loop body we add an *iteration contract*, which specifies the iteration's resources, i.e., the variables read and written by one iteration of the loop. We prove that if the iteration contract can be proven correct without any further annotations, the iterations are independent and the loop is parallelisable. If a loop has dependences, we can add additional annotations that capture these dependences. These annotations specify how resources are transferred to another iteration of the loop. We then identify a class of annotation patterns for which we can prove that the loop can be vectorised because they capture forward dependences.

Listing 1 shows an example of an *independent loop* with its iteration contract. This contract requires that at the start of iteration i, permission to write a[i] is available, as well as permissions to read b[i]. Further, the contract ensures that these permissions are returned at the end of iteration i. The iteration contract implicitly requires that the *separating conjunction of all iteration preconditions* holds before the first iteration of the loop, and that the *separating conjunction of all iteration postconditions* holds after the last iteration of the loop. For example, the contract in Listing 1 implicitly specifies that upon entering the loop, permission to write the first N elements of a must be available, as well as permission to read the first N elements of b.

We formally prove the correctness of our approach by showing that a correct iteration contract capturing a loop independence or a forward dependence indeed implies that a loop can be parallelised or vectorised, while preserving the behaviour of the sequential loop. To construct the proof, we first define the semantics of the three loop execution paradigms: sequential, vectorised, and parallel. We also define the instrumented semantics for a loop specified with an iteration contract. Next, to prove the soundness of our approach we show that the instrumented semantics of an independent loop is equivalent to the parallel execution of the loop, while the instrumented semantics of a loop with a forward dependence is an extension of the vectorised execution of the loop. Functional equivalence of two semantics is shown by transforming the computations in one semantics into the computations in the other semantics by swapping adjacent independent execution steps.

Moreover, we provide automated tool support for our technique as provided by the VerCors tool

*The full version of this work has been submitted to the FASE 2015 conference.

```
for(int i=0; i < N; i++) /*@
      requires Perm(a[i],1) ** Perm(b[i],1/2);
      ensures  Perm(a[i],1) ** Perm(b[i],1/2);
 @*/ { a[i]= 2 * b[i];       }
```

Listing 1: Iteration contract for an independent loop

set. The VerCors tool set was originally developed to reason about multi-threaded Java programs, but it has been extended to support verification of OpenCL kernels [2] and parallel loops.

Finally, we also support translation of a verified iteration contract (including possibly functional property specifications) into a verifiable contract for the parallelised or vector program, written as a kernel. As an extra benefit, the iteration contract can help parallelsing compilers to reveal where synchronisation is needed in the parallelised program.

Our approach is motivated by our work on the CARP project[1]. As part of this project the PENCIL language has been developed [1]. It is a high-level programming language designed to ease the programming of many-core processors such as GPUs. Its core is a subset of sequential C, imposing strong limitations on pointer-arithmetic. However, it should be noted that our approach also is applicable to other programming languages or libraries that have a similar parallel loop construct, such as OpenMP [5], parallel_for in C++ TBB [7] and Parallel.For in .NET TPL [6].

The main contributions of our work are the following:

- a specification technique, using iteration contracts and dedicated transfer annotations that can capture loop dependences;
- a soundness proof that loops respecting specific patterns of iteration contracts can be either parallelised or vectorised; and
- compilation of iteration contracts to kernel contracts for the parallelised or vectorised program.

References

[1] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. *CoRR*, abs/1302.5586, 2013.

[2] S. Blom, M. Huisman, and M. Mihelčić. Specification and verification of GPGPU programs. *Science of Computer Programming*, 2014.

[3] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[4] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[5] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[6] Microsoft TPL. http://msdn.microsoft.com/enus/library/dd460717.aspx.

[7] Threading Building Blocks. `http://threadingbuildingblocks.org`.

[1] See `http://www.carpproject.eu/`.