RVSDG: An Intermediate Representation for the Multi-Core Era

Nico Reissmann, Jan Christian Meyer, and Magnus Själander
Norwegian University of Science and Technology
Trondheim, Norway
firstname.middlename.lastname@ntnu.no

ABSTRACT

The embrace of chip multiprocessors by the microprocessor industry moved thread-level parallelism (TLP) into the limelight. As the manual creation of multi-threaded code is difficult, error prone, and time consuming, it is desirable to find automatic ways to extract TLP from programs. Current state-of-the-art compilers, however, are ill-equipped for this task, as they employ inherently sequential intermediate representations (IRs).

This paper advocates a paradigm shift in compiler IRs, and presents the Regionalized Value State Dependence Graph (RVSDG) as a concurrent alternative. The RVSDG represents programs in demand-dependence form, implicitly supports structured control flow, and explicitly models the flow of data. It is capable of representing an entire program within a single IR, exposes structures and enforces invariants that are beneficial for optimizations, and simplifies the extraction of concurrent computations in programs. This makes RVSDG an appealing IR for optimizing and parallelizing compilers.

1 INTRODUCTION

Static exploitation of parallelism has become the primary way to improve performance and power efficiency. Modern processors feature multiple cores and rely on multi-threaded code to fully utilize these cores and gain performance. The task of producing multi-threaded code could be left to developers, but this approach has several drawbacks. First, writing parallel code is demonstrably more difficult than writing sequential code. It requires developers to reason about new sources of bugs, such as data races, deadlocks, and livelocks. Second, it can lead to new performance bottlenecks, such as lock contention and synchronization overheads. These performance issues are often difficult to identify and might (re-)appear for different architectures. Third, there exists many legacy applications that were written for single-core processors. It would require an enormous programming effort to parallelize these programs.

The alternative to manual parallelization is to let the compiler handle the task. This relieves the programmer from the burden of code parallelization and permits the compiler to target the parallelization to the underlying architecture. Automatic parallelization has been extensively studied in the domain of scientific and numeric computing, where programs contain a lot of data-level parallelism in the form of DOALL loops. Progress has also been made for inherently sequential programs, where the Helix project [1] and methods such as Decoupled Software Pipelining [10] managed to extract thread-level parallelism (TLP) from individual loops and achieved considerable speedups.

MCC'18, November 2018, Gothenburg, Sweden

Even though these techniques only transform local loop structures, they crucially rely on global program information to resolve dependencies between loop iterations and of operations within a loop. The more dependencies the compiler manages to resolve, the easier it is to extract profitable TLP. The task of resolving dependencies, however, is currently overly complicated by the used compiler IRs. The employed representations, such as the control flow graph (CFG), fail to expose the necessary program structures, such as loops, provide no global program view, and are unable to encode the (in-)dependence between operations and/or program structures. For example, even if the compiler manages to detect that two loops are independent, it has no way to encode this information in the IR. The CFG is an inherently sequential IR and a remnant of the sequential programming era, where the majority of the performance improvements were achieved transparent to the software. It is unsuitable in a time where the exploitation of parallelism is paramount to performance improvements.

