Thesis for the degree of Philosophiae Doctor

Principles, Techniques, and Tools for Explicit and Automatic Parallelization

Nico Reissmann

January 28, 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The end of Dennard scaling also brought an end to technology scaling as a means to improve performance. Chip manufacturers had to abandon frequency and superscalar scaling as processors became increasingly power constrained. An architecture's power budget became the limiting factor to performance gains, and computations had to be performed more energy-efficiently. Designers turned to chip multiprocessors (CMPs) and developers began to employ specialized architectures, such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), to further improve performance while meeting the power envelope. The exploitation of parallelism in an energy-efficient manner became the only way forward.

Until the end of technology scaling, programs experienced transparent performance gains with every new processor generation. However, CMPs, GPUs, and FPGAs rely on the static extraction of parallelism to improve performance, and programs need to be modified to take advantage of these architectures. Thus, performance gains are no longer achieved transparently, and developers and tools are forced to face new, as well as long-neglected challenges in program parallelization. These challenges include the detection and encoding of potential parallelism in automatic approaches, application portability issues on GPUs, and performance portability issues on CMPs. It is essential to address these challenges, as the continuous increase in computer performance now solely relies on the exploitation of parallelism.

This thesis consists of three parts, each addressing one of the aforementioned challenges in program parallelization. The first part addresses the detection and encoding of potential parallelism in automatic approaches. It presents the Regionalized Value State Dependence Graph (RVSDG) as an alternative intermediate representation for optimizing and parallelizing compilers. The RVSDG exposes the hierarchical structure of programs and explicitly models the dependencies between computations, permitting the explicit encoding of

concurrent operations and program structures, such as conditionals, loops, and functions. This helps to expose the inherent parallelism in programs and its structures by employing well-known methods for the extraction of instruction level parallelism.

The second part addresses application portability issues on GPUs. A GPU's specialized architecture is optimized for highly regular data-parallel applications, but compromises program performance for workloads with irregular control flow, potentially leading to redundant code execution. We propose a control flow restructuring method to effectively eliminate repeated code execution on GPUs and potentially improve performance.

The third part addresses performance portability on CMPs. This issue arises as developers overfit their application to a specific architecture, which results in suboptimal performance for different program inputs or different architectures. We improve performance analysis for OpenMP programs by addressing the scalability challenges of the grain graph visualization method. We present an aggregation method for grain graphs that hierarchically groups related nodes into a single node. This aggregated graph can then be navigated by progressively uncovering nodes with performance issues, while hiding unrelated regions of the graph. This enhances productivity by enabling developers to understand performance problems of highly-parallel OpenMP programs more easily.

The insights and techniques developed by addressing these three challenges may result in improved methods and tools for the exploitation of parallelism. The RVSDG is a promising IR for parallelizing compilers, as it permits the encoding of concurrent computations. The grain graph offers a familiar structural view to developers along with the performance issues of a particular program. In the future, it is necessary to cast these ideas into mature tools to make them applicable in practice and foster further research.

List of Contributions

Thesis Articles

1. ***RVSDG: An Intermediate Representation for Optimizing Compilers***
   Nico Reissmann, Jan Christian Meyer, Magnus Själander.
   Unpublished Manuscript

2. ***Perfect Reconstructability of Control Flow from Demand Dependence Graphs***
   Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer.
   In *Transactions on Architecture and Code Optimization (TACO), Volume 11 Issue 4*, ACM, 2015

3. ***Efficient Control Flow Restructuring for GPUs***
   Nico Reissmann, Thomas L. Falch, Benjamin A. Björnseth, Helge Bahmann, Jan Christian Meyer, and Magnus Jahre.
   In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, 2016
   Winner of Outstanding Paper Award

4. ***Diagnosing Highly-Parallel OpenMP Programs With Aggregated Grain Graphs***
   Nico Reissmann and Ananya Muddukrishna.
   In *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*, Springer, 2018

Other Articles

1. ***A Study of Energy and Locality Effects using Space-filling Curves***
   Nico Reissmann, Jan Christian Meyer, Magnus Jahre.
   In *Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS) and IPDPS Workshops (IPDPSW)*, 2014.

2. ***Towards fine-grained dynamic tuning of HPC applications on modern multi-core architectures***
   Mohammed Sourouri, Espen Birker Raknes, Nico Reissmann, Johannes Langguth, Daniel Hackenberg, Robert Schöne, Per Gunnar Kjeldsberg.
   In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

3. ***Aggregating Large Grain Graphs For Improved OpenMP Productivity***
   Nico Reissmann, Magnus Jahre, Ananya Muddukrishna.
   In *Fourth International Workshop on Visual Performance Analysis (VPA)*, 2017.

4. ***RVSDG: An Intermediate Representation for the Multi-Core Era***
   Nico Reissmann, Jan Christian Meyer, Magnus Själander.
   In *11th Nordic Workshop on Multi-Core Computing (MCC)*, 2018.

5. ***Load Balancing Domain Decompositions of a Lattice-Boltzmann Proxy Application***
   Jan Christian Meyer, Janusa Ragunathan, Jørgen Valstad, Nico Reissmann.
   In *11th Nordic Workshop on Multi-Core Computing (MCC)*, 2018.

Acknowledgements

First and foremost, my deepest and sincerest thanks to Helge Bahmann and Jan Christian Meyer, for their invaluable guidance over the last twelve years. Both have challenged me to explore new and unfamiliar topics, broadened my perspective, sparked my love for compilers, and helped me to become a better engineer and researcher. My heartfelt thanks also to Magnus Själander for his advice and encouragement over the last few years, as well as for his interest in my work. Sincere thanks also to Gunnar Tufte for his support and the many insightful conversations. Gratitude is also due to my co-supervisors, Lasse Natvig and Per Gunnar Kjeldsberg, for their support throughout the years.

Moreover, I am grateful to Anne Berit Dahl, Birgit Sørgård, Berit Hellan, and Ellen Solberg for finding solutions instead of problems, and trying to reduce bureaucracy to a minimum. I would also like to thank my colleagues from the Computing group for making the fourth floor a fun and engaging workplace, and especially for the innumerable and illuminating discussions during lunch time and over payday drinks. They have been an indispensable part of my academic life, and contributed substantially to my social well-being and growth as a researcher.

A big round of thanks to my friends from Germany, Gothenburg, and Trondheim, for helping me to escape from the office and enriching my life. In particular, I would like to thank my long-term friends Carina Walter, René Kaiser, Jens Teuscher, Franziska Holzhauer-Pansa, Oliver Holzhauer, Aleksejs Sencenko, Marina Vazhnova, Dragana Laketić, Leif Tore Rusten, Elizabeth and Jacob Sturdy, David Fallon, and John Naliboff.

My deepest gratitude is also due to my parents, Luise and Friedmar Reissmann, to my brother's family, Mario, Yvonne, and Felix Reissmann, and to my close family, Birgit and Joachim Reissmann, as well as Urs, Anja, Nadja, and

Larissa Weber, for their boundless patience, unwavering support, and endless encouragement. Thanks for always having my back!

Last, but not least, I am eternally grateful to my better half, Ramona Enache, for her help throughout in preserving my sanity, providing support in times of need, and her constant love. You make me a better person!

Nico Reissmann
Trondheim, 4th November 2018

Contents

Part A.

Overview

1. Introduction

The continuous increase in computer performance over the last half-century catapulted society into the digital age. Computer systems are nowadays omnipresent and are a fundamental part of our lives. New advances in health, product manufacturing, transportation and energy, science and environmental modeling, and financial analysis, are all dependent on computer systems and their continuous increase in performance [63].