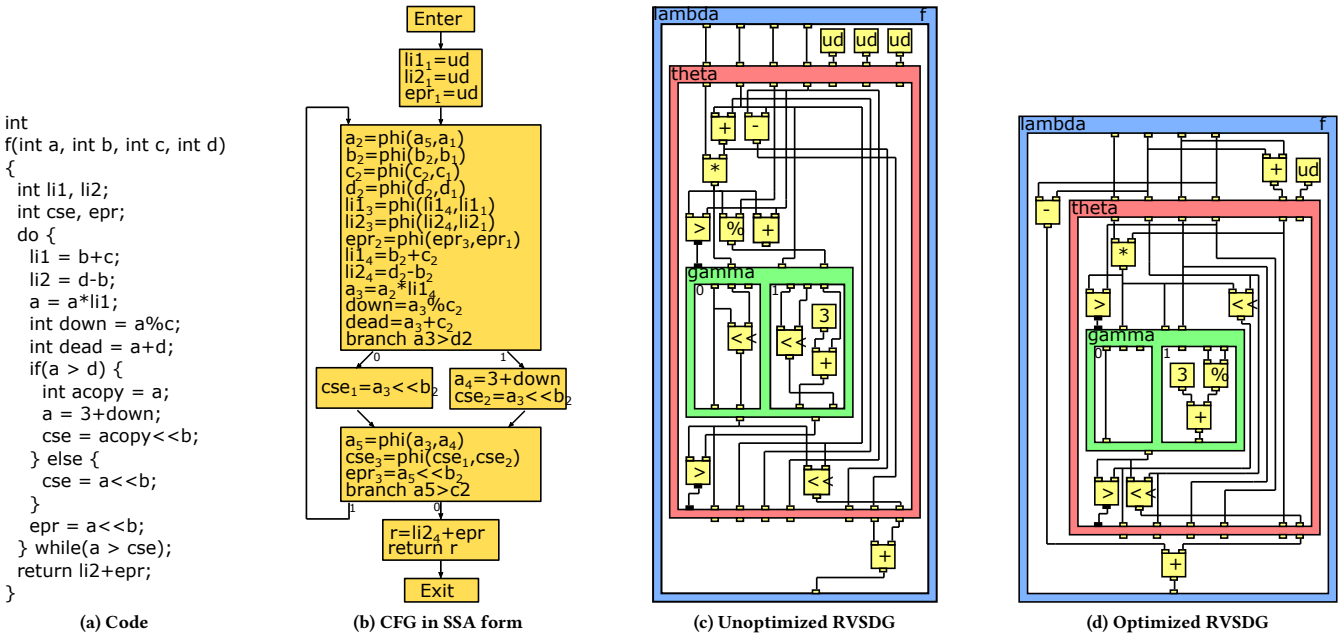
This paper advocates for a fundamental change in compiler IRs. A good IR provides abstractions to facilitate the implementation of analyses, optimizations, and program transformations. It should highlight and expose program properties that are important to transformations, reducing their complexity and simplifying their implementation. We present the Regionalized Value State Dependence Graph (RVSDG) as an alternative IR for optimizing and parallelizing compilers. The RVSDG represents programs in demand-dependence form, only encodes structured control flow, and explicitly models the data flow between operations. This raises the IR's abstraction level, permits simple and powerful implementations of optimizations, and helps to expose the parallelism inherent in programs.

The remainder of this paper introduces the RVSDG through examples and illustrates its benefits in Section 2. We provide some preliminary results from jlm[1], a compiler that uses the RVSDG for optimizations, in Section 3, and conclude in Section 4.

2 MOTIVATION

This section motivates the use of the RVSDG as an IR for optimizing and parallelizing compilers. It illustrates its potential to simplify conventional compilation and to expose concurrent computations. We show that the RVSDG raises the IR abstraction level by enforcing desirable properties, such as SSA form, explicitly encodes important structures, such as loops, and relaxes the overly strict order of the input program. This leads to a more normalized program representation and avoids many idiosyncrasies and artifacts from other IRs and helps to expose parallelism in programs.

[1] https://github.com/phate/jlm



(a) Code (b) CFG in SSA form (c) Unoptimized RVSDG (d) Optimized RVSDG

Figure 1: Simplifying conventional compilation.

Table 1: Ten most invoked LLVM passes at 03.

Optimization	# Invocations
1. Alias Analysis (`-aa`)	16
2. Dominator Tree Construction (`-domtree`)	14
3. Basic Alias Analysis (`-basicaa`)	13
4. Scalar Evolution Analysis (`-scalar-evolution`)	10
5. Natural Loop Canonicalization (`-loop-simplify`)	9
6. Redundant Instruction Combinator (`-instcombine`)	8
7. Loop-Closed SSA Form (`-lcssa`)	8
8. Loop-Closed SSA Form Verifier (`-lcssa-verification`)	8
9. CFG Simplifier (`-simplifycfg`)	7
10. Natural Loop Information (`-loops`)	6
Total	99
SSA Restoration	12

2.1 Simplifying Conventional Compilation

Figure 1a shows a function with a simple loop and a conditional, and Figure 1b shows the corresponding control flow graph (CFG) in static single assignment (SSA) form [4]. The CFG in SSA form is the predominant IR for optimizations in modern imperative language compilers [14]. Its nodes represent a list of totally ordered operations and its edges a program's possible control flow paths, permitting efficient control flow optimizations and simple code generation. The CFG's translation to SSA form improves the efficiency of many data flow optimizations [13, 15].