Figure 1.1 shows the processor performance growth from the end of the 1970s onwards. From the mid 1980s until around 2003, Moore's law [180] enabled processor performance to improve by three orders of magnitude at a rate of approximately 52% per year, *i.e.*, performance doubled every second year. Almost two orders of magnitude of this improvement are due to increased transistor frequency, while one order of magnitude can be attributed to microarchitectural advances [27].

This trend, however, decelerated from the beginning of 2003. Figure 1.1 shows a decline of yearly performance improvements from 2003 onwards to today's 3.5%, *i.e.*, performance doubles only every twenty years. The main cause behind this decline is the end of technology scaling. Transistor frequencies could no longer be increased without exceeding a chip's power budget and the performance return of more advanced microarchitectural features became negligible.

The need to limit a chip's power dissipation and to improve energy-efficiency pushed chip designers towards more decentralized and modular architectures. Chip multiprocessors (CMPs) along with more specialized architectures, such as Graphics Processing Units (GPUs) and special-purpose accelerators, emerged as alternatives to centralized and monolithic single-core processors. Instead of using transistors to improve single-thread performance within a single-core,

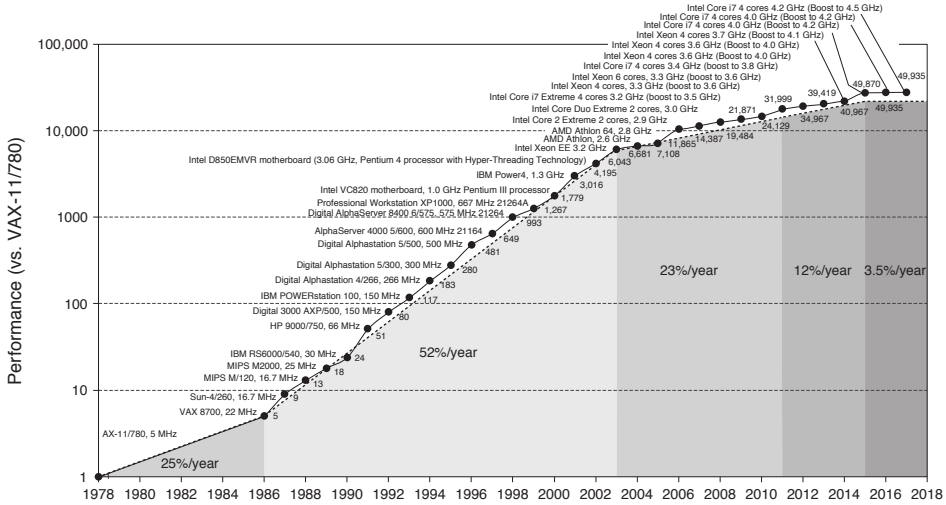


Figure 1.1.: 40 years of processor performance growth. Figure taken from [81].

designers use them to provide multiple cores on a chip to improve multi-thread performance.

This paradigm shift, however, was not transparent to the software. Single-core processors dynamically exposed and exploited a program's parallelism with increasingly complex microarchitectural structures. Programs written 40 years ago experienced with every new processor generation performance improvements without programmer intervention. For multi-core processors, however, the responsibility of exposing and exploiting more parallelism to further improve performance lies with the programmer or tools.

The static extraction and exploitation of parallelism forces programmers and tools to face new as well as long-neglected challenges in code parallelization [48]. Concurrent programming requires additional reasoning of developers about the correctness of their programs as new error classes, such as data races and deadlocks, arise. New performance bottlenecks, such as synchronization overheads and load balancing issues, may occur as developers port their programs to different architectures. Moreover, automatic code parallelization is still stuck in the sequential programming era as tools employ sequential representations inadequate for the automatic exposure of concurrent computations. In order

to mitigate a new software crisis [64, 6] and progress on as well as eventually overcome these challenges, novel principles and techniques are needed that eventually equip developers with mature tools.

This thesis addresses some of these challenges and is based on four articles:

Article B1 – *RVSDG: An Intermediate Representation for Optimizing Compilers*

Article B2 – *Perfect Reconstructability of Control Flow from Demand Dependence Graphs*

Article C1 – *Efficient Control Flow Restructuring for GPUs*

Article D1 – *Diagnosing Highly-Parallel OpenMP Programs With Aggregated Grain Graphs*

The articles B1 and B2 address the exposure issue by presenting the Regionalized Value State Dependence Graph (RVSDG) as an alternative IR for optimizing and parallelizing compilers. The RVSDG is a single unified IR that normalizes program representation and exposes the hierarchical structure of a program. It explicitly encodes the dependencies (and therefore their lack of) between operations and high-level structures, such as loops, conditionals, and functions. This helps to expose the inherent parallelism in programs and its structures, as well as avoids many support data structures necessary for optimizations.

The article C1 and D1 address application portability to GPUs and performance portability on CMPs, respectively. Concretely, the first article proposes a control flow restructuring method to avoid repeated code execution on GPUs due to thread divergence, whereas the second article improves OpenMP performance analysis by addressing scalability challenges of the grain graph visualization method.

The remainder of this thesis is structured as follows: The four aforementioned articles are attached in Part B, C, and D. Specifically, Part B contains the articles *RVSDG: An Intermediate Representation for Optimizing Compilers* and *Perfect Reconstructability of Control Flow from Demand Dependence Graphs*, Part C contains the article *Efficient Control Flow Restructuring for GPUs*, and Part D contains the article *Diagnosing Highly-Parallel OpenMP Programs With Aggregated Grain Graphs*. The remainder of the thesis puts the articles in a

context and provides the necessary background to understand them. In particular, the rest of Part A discusses the scope of this thesis in Chapter 2, summarizes the research in detail and discusses the concrete research contributions in Chapter 3, provides necessary background knowledge to understand the articles in Chapter 4, and concludes and suggests further research directions, as well as attempts an outlook in Chapter 5.

2. Motivation and Scope

Dennard Scaling [54] was the driving force behind the steady growth of processor performance between the end of the 1970s until 2003. It posits that the overall power consumption can be kept constant as transistor dimensions shrink by lowering both voltage and current accordingly. This meant that manufacturers could with every new processor generation pack more transistors on a chip and increase the clock frequency without a significant increase in overall power consumption.

At the same time, computer architects used the ever growing abundance of transistors to build increasingly complex structures that dynamically expose and exploit the instruction level parallelism (ILP) inherent in programs. They incorporated increasingly deeper pipelines, superscalar and out-of-order execution, as well as aggressive branch prediction, register renaming, and dynamic memory disambiguation to maximize ILP exploitation. The result was that processor performance doubled approximately every two years for over 40 years. Figure 2.1 shows that from the beginning of the 1980s, transistor count increased exponentially, and until 2003, chip frequency as well as single-thread performance increased accordingly.