While the CFG is simple to construct and destruct, it provides few abstractions and invariants to facilitate the implementation of optimizations and analyses. CFG-based compilers must constantly (re-)discover and canonicalize loops, establish invariants, or restore SSA form. Table 1 shows the ten most invoked LLVM passes and highlights the helper passes in bold that only or partially perform such tasks. They amount to 52 pass invocations (excluding SSA restoration, as it is implemented ad-hoc), or 23% of all pass invocations. This lack of enforced invariants complicates the implementation of optimizations and analyses, increases engineering effort, unnecessarily prolongs compilation time, and leads to compiler bugs [7–9].

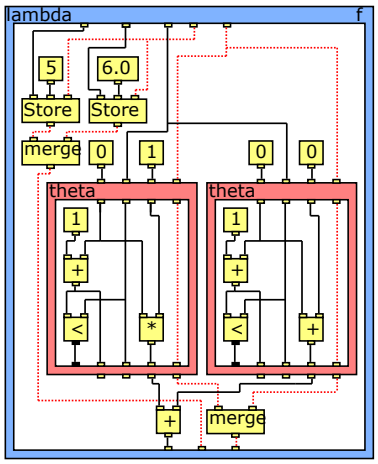
In contrast, the RVSDG exposes program structure and enforces invariants beneficial for optimizations and analyses. Figure 1c shows the RVSDG corresponding to Figure 1a. It is an acyclic demand-dependence graph where nodes represent simple operations or control flow constructs, and edges the dependencies between computations. In Figure 1c, simple operations are colored yellow, conditionals are green, loops are red, and functions are blue.

The RVSDG is a data centric IR focusing on explicit data flow instead of control flow. This leads to a more normalized program representation and simplifies the implementation of transformations [5, 6, 16]. For example, the RVSDG is always in strict SSA form as edges connect each operand input to only one output, eliminating the need for SSA restoration passes [2]. The representation of intra- and inter-procedural control flow as nodes permits to encode the entire program within a single representation, and avoids additional data structures, such as the call graphs, or passes, such as loop detection and normalization.

These properties combined with its explicit data flow enable simple and powerful optimizations. Figure 1d shows the optimized RVSDG of Figure 1c, illustrating some of these optimizations. For example, the dead addition can be removed from the loop as it has no users. The addition and subtraction computing `li1` and `li2` are moved out of the loop as their operands, *i.e.* `b`, `c`, and `d`,

```
int
f(int* x, float* y,
  int k)
{
  *x = 5;
  *y = 6.0;
  int i=0, f=1;
  int sum=0, fac=1;
  do {
   sum += i;
   i++;
  } while(i < k);
  do {
    fac *= f;
    f++;
  } while(f < k);
  return fac+sum;
}
```

(a) Code



(b) RVSDG

Figure 2: Exposing concurrent computations.

are loop invariant (all three of them connect the entry of the loop to the exit). The shift operations from the conditional are hoisted out and combined, while the division operation is moved into the conditional as it is only used in one alternative. In contrast to CFG-based compilers, all these optimizations are performed directly on the unoptimized RVSDG of Figure 1c. No additional data structures or helper passes are required.

2.2 Exposing Concurrent Computations

The RVSDG's explicit representation of program states enables it to encode the relations of side-effecting operations in the graph, as illustrated in Figure 2. The depicted function contains two non-aliasing store operations and two independent loops. In a CFG, the stores and loops are strictly ordered, and optimizations require an additional data structure for alias information, as well as passes to determine the independence of these loops. The CFG is incapable to encode such information directly in the graph.

In contrast, the RVSDG permits to represent this information in the graph, as shown in Figure 2b. The function has two additional input states (red dotted lines) that are used for sequentializing memory operations and (potentially non-terminating) loops, respectively. The first state is used to express that both stores are non-aliasing, while the second state preserves (potentially) non-terminating loops. This exposes the parallelism of these loops, as they are represented explicitly as nodes and share no dependencies.

Generally, the extraction of concurrent computations in the RVSDG can be accomplished using graph refinement. The overly conservative and sequential execution order of the input program can be relaxed by splitting and redirecting state edges to encode the results of analyses, such as alias analysis, in the graph. This exposes concurrent operations and program parts, which could subsequently be exploited for parallel execution.

3 EVALUATION

We have implemented *jlm*, a publicly available [12] prototype compiler that uses the RVSDG for optimizations. Jlm takes LLVM IR as input, constructs an RVSDG, transforms and optimizes this RVSDG, and destructs it again to LLVM IR. Jlm's LLVM IR support is currently limited to function, integer, floating point, pointer, array, structure, and vector types as well as their corresponding operations. Moreover, no support exists in the current implementation for exceptions and intrinsic functions.