This trend, however, came to an end in 2003. It was no longer sustainable to simply increase processor frequency and build ever more complex structures to dynamically exploit ILP. Figure 2.1 shows that the increase in frequency and single-thread performance started to decelerate from 2003 onwards. The reason for this decline is due to several factors, known as the ILP, memory, power, and complexity wall [144]:

1. **The ILP Wall**: Superscalar architectures dynamically extract ILP from a *single* flow of control of a sequential instruction stream. Studies by Wall [221] and Lam *et al.* [105] showed that, except for data-parallel applications with simple control flow, the ILP from a single flow of control

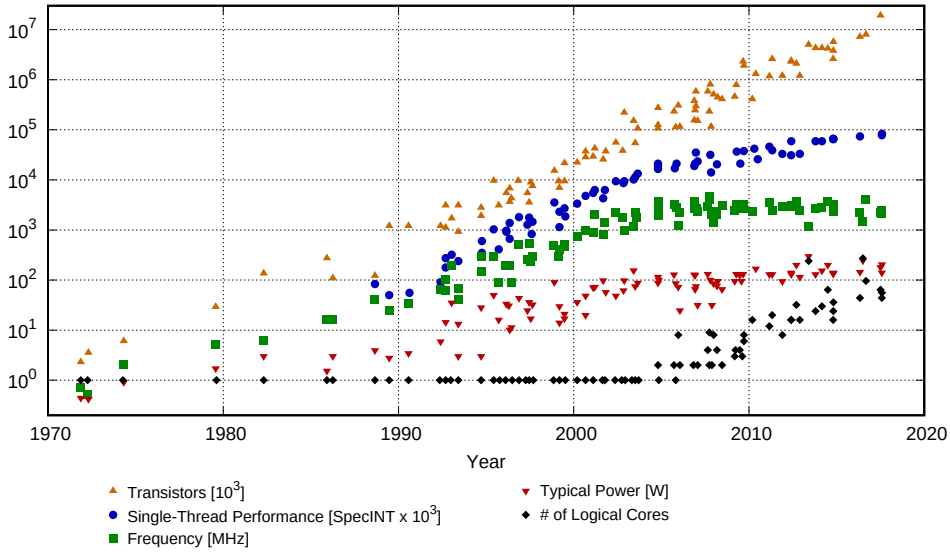


Figure 2.1.: 50 years of microprocessor trend data. Figure taken from [175].

is limited to 4-8 instructions per cycle. This sets an upper bound for superscalar processor scaling beyond which very little performance gains could be expected.

2. **The Memory Wall**: Both processor and DRAM clock rates improved exponentially, but processor rate substantially outpaced DRAM rate. This meant that main memory access latency grew also exponentially, as the difference of two diverging exponentials is exponential [228, 126]. The introduction of caches mitigated this issue by exploiting a program's temporal and spatial locality. The caches grew bigger and cache hierarchies deepened as the gap between processor and main memory widened. Nowadays, processors dedicate as much as 50% of the die area to caches.

3. **The Complexity Wall**: The monolithic design of superscalar processors required ever more complex and broadcast-intensive structures to dynamically exploit more ILP, maintain a coherent program state, and provide precise exceptions. However, such structures scale poorly with the number of exploited instructions [210, 240]. In addition, the global wires

on a chip do not scale with transistor size [84], which reduces the distance signals can propagate as frequency increases. This means that fewer structures can be accessed within a single clock cycle, limiting the exploitable ILP as chips become more communication bound [2]. These factors along with the ILP wall severely diminished overall performance improvements even as design complexity, effort, and costs soared [18, 149].

4. **The Power Wall**: For each process generation, Dennard scaling enabled an increase in transistor performance of ca. 40% and the doubling of transistors while keeping the overall power consumption constant [27]. However, Dennard scaling does not account for the increased leakage current as transistor dimensions shrink. This current is negligible for micrometer technology, but becomes prominent for smaller transistor sizes and leads to an exponential increase in relative static power dissipation [33]. Dennard scaling does also not account for the increased complexity of microarchitectures and the poor scaling of wires, both of which increase overall power dissipation. These factors lead to static power becoming a major contributor to overall power dissipation [207] and resulted in projections that chip power density would approach those of nuclear reactors [171].

The ILP, memory, complexity, and power wall forced chip manufacturers to abandon frequency and superscalar processor scaling. This was necessary as processor complexity became unmanageable, while the return on overall performance compared to the investment became negligible. The end of frequency scaling stopped memory latency from worsening and halted the increase in power dissipation. This trend can be observed in Figure 2.1. Processor frequency and power dissipation stagnated from 2003 onwards.

However, the end of frequency scaling and the ILP Wall also meant that further performance improvements would have to come solely from the extraction of coarse-grained parallelism, *i.e.*, data- and task-level parallelism. The exploitation of multiple, independent code regions, achieved without an increase in architectural complexity, became essential to further improve performance.

2.1. Modern Architectures

Chip manufacturers turned to multi-core architectures to answer the need for exploiting coarse-grained parallelism without increasing architectural complexity. Instead of relying on monolithic and complex superscalar architectures, manufacturers placed multiple copies of simpler conventional processors on a single die. This addressed the complexity wall by switching to a modular and decentralized design, while enabling the exploitation of coarse-grained parallelism in addition to ILP. Moreover, it permits continued performance scaling, while keeping power dissipation stagnant, therefore increasing energy efficiency, *i.e.*, performing more computations for the same amount of energy. The switch from superscalar to multi-core architectures can be observed in Figure 2.1. Processors with multiple cores, *i.e.*, chip multiprocessors (CMPs), became mainstream around 2003, which coincides with the stagnation of frequency and power dissipation.

In addition to CMPs, other and more specialized architectures, such as Graphics Processing Units (GPUs) or Field Programming Gate Arrays (FPGAs), emerged to accelerate certain classes of applications more energy-efficiently. GPUs target applications with abundant data parallelism, while special-purpose accelerators can be employed to accelerate domain specific kernels. The remainder of this section takes a closer look at the three architectures and their application domains.

2.1.1. Chip Multiprocessors

Designers addressed the consequences of the increasingly complex uniprocessor designs with the introduction of chip multiprocessors, and started to replicate simpler conventional cores on the same die. Instead of only focusing on exploiting the ILP from a single flow of control with a monolithic and complex processors, they shifted focus to the exploitation of coarse-grained parallelism from several independent flows of control. CMPs target applications that either have abundant task level parallelism, such as web or database servers, or ample data level parallelism, such as numeric or multimedia applications. Such applications consist of ample coarse-grained parallelism that is exploitable with multiple cores [15].

Ideally, designers would like to extend the effect of Moore's law into an exponential growth of cores. However, CMPs rely on the abundance of coarse-grained parallelism that is exploited by threads in a shared-memory programming model. This makes them an ideal fit for embarrassingly parallel application domains, but limits their use for applications with low task-level parallelism, complex and data-dependent control flow, and irregular memory access patterns. Such applications often consist of a significant sequential part and can effectively not be parallelized.

2.1.2. Graphics Processing Units

Graphics Processing Units (GPUs) originated as graphics accelerators and became over the years increasingly more programmable [24]. GPUs evolved to highly parallel architectures suitable for accelerating data parallel applications. They consist of multiple computational units with each containing several processing units, and feature a high bandwidth memory system with smaller and simpler caches as compared to conventional CPUs (see Section 4.2 for more details).

Nowadays, GPU vendors offer support for general purpose computations by providing the necessary software tools [50, 146]. This resulted in a wide adoption of GPUs in the scientific and multimedia domain, as applications from these domains tend to exhibit abundant data parallelism. GPUs have been used to accelerate linear programming [195], fluid simulations [104, 176], ray tracing [119], compression [10], and medical image processing [192, 193, 194], and are employed as accelerators in five of the top ten systems on the June 2018 list of most powerful supercomputers [208].

2.1.3. Special-Purpose Accelerators

Another way to increase performance and/or energy-efficiency is to implement dedicated hardware in the form of accelerators. Such specialized logic can offer multiple order of magnitude greater energy efficiency at a similar or greater level of performance than general purpose architectures [44, 184, 185]. However, hardware development is tedious and costly, and is therefore mostly

done for critical kernels of important application domains, such as cryptography or multimedia, and for markets where the design costs can be amortized. Accelerators are already common in more energy sensitive platforms such as smartphones. For example, modern devices can include dedicated hardware for image [160] and natural language processing [129], neural networks and machine learning [102, 209], as well as context awareness [129].

In the future, the integration of Field-Programmable Gate Arrays (FPGAs) with CPUs, *e.g.*, as announced by Intel [92], might also permit to leverage the advantages of specialized logic outside the embedded systems domain. A shift towards specialized hardware can already be experienced in today's research communities. Researchers propose ample accelerators for important kernels and different application domains [11], such as neural networks [215, 72], graph analytics [214, 12, 19], and sparse linear algebra [213, 216].

2.2. Programming Modern Architectures

Central to all these systems is the software that runs on them. In the past, programmers mostly used sequential programming models and relied on Dennard Scaling and Moore's law to gain performance. Every new processor generation provided speedup transparent to the software due to technology and superscalar processor scaling. However, the end of this scaling also brought an end to transparent performance gains, or as Herb Sutter famously put it: "the free lunch is over" [203].

In addition to fine-grained ILP, modern architectures try to exploit coarse-grained parallelism by exposing hardware structures in the form of replicated cores, single-instruction multiple-data (SIMD) units, or accelerators. Coarse-grained parallelism, however, is not extracted dynamically, and programs must be explicitly modified to take advantage of these structures. Thus, the paradigm shift from dynamic ILP exploitation to the exploitation of coarse-grained parallelism is not transparent to the software, and sequential programs experience no longer automatic performance improvements. Contrarily, sequential programs might even perform worse due to decreased single-thread performance [27, 204]. This is particularly concerning as most existing mainstream software is written in sequential programming models.

A program's coarse-grained parallelism can either be encoded **explicitly** in the source code by developers, or **automatically** extracted and encoded by tools. The remainder of this section discusses explicit and automatic parallelization methods for CMPs, GPUs, and accelerators. It provides the basis for a discussion of these method's parallelization challenges in Section 2.3.

2.2.1. Explicit Parallelization

Explicit parallelization is a developer's task of modifying a sequential program for parallel execution. It is the programmers responsibility to extract independent tasks and manage their communication and synchronization. This section summarizes explicit approaches for CMPs and GPUs. It omits a discussion of explicit parallelization approaches for accelerators, as they lie outside the scope of this thesis.

Chip Multiprocessors

CMPs rely on the exploitation of coarse-grained parallelism by threads in a shared-memory programming model. Programmers or tools are required to expose this parallelism by partitioning concurrent code sections into threads, and manage the communication and synchronization of these threads at runtime. However, explicitly programming CMPs at this fundamental level is error-prone and burdensome, as the programmer must manage every explicit detail of code partitioning, communication, and synchronization [58].

Consequently, a cornucopia of different programming models try to raise the abstraction level above the simple usage of threads, as well as manual communication and synchronization management. Such models include compiler directive-based parallelization, such as OpenMP [53], the incorporation of parallelization constructs into serial languages, such as Cilk Plus [170], libraries with parallel routines, such as Intel MKL [128], concurrent data structures, such as Intel TBB [101], domain-specific programming models, such as MapReduce [163], and abstractions for expressing parallelism, such as Wool [70].

A common feature of all these models is that they are partially implicit, *i.e.*, the methods' implementation carries the parallelization of the code to various

degrees [58]. For example, in OpenMP it is the programmer's responsibility to identify code regions for parallelization, but the implementation's responsibility to schedule and load balance these threads.

Graphics Processing Units

The two most popular methods for programming GPUs are OpenCL [146] and CUDA [50]. These methods handle an application's sequential part on the CPU, while the data parallel kernels are offloaded to the GPU. The kernels are executed in parallel by a large number of threads, performing individual instruction execution in lock-step. Both methods demand from the programmer to divide a problem into subproblems that can be solved independently, while each subproblem is solved cooperatively in parallel.

Other methods emerged as alternatives to the low-level abstractions provided by CUDA or OpenCL. Directive-based approaches [229], such as OpenACC [145] or HMPP [59], permit programmers to annotate plain C to indicate the regions that should be offloaded to other devices like GPUs. Similarly to parallelization approaches for CMPs, these methods try to raise the abstraction level and hide implementation details from the programmer.

2.2.2. Automatic Parallelization

Automatic parallelization refers to a tool's task to autonomously extract and encode parallelism from a sequential program. It is the tools sole responsibility to find enough parallelism, encode it, and produce a (performant) parallel execution schedule. This section summarizes automatic approaches for CMPs and accelerators. It omits a discussion of automatic parallelization methods for GPUs, as they lie outside the scope of this thesis.

Chip Multiprocessors

Auto-parallelizing compilers detect parallelizable code regions in sequential programs and transform them into parallel code. They try to exploit different

types of parallelism, such as loop level, pipeline, or divide and conquer parallelism, from loops and/or functions, and transform these constructs into data- and/or task-parallel code. Here, we closer examine automatic loop vectorization, and the non-speculative extraction of thread-level parallelism (TLP) from loops.

Vectorization: The introduction of SIMD units in microprocessors lead to the exploitation of data-level parallelism using vectorization [108]. Many modern compilers, such as LLVM [41], GCC [138, 141, 173], or ICC [21], incorporate vectorization passes that transform sequential to vectorized code.

Two methods for code vectorization exist: *loop* and *superword level parallelism* (SLP) vectorization. Loop vectorization extracts parallelism across loop iterations and requires loops in a "vectorizable" form, preferably with no loop-carried dependencies. Compilers often need to perform advanced loop transformations, such as loop fission, interchange, skewing, splitting, and peeling, in order to transform loops to such a form.

SLP vectorization exploits the ILP within a basic block to pack instructions into vectors and is applicable to loop bodies as well as straight-line code. In case of loops, the body of the loop only needs to be unrolled to expose the ILP between instructions before the vectorizer can pack them. In contrast to loop vectorization, the SLP vectorizer is not dependent on advanced loop transformations to produce vectorizable code, but relies on simple techniques for exposing ILP that are already present in modern compilers.

Both schemes rely on advanced (inter-procedural) analyses, such as alias, alignment, and data analyses to expose independent loop iterations or instructions, as well as on an accurate profitability analysis to successfully produce performant vector code.

TLP Extraction: The advent of CMPs puts automatic extraction of TLP into the limelight, but despite for its pressing need, it remains a hard problem with limited success. ICC offers it as an option, but still fails to produce significant speedups [137, 222]. Even though TLP extraction has not been successful so far, progress has been made. We therefore closer examine the recent advances in this field to provide the basis for the discussion of its challenges in Section 2.3.

In the past, computer scientists tried to produce multi-threaded code from sequential programs, but were only successful for specific application domains, such as scientific and numerical applications. These applications contain counted loops that manipulate very regular, analyzable structures and contain mostly predictable array accesses. In many cases, these loops have no data dependences between iterations or are easily transformed into such a form. Such DOALL parallelism can be easily exploited by implementing each loop iteration as a separate thread [89].

Challenges arise for loops with dependencies across individual iterations. Separate threads can still be assigned to the iterations of such DOACROSS loops, but synchronization is required among these threads to satisfy loop-carried dependencies. The synchronization overhead might outweigh the parallelization benefits, and an accurate profitability analysis is necessary for such parallelization efforts to succeed. Here, we examine two recent approaches, Decoupled Software Pipelining (DSWP) [148] and Helix [35], that managed two extract significant speedup from parallelizing loops of sequential code.

DSWP is an automatic method that statically extracts TLP by exploiting the pipeline parallelism of loops [148]. The DSWP algorithm generates a loop's dependence graph, computes the acyclic strongly connected component (SCC) graph of this dependence graph, and partitions the SCC graph into threads. DSWP ensures that the loop's critical path dependence chain, *i.e.*, the longest path in the SCC graph, is assigned to the same thread. This increases execution efficiency and provides latency tolerance. While DSWP managed to achieve significant speedups, it relied on architectural extensions for inter-thread communication. Moreover, its scalability is limited to the number of SCCs and recurrences found in the loop body, which are typically smaller than the number of loop iterations [162]. Raman *et al.* [161] addresses the last issue by enhancing DSWP to also exploit DOALL parallelism for SCC graph nodes that feature no loop-carried dependencies.