We use the polybench 4.2.1 beta benchmark suite [11] to evaluate the RVSDG's usability and efficacy. This benchmark suite provides structurally small benchmarks, and therefore reduces the implementation effort, as well as the number and complexity of optimizations.

We use clang 4.0.1 [3] to convert C files to LLVM IR, pre-optimize the IR with LLVM's `opt`, and then optimize it either with `jlm`, or `opt` using different optimization levels. The optimized output is converted to an object file with LLVM's `llc`. The pre-optimization step is necessary to avoid a re-implementation of LLVM's `mem2reg` pass, since clang allocates all values on the stack by default. We implemented several optimizations, such as inlining, code motion, loop unrolling, and some operation simplifications. The experiments are performed on an Intel Core i7-4790 running Ubuntu 17.10. The core frequency is pinned to 1.0 GHz to avoid performance variations due to frequency scaling. The lowest frequency is chosen to avoid thermal throttling effects. All outputs of the benchmark runs are verified to equal the corresponding outputs of the executables produced by clang.

3.1 Performance

Figure 3 shows the speedup at five different optimization levels. The `O0` optimization level serves as baseline. The `O3-no-vec` optimization level is the same as `O3`, but without slp- and loop-vectorization. Optimization level `O3-no-vec-stripped` is the same as `O3-no-vec`, but the IR is stripped of named metadata and attribute groups before invoking llc. Since jlm does not support metadata and attributes yet, this optimization level permits us to compare the pure optimized IR against jlm without the optimizer providing hints to llc. We omit optimization level `O2` as it was very similar to `O3`. The gmean column in Figure 3 shows the geometric mean of all benchmarks.

The results show that the executables produced by jlm (gmean 1.22) are faster than `O1` (gmean 1.17), but slower than `O3` (gmean 1.38), `O3-no-vec` (gmean 1.28), and `O3-no-vec-stripped` (gmean 1.26). Optimization level `O3` tries to vectorize twenty benchmarks, but only produces measurable improvements for eight of them, namely atax, durbin, fdtd-2d, gemm, gemver, heat-3d, jacobi-1d, and jacobi-2d. Jlm would require a vectorizer to achieve similar speedups.

Disabling vectorization with `O3-no-vec` and `O3-no-vec-stripped` shows that jlm achieves similar speedups for fdtd-2d, gemm, heat-3d, javobi-1d, and jacobi-2d. The metadata transferred between the optimizer and llc only makes a significant difference for doitgen, durbin, fdtd-2d, floyd-warshall, gemm, jacobi-1d, and jacobi-2d. In the case of fdtd-2d, gemm, jacobi-1d, and jacobi-2d, performance drops below jlm. Jlm is outperformed by optimization level `O1` at four benchmarks: adi, durbin, ludcmp, and seidel-2d. We inspected

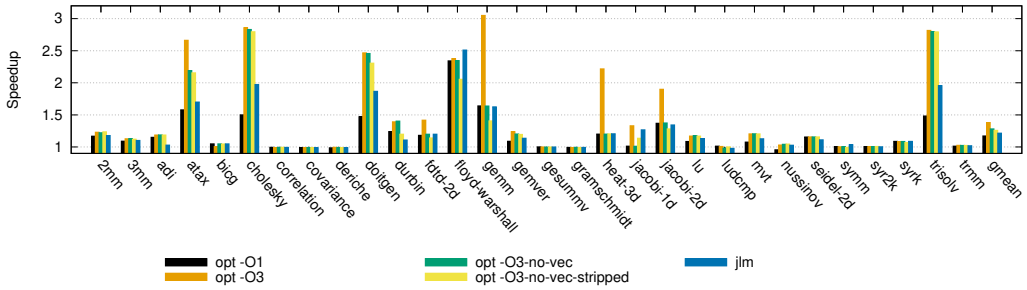


Figure 3: Speedup relative to `O0` at different optimization levels.

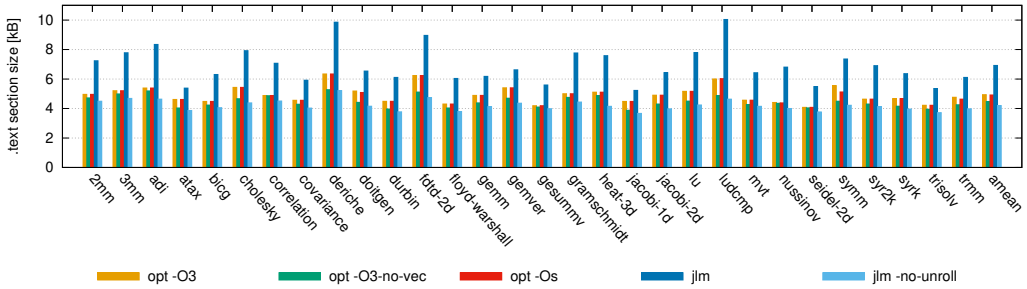


Figure 4: Code size at different optimization levels.

the output files and found that it is due to missing optimizations, such loop load elimination (`-loop-load-elim`) and loop idiom recognition (`-loop-idiom`).