Helix [35] is another fully automatic technique that extracts TLP from loop iterations by carefully selecting the most profitable loops and distributing its iterations in round-robin order to different threads. Helix hides the overhead of inter-thread communication using profile-guided loop selection based on a simple heuristic and the exploitation of a modern processors' simultaneous multi-threading capabilities. It reduces the inter-core signaling overhead by

employing helper threads to prefetch synchronization signals. Helix manages to achieve impressive speedups for irregular sequential programs, but limits parallel execution to one loop at a time. Moreover, performance gains fail to scale beyond four cores due to communication latency and additional hardware support seems to be required to overcome this limitation [34].

Special-Purpose Accelerators

Special-purpose accelerators have the potential to provide an order of magnitude increase in performance for a fraction of a general-purpose processor's energy. Their main drawback is the tedious and (prohibitively) expensive design process. Conventionally, designers use hardware description languages, such as VHDL [218] or Verilog [205], to describe their hardware designs. These languages require advanced hardware expertise and only offer a very low abstraction level, leading to long development times [140]. This restricts the design of custom accelerators to markets where the design costs can be amortized, such as for embedded systems.

Another way to realize accelerators is to employ high-level synthesis (HLS). HLS tools use a high-level language, such as C, C++, or SystemC, as input to (semi-)automatically output a circuit specification, commonly in a hardware description language. These tools raise the abstraction level and permit to drastically reduce hardware design costs, as they relieve developers of the complex and error prone hardware design and debugging task. In combination with FPGAs, HLS holds the promise for software developers to harvest the performance and energy efficiency benefits of specialized hardware without the need to become hardware designers.

2.3. Parallelization Challenges

Performance improvements happen no longer transparently to the software, and the exploitation of parallelism is essential to higher performance. Modern

architectures require programs with statically encoded coarse-grained parallelism to exploit exposed hardware structures. This requirement forces developers and tools to face new as well as long-neglected challenges in code parallelization [48].

Conceptually, the task of parallelization can be divided into three major steps, regardless of whether the programmer explicitly encodes the parallelism or tools automatically extract it:

1. **Detection**: The first step is to detect parallelizable program sections. This can be as simple as identifying a readily parallelizable loop, or as complex as exchanging an inherently sequential algorithm with a parallel one. Automatic approaches require often advanced analyses to succeed.

2. **Exposure**: The second step is to expose the detected parallelism. In the explicit case, this is done by the developer changing the original source code, whereas in the automatic case, the tools' internal program representation is modified.

3. **Exploitation**: The third step is to exploit the exposed parallelism by mapping it to an architecture. This involves adjusting the problem to the concrete hardware parameters in order to gain optimal performance. If done explicitly, this might lead to performance portability issues [151], whereas the automatic case requires profitability analysis based on accurate cost models of the underlying architecture to succeed.

All three steps are necessary to exploit coarse-grained parallelism on modern architectures. While explicit parallelization requires the developer to perform at least the first two steps, and maybe partially the third, all three steps must be performed by automatic approaches. The remainder of this section discusses some challenges that arise for the identified architectures with explicit and automatic parallelization.

2.3.1. Chip Multiprocessor Challenges

The challenges faced by tools and developers on CMPs are different for explicit and automatic code parallelization. This section identifies some of these

challenges. It is separated into two sections, one for each of the parallelization methods.

Explicit Parallelization

Explicit parallel programming puts the burden of code parallelization on the programmer [91]. At its lowest abstraction level, *i.e.*, using threads and locks, the challenges are immense. Programmers must not only detect and expose concurrent program sections and manually partition them into threads, but also manage their communication and synchronization using locks. This amount of control enables programmers to precisely map programs to the underlying architecture, but is (prohibitively) expensive. In addition to challenges from sequential programming, programmers face new challenges in terms of correctness and performance portability [204]:

1. **Correctness**: The shared-memory programming model is inherently non-deterministic [112], as inadequate thread synchronization can result in data races [86], deadlocks, and livelocks. These types of errors are hard to find and understand, as their non-deterministic behavior makes them difficult to reproduce.

2. **Performance Portability**: Concurrent programming introduces new potential sources of performance bottlenecks, such as lock contention, synchronization overheads, lock convoys, load balancing issues, and resource over/under-subscription [204, 131]. These performance issues are often difficult to identify and might (re-)appear for different architectures or program inputs.

As discussed in Section 2.2.1, an abundance of programming models emerged for different use cases and forms of parallelism that raise the abstraction level above the simple usage of threads and locks. These models try to hide complexity behind abstraction layers and leave architectural exploitation to compilers or runtime systems. This mitigates the correctness and performance issues, but does not solve them.

The cost-efficient programming of parallel hardware, however, becomes increasingly important as performance improvements happen no longer transparently. The abundant availability of CMPs as a low-cost commodity renders

the writing of parallel software and the parallelization of sequential programs a significant cost factor in the development of systems [127]. Explicit parallel programming becomes a productivity challenge as software developers struggle with low abstractions, poor tool support [131, 182], and the restructuring of programs to extract the necessary parallelism. While automatic code parallelization could mitigate this productivity challenge, the next section shows that the currently employed tools have already problems detecting and encoding the necessary parallelism. Some researchers consider the current state of tools along with a CMPs' reliance on statically extracted parallelism even the beginning of a new software crisis [64, 6].

Automatic Parallelization

Automatic code parallelization can be performed by different methods for various levels of parallelism. This section examines two methods, vectorization and TLP extraction, and identifies their respective challenges.

Vectorization: In 2011, Maleki *et al.* [124] analyzed the auto-vectorizers of the GCC, ICC, and XLC compiler on a set of 151 loops. They found that only a fraction of the loops were vectorized, despite the widespread availability of SIMD units for over a decade and ample research on improving auto-vectorization [109, 187, 188, 189, 190]. The three identified main causes are:

1. **Inter-procedural Analysis**: A lack of or inaccurate inter-procedural analyses to disambiguate pointers and determine array dependencies. These analyses are necessary to perform transformations that in turn permit code vectorization.

2. **Code Transformations**: The inability of compilers to perform transformations that would enable vectorization or make it profitable. The authors identified memory layout change, code replacement, and data alignment transformations as the main impediments to code vectorization.

3. **Profitability Analysis**: A profitability analysis based on accurate cost models is necessary to determine the benefits of vectorization. It is essential that the analysis correctly predicts speedup of vectorized code,

as mispredictions can lead to missed opportunities, or worse, to performance degradation. Studies by Pohl *et al.* [153, 154], however, showed no or only a weak correlation between cost predictions and actual performance gains for the profitability analysis of LLVM's vectorizer. Another study performed by Pohl *et al.* [155] in 2018 showed similar results for GCC.

This slow progress leads to cases where auto-vectorized code is still significantly outperformed by manually vectorized code [38]. Researchers try to further mitigate these issues by employing OpenMP directives to improve data dependency analysis [96], extending vectorization to non-isomorphic instruction sequences [157], or improving the profitability of vectorization [156].

TLP Extraction: The challenges of TLP extraction are similar to the challenges of auto-vectorization. Specifically, it relies on:

1. **Inter-procedural Analyses**: An accurate inter-procedural analyses is essential to the success of TLP extraction as it helps to relax dependencies and therefore uncover independent operations. For example, the Helix project relied on a state-of-the-art inter-procedural pointer analysis of the whole program [75].

2. **Code Transformations**: Both DSWP and Helix try to exploit loop-level parallelism by inserting synchronization points to satisfy loop-carried dependencies and exploit the parallelism in the rest of the code. In order to minimize these synchronization points and improve performance, parallelizers rely on optimizations, such as privatization or induction variable elimination, to remove loop-carried dependencies or arrange them to a sufficient distance such that no conflicts occur for concurrent iterations.

3. **Profitability Analyses**: TLP extraction relies on an accurate profitability analyses to determine the best loops for parallelization. In addition, an accurate analysis is required to determine whether the concurrent execution of iterations outweighs inter-thread communication between iterations. For example, Helix [35] uses a heuristic based on the dynamic loop nesting graph and a speedup model derived from Amdahl's law [83] to determine the best loops for parallelization.

Auto-vectorization and TLP extraction require accurate inter-procedural analyses, enabling code transformations, and accurate cost models to determine profitability. Both schemes require global information to resolve dependencies between operations within loops, even though they only transform local loop structures. This global task is overly complicated by the usage of the Control Flow Graph (CFG) [4] as dominant IR for analyses and transformations [199].

The CFG is an inherently sequential IR that complicates the **exposure of concurrent computations** as it lends itself towards sequential languages. It highlights the structure of a function's control flow and is simple to construct as well as to destruct as it encodes program counter progress of Von-Neumann architectures. Even though the CFG is widely used in the literature and in mainstream compilers, such as LLVM or GCC, it is criticized as an IR for optimizing and parallelizing compilers [71, 97, 98, 111, 224, 234, 233]. Specifically, the CFG is criticized for the following deficiencies:

1. It is incapable of representing inter-procedural information. It requires additional IRs, *e.g.*, the call graph, to represent such information.

2. It provides no structural information about a procedure's body. Important structures, such as loops, and their nesting needs to be constantly (re-)discovered for optimizations, as well as normalized to make them amenable for transformations. For example, the first two steps of the Helix compiler is to detect loops for parallelization and to normalize them [35].

3. It emphasizes control dependencies, even though many optimizations are based on the flow of data. This is somewhat mitigated by translating it to static single assignment (SSA) form [52], but in turn requires SSA restoration passes [43] to ensure SSA invariants.

4. It is an inherently sequential IR. The operations in a basic block are always listed in sequential order, even if they are not dependent on each other. Moreover, this sequentialization also exists for coarse-grained structures such as loops, as two independent loops can only be encoded in sequential order. Thus, the CFG is by design incapable of explicitly encoding independent operations.

5. It provides no means to encode additional dependencies other than control and true data dependencies. Other information, such as loop-carried dependencies or alias information, must be regularly recomputed and/or memoized in addition to the CFG.

These deficiencies render the CFG inadequate as an IR for analyses and optimizations in a time where the static exploitation of parallelism is paramount to improve performance. Its inherent sequential nature, the lack of a global program view and exposure of important structures, as well as the inability to encode (the lack of) dependencies between operations and structures overly complicate code parallelization. The CFG requires ample supporting data structures, such as call graphs, loop trees, and alias representations, to provide the necessary information to complex optimizations, such as auto-vectorization or TLP extraction. This unnecessarily increases the complexity of these transformations and leads to missed parallelization opportunities.

2.3.2. Graphics Processing Unit Challenges

GPUs evolved to highly parallel architectures that are optimized for accelerating data-parallel applications. The CUDA and OpenCL programming models offer interfaces to efficiently program these architectures, but leave the actual parallelization of the code to developers. It is the developer's task to identify suitable data-parallel kernels in a program and parallelize them so that they execute correctly and efficiently on GPUs. Programmers therefore face the same challenges for GPUs in terms of correctness [114, 20, 85, 239] and performance portability [121, 120] as for CMPs, and need techniques as well as tools to address these challenges.

In addition to correctness and performance portability, developers face the challenge of efficient **application portability**. The GPU's specialized architecture is optimized for highly regular data-parallel applications, such as scalar vector multiplication, but less regular applications must handle the GPU's architectural idiosyncracies to not compromise performance. An example of such an idiosyncracy is *thread divergence* [177]. On a GPU, the same instruction is executed in lock-step on single-instruction and multiple-data (SIMD)

units for a group threads[1]. These threads can diverge in the presence of control flow and lead to severe performance degradation [110].

2.3.3. High-Level Synthesis Challenges

HLS tools synthesize parallel hardware from sequential programs. In order to generate efficient hardware, these tools need to discover operations and code regions that can be executed in parallel. This is the same task that parallelizing compilers face and consequently the challenges are very similar. Specifically, HLS tools must cope with:

1. **Concurrent Computation Exposure**: Modern HLS tools, such as LegUp [36] or Bambu [152], use a variant of the Control Data Flow Graph (CDFG) [139] as main IR. The CDFG tries to mitigate the sequential nature of the CFG by replacing the sequence of operations in basic blocks with the Data Flow Graph (DFG) [55]. The DFG is an acyclic graph that represents the flow of data between individual operations.

   While the CDFG relaxes the strict ordering within a basic block, it does not expose ILP beyond basic block boundaries or between program constructs. This severely limits the discoverable parallelism [140, 234] and HLS tools try to mitigate this problem by employing various optimizations, such as loop unrolling, loop flattening [103, 230, 74], loop pipelining [115], and if-conversion [122, 212] to expose more parallelism. Moreover, HLS tools started to support standard software parallelization techniques, such as pthreads [32] and OpenMP, to expose more coarse-grained parallelism [42].

2. **Complex control flow**: Sequential input programs often contain complex control flow, such as data-dependent branches, function calls, and nested loops, that leads to poor synthesis results [234].

3. **Input Canonicalization**: A lot of the input languages of HLS tools permit designers to express the same algorithm using widely different programming constructs and styles. This expressiveness offers developers freedom of choice for implementing their algorithms, but compli-

[1] See Section 4.2 and Chapter C1 for more details

Table 2.1.: Parallelization Challenges

	Explicit Parallelization	Automatic Parallelization
CMPs	‡ Correctness ‡ Performance Portability	* Inter-procedural analyses * Code transformations † Concurrent Computations ‡ Profitability analysis
GPUs	‡ Correctness ‡ Performance Portability ‡ Application Portability	Not discussed
Accelerators	Not discussed	† Concurrent Computations † Complex control flow † Input canonicalization

* Detection † Exposure ‡ Exploitation

cates synthesis and can lead to unpredictable quality of synthesis results [191].

2.4. Summary

For over two decades, Dennard Scaling enabled software-transparent performance gains at an unprecedented rate and catapulted society into the digital age. As the driving factors behind these gains, technology and superscalar scaling, ceased to be sustainable, chip manufacturers turned to the exploitation of coarse-grained parallelism to further improve performance. This paradigm shift was not transparent to software. Developers and tools could no longer rely on the processor to dynamically extract parallelism, but must statically detect, expose, and exploit it to gain performance. This forces developers and tools to face new as well as long-neglected challenges in code parallelization. Table 2.1 summarizes these challenges. It shows a division between explicit and automatic parallelization. Explicit parallelization faces challenges in terms of parallelism exploitation, whereas automatic parallelization faces already challenges in terms of parallelism detection and exposure.

The reason for explicit parallelization to only face challenges in terms of exploitation is that developers are responsible for parallelism detection and exposure. This manual encoding permits effective exploitation of modern architectures, but is exorbitantly expensive as concurrent programming is demonstrably more difficult than sequential programming [204]. It further causes performance portability issues as developers fit the parallelized program to the underlying architecture and/or input parameters. These issues render the parallelization of sequential programs time consuming and error-prone, resulting in low programmer productivity. Thus, developers require mature techniques and tools to support them in parallelizing sequential programs.