Figure 3 shows that it is feasible to produce competitive code using the RVSDG, but also that more optimizations and analyses are required to reliably do so. The differences in performance are not due to inherent characteristics of the RVSDG, but can be attributed to missing analyses, optimizations, as well as heuristics for their application. Specifically, jlm requires more complex analyses, such as alias analysis and value range propagation, as well as more optimizations exploiting the results of these analyses to compete with mature compilers at more complex benchmarks.

3.2 Code Size

Figure 4 shows the code size for `O3`, `O3-no-vec`, `Os`, and for jlm with and without loop unrolling. The amean column shows the arithmetic mean of all benchmarks.

The code size for optimization levels `O3` and `Os` are almost identical, with noticeable differences only for doitgen, symm, or trmm. Moreover, the average code size of `O3-no-vec` is smaller than `Os`. Inspecting the individual optimizations for `O3` and `Os` reveals that both levels differ only in two optimization. Optimization level `O3` adds `-argpromotion`, which promotes by-reference arguments to scalars, and `-libcalls-shrinkwrap`, which conditionally eliminates dead library calls.

In comparison to `Os`, jlm produces ca. 40% bigger text sections. The experiments without loop unrolling show that this can be attributed to the naive heuristic used for this optimization. Jlm does not take code size into account and unrolls every inner loop unconditionally four times, leading to excessive code expansion. Avoiding unrolling completely results in text section sizes that are on average smaller than `Os`. This indicates that the excessive code size is due to naive heuristics and shortcomings in the implementation, but not to inherent characteristics of the RVSDG.

4 CONCLUSION AND FUTURE WORK

This paper presents the RVSDG as an IR for optimizing and parallelizing compilers. We implemented jlm, a publicly available compiler that uses the RVSDG for optimizations, and evaluate it in terms of performance and code size. The results suggest that the RVSDG combines the abstractions and benefits of data centric IRs with the CFG's advantages to generate efficient control flow.

In the future, we plan to employ the RVSDG for the extraction of TLP from programs. This requires further research into the encoding of analyses information into the IR, and the exploitation of the discovered concurrent computations.

REFERENCES

[1] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, 84–93.

[2] Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. 1996. Incremental Computation of Static Single Assignment Form. In *Proceedings of the International Conference on Compiler Construction*. Springer-Verlag, 223–237.

[3] Clang. 2017. Clang: A C Language Family Frontend for LLVM. https://clang.llvm.org. Accessed: 2017-12-13.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. K Zadeck. 1991. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. Technical Report.

[5] Neil E. Johnson. 2004. *Code size optimization for embedded processors.* Technical Report. University of Cambridge, Computer Laboratory.
[6] Alan C. Lawrence. 2007. *Optimizing compilation with the Value State Dependence Graph.* Technical Report. University of Cambridge, Computer Laboratory.
[7] LLVM. 2018. https://bugs.llvm.org/show_bug.cgi?id=31851. Accesssed: 2018-05-07.
[8] LLVM. 2018. https://bugs.llvm.org/show_bug.cgi?id=37202. Accesssed: 2018-05-07.
[9] LLVM. 2018. https://bugs.llvm.org/show_bug.cgi?id=31183. Accesssed: 2018-05-07.
[10] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture.* IEEE, 105–118.
[11] Louis-Noël Pouchet. 2017. Polybench/C 4.2. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/. Accessed: 2017-12-13.
[12] Nico Reissmann. 2017. jlm. https://github.com/phate/jlm. Accessed: 2017-12-13.
[13] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM, 12–27.
[14] James Stanier and Des Watson. 2013. Intermediate Representations in Imperative Compilers: A Survey. *ACM Computing Surveys (CSUR)* 45, 3 (2013), 26:1–26:27.
[15] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems* 13, 2 (1991), 181–210.
[16] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. 1994. Value Dependence Graphs: Representation Without Taxation. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM, 297–310.