The alternative to explicit parallelization is automatic parallelization. It holds the promise to relieve developers of the time consuming and error-prone task of manually parallelizing code and therefore boost productivity, but currently fails to accomplish this as it already faces challenges in terms of parallelism detection and exposure. In particular, modern compilers still employ an inherently sequential IR for optimizations and analyses, complicating the exposure of concurrent computations. This was acceptable in an era where performance improvements happened transparently to software, but hinders progress in a time when the static exploitation of parallelism is paramount.

This thesis addresses three of the aforementioned challenges. The first addressed challenge is the exposure of concurrent computations, the second challenge is appplication portability to GPUs, and finally, the third challenge is performance portability on CMPs. The next section describes the detailed contributions of this thesis, and how these contributions are linked to those three challenges.

3. Research Contributions

The performed research culminated in nine articles, of which four are included in this thesis[1]:

Article B1 – ***RVSDG: An Intermediate Representation for Optimizing Compilers***
Nico Reissmann, Jan Christian Meyer, Magnus Själander
Unpublished Manuscript

Article B2 – ***Perfect Reconstructability of Control Flow from Demand Dependence Graphs***
Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer
In *Transactions on Architecture and Code Optimization (TACO), Volume 11 Issue 4*, ACM, 2015

Article C1 – ***Efficient Control Flow Restructuring for GPUs***
Nico Reissmann, Thomas L. Falch, Benjamin A. Björnseth, Helge Bahmann, Jan Christian Meyer, and Magnus Jahre
In *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, 2016
Winner of Outstanding Paper Award

Article D1 – ***Diagnosing Highly-Parallel OpenMP Programs With Aggregated Grain Graphs***
Nico Reissmann and Ananya Muddukrishna
In *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*, Springer, 2018

[1] See Section 3.3 for the other five articles

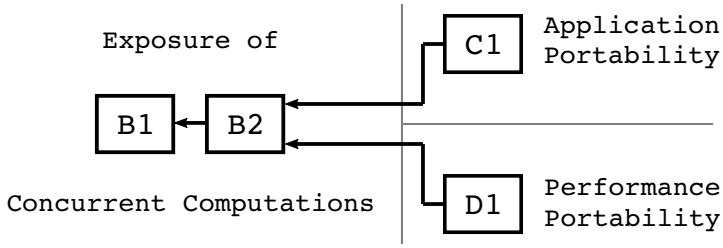


Figure 3.1.: Addressed challenges and relationship between articles.

The four articles are categorized into three groups according to the challenges mentioned in Secion 2.4. The first two articles are both about the RVSDG and therefore are grouped in the same part of this thesis. The other two articles have each their own part. Throughout the remainder of this thesis, each article is identified by its part and number, *i.e.*, B1, B2, C1, and D1, for the sake of brevity.

Figure 3.1 shows the relationship of the four articles as a simple dependency tree, categorized according to the challenges they address. Article B1 is the root of this tree. It presents the RVSDG and parts of its construction and destruction, exemplifies its utility by presenting optimizations, and provides an evaluation of a practical implementation. Article B2 depends on article B1, as it details the control flow restructuring of RVSDG construction and the control flow recovery of RVSDG destruction. Together, both articles provide the theoretical basis for the RVSDG, its construction, as well as destruction, and the foundation of a practical RVSDG compiler implementation.

The insights gained from the control flow restructuring in article B2 could be further applied to other research fields. Article C1 uses control flow restructuring to reduce the impact of control flow divergence on GPUs, and article D1 uses it to reduce the visual and computational overhead of grain graphs in order to simplify OpenMP performance analysis. Thus, article C1 and D1 share a common origin in article B2, even though both articles are unrelated and positioned in different research fields.

The rest of this chapter discusses the detailed research contributions of the four articles in Section 3.1, provides an overview of the produced tools in Section 3.2, and presents the other published articles in Section 3.3.

3.1. Thesis Articles

Article B1 – RVSDG: An Intermediate Representation for Optimizing Compilers

This article presents the RVSDG as an IR for optimizing and parallelizing compilers, as well as its construction, *i.e.*, its generation from the input language, and destruction, *i.e.*, the generation of the target language. The RVSDG is a hierarchical acyclic multi-graph where nodes represent computations and edges the dependencies between these computations. Its nodes can represent simple operations, such as addition, subtraction, load, and store, but also complex computations, such as conditionals, loops, and functions. Its edges serve two purposes: first, they model the flow of data between computations, and second, they are used to enforce a sequential order for operations with side-effects. Specifically, the RVSDG has the following properties:

1. It is a **single unified IR** that is capable to represent an entire translation unit or program.

2. It normalizes programs by representing them in a **canonical representation**, where different input language constructs are mapped to the same IR constructs.

3. It **enforces SSA form** and therefore avoids SSA restoration.

4. It **exposes the hierarchical structure of programs**, *e.g.*, the nesting of loops.

5. It **exposes the parallelism** inherent in programs as operations are only dependent on the necessary edges and complex computations, such as conditionals or loops, are modeled explicitly.

The article presents two optimizations in detail that exploit these properties, and evaluates the IR in terms of performance, code size, compilation overhead, as well as engineering effort using a prototype compiler.

Article B2 – Perfect Reconstructability of Control Flow from Demand Dependence Graphs

This article focuses on the construction and destruction of the RVSDG's intra-procedural constructs. Since the RVSDG only supports conditionals and tail-controlled loops, input languages with more complex control flow require control flow restructuring to make them amenable to the IR. This article introduces an algorithm that structurally converts any intra-procedural control flow to a form that permits RVSDG construction.

For RVSDG destruction, the article proposes two novel algorithms for intra-procedural control flow extraction. The first algorithm only produces intra-procedural control flow that consists of conditionals and tail-controlled loops, and therefore mirrors the constructs of the RVSDG. The second algorithm is capable to extract *any* control flow from the RVSDG, but requires the IR to be in predicate continuation form (PCF). This normal form ensures that there only exist unique paths from nodes producing control flow predicates to the conditionals and loops. The article proves termination and correctness of the algorithms, as well as that the original CFG, of which an RSVSDG was created from, can be perfectly reconstructed. It further empirically evaluates the performance, the representational overhead at compile time, and the reduction in branch instructions for the destruction algorithms.

Article C1 – Efficient Control Flow Restructuring for GPUs

This article uses the insights gained from the control flow restructuring step in the RVSDG construction and applies them to GPUs to reduce the impact of branch divergence. In GPUs, divergent branches cause performance degradation by under-utilizing the execution pipeline. If also unstructured control flow is present in addition to these branches, then performance can further degrade as this results in repeated code execution.

This article extends and applies the control flow restructuring algorithm from article B2 to GPUs. The proposed algorithms convert unstructured to structured control flow to effectively eliminate redundant code execution and potentially improve execution time. The algorithms are empirically evaluated in terms of performance and representational compile-time overhead.

Article D1 – Diagnosing Highly-Parallel OpenMP Programs with Aggregated Grain Graphs

This article uses the insights gained from the restructuring algorithm of article B2 to aggregate large grain graphs and simplify OpenMP performance analysis. The performance of OpenMP programs can vary for different program inputs and between different architectures. In order to optimize performance, developers need methods that help them to adapt their programs to these varying parameters. Grain graphs emerged as such a method. Instead of presenting the programmer with intricate details of the runtime system, grain graphs mirror the nested structure of OpenMP programs by presenting performance issues in a familiar fork-join perspective. However, for highly-parallel OpenMP programs, grain graphs can easily contain more than 100000 nodes, rendering viewers irresponsive and overwhelming developers.

This article simplifies performance analysis for such large graphs by proposing an aggregation method. This method matches recurring patterns in the grain graphs, groups related nodes, and ultimately reduces the graph to a single node. The aggregated graph can then be navigated by progressively uncovering problematic groups, while hiding unproblematic nodes from the developer. The article empirically evaluates the aggregation method using standard OpenMP programs from SPEC OMP 2012, Barcelona OpenMP Task Suite 2.1.2, and Parsec 3.0.

3.2. Frameworks and Tools

Aside from the articles, the research created or extended three frameworks and tools worth mentioning:

Jive: The jive compiler back-end implements the RVSDG intermediate representation. It provides the abstractions, data structures, and necessary interfaces for tools that want to use the RVSDG for compilation and code optimization. The project is a joint development effort with Helge Bahmann.

Repository: https://github.com/phate/jive.git

Jlm: The jlm framework is a collection of tools that provide an RVSDG-based compiler and optimizer for the LLVM IR. It consists of the following components:

1. **libjlm**: The core library that uses the Jive compiler back-end to implement the RVSDG-based LLVM IR. It provides the implementation for LLVM's type system, its operators, as well as the construction and destruction methods necessary to convert from and to the original LLVM IR, respectively.

2. **jlm-opt**: An optimizer for the LLVM IR that uses the RVSDG for analyses, optimizations, and transformations.

3. **jlm-print**: A pretty printer that can emit the program or outputs of the individual compilation stages in various formats. It can print the RVSDG IR as XML, which is the input format of the RVSDG viewer.

4. **jlc**: A C compiler that uses the RVSDG for analyses, optimizations, and transformations.

The project has so far been a sole development effort.

Repository: https://github.com/phate/jlm.git

RVSDG viewer: The RVSDG graph viewer is a graphical user interface program that can be used to inspect RVSDGs. Its main use is for debugging RVSDG optimizations and transformations. The development effort has so far solely been done by Asbjørn Djupdal.

Repository: https://github.com/phate/rvsdg-viewer.git

3.3. Other Articles

Several other research articles were produced during the course of this PhD. They are not part of this thesis, as either their extended version is included, or they lie outside the thesis' scope:

1. ***A Study of Energy and Locality Effects using Space-filling Curves***
   Nico Reissmann, Jan Christian Meyer, Magnus Jahre.
   In *Proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS) and IPDPS Workshops (IPDPSW)*, 2014.

2. ***Towards fine-grained dynamic tuning of HPC applications on modern multi-core architectures***
   Mohammed Sourouri, Espen Birker Raknes, Nico Reissmann, Johannes Langguth, Daniel Hackenberg, Robert Schöne, Per Gunnar Kjeldsberg.
   In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

3. ***Aggregating Large Grain Graphs For Improved OpenMP Productivity***
   Nico Reissmann, Magnus Jahre, Ananya Muddukrishna.
   In *Fourth International Workshop on Visual Performance Analysis (VPA)*, 2017.

4. ***RVSDG: An Intermediate Representation for the Multi-Core Era***
   Nico Reissmann, Jan Christian Meyer, Magnus Själander.
   In *11th Nordic Workshop on Multi-Core Computing (MCC)*, 2018.

5. ***Load Balancing Domain Decompositions of a Lattice-Boltzmann Proxy Application***
   Jan Christian Meyer, Janusa Ragunathan, Jørgen Valstad, Nico Reissmann.
   In *11th Nordic Workshop on Multi-Core Computing (MCC)*, 2018.

4. Background and Related Work

This chapter provides the background and context necessary to understand the articles of this thesis. It covers compiler intermediate representations, Graphics Processing Units, and the OpenMP application programming interface.

4.1. Compiler Intermediate Representations

Compilers are language translators that read a program written in one programming language, *i.e.*, the source language, and translate it into a program of another language, *i.e.*, the target language [3]. It is often the case that the source language is a high-level programming language, such as C, C++, or Java, and the target language the machine instructions of an instruction set architecture, such as x86, ARM, or PowerPC. The important tasks of a compiler are the reporting of errors in the source program, the semantically correct translation of input to output programs, and the optimization of programs to improve code quality.

The internal structure of a compiler is generally split into stages where each stage performs a specific task of the compilation process, as exemplified in Figure 4.1. The input language specific tasks, such as lexical, syntactic, and semantic analysis, are performed in the front-end, whereas optimizations and code generation are performed in the back-end. At the heart of a compiler are intermediate representations, illustrated in Figure 4.1 as little rectangles between stages. These data structures represent programs in memory throughout compilation, connect the individual compiler stages, as well as highlight and expose program properties that are important for a specific stage.

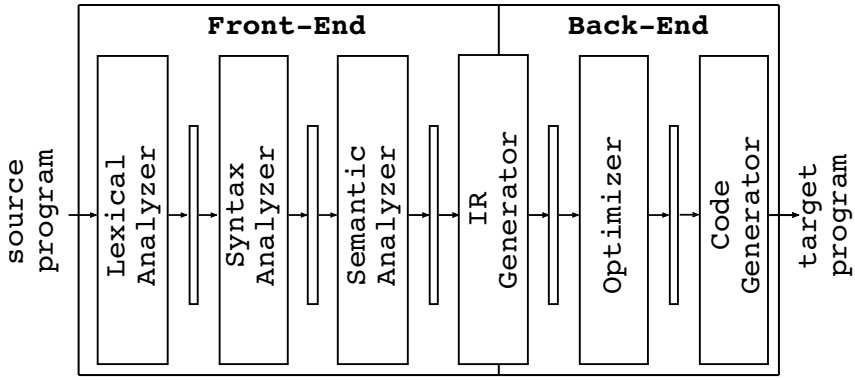


Figure 4.1.: Simplified view of a compiler.

As program optimizations are an integral part of every modern compiler, a cornucopia of IRs has been presented in the literature to better expose desirable program properties for optimizations. Section 2.3.1 and 2.3.3 already highlighted the strengths and weaknesses of the C(D)FG, and we discuss in the remainder of this section the more prominent alternatives that emerged over the years.

4.1.1. Program Dependence Graph

The Program Dependence Graph (PDG) [71] combines control and data flow within a single representation. It features data and control flow edges, as well as statement, predicate, and region nodes. Statement nodes represent operations, predicate nodes represent conditional choices, and region nodes group other nodes with the same control dependency. If a region's control dependencies are fulfilled, then its children could potentially be executed in parallel. Horwitz *et al.* [88] extended the PDG to model inter-procedural dependencies by incorporating procedures into the graph.

The PDG improves upon the CFG by employing region nodes to relax the overly restrictive sequence of operations. The relaxed sequence combined with the unified representation of data and control dependencies simplifies complex optimizations, such as code vectorization [17] or TLP extraction [148, 178], but also increases maintenance cost during and after transformations.

The unified data and control flow representation results in a large number of edge types, five in Ferrante *et al.* [71] and four in Horwitz *et al.* [87], which need to be maintained to ensure the graph's invariants. The PDG suffers from aliasing and side-effect problems, as it supports no clear distinction between data held in registers and memory. This complicates or can even preclude construction altogether [98]. Moreover, program structure and SSA form still need to be discovered and maintained.

4.1.2. Value (State) Dependence Graph

The Value Dependence Graph (VDG) [224] abandons the explicit representation of control flow and only models the flow of values using ports. Its nodes represent simple operations, or program constructs, such as conditionals or functions. Loops are modeled as recursive functions. The VDG is implicitly in SSA form and abandons the sequential order of operations from the CFG, as each node is only dependent on its values. However, modeling only data flow raises a significant problem in terms of preservation of program semantics, as the "evaluation of the VDG may terminate even if the original program would not..." [224].

The Value State Dependence Graph (VSDG) [97, 98] addresses the VDG's termination problem by introducing state edges. These edges are used to model the sequential execution of operations with side-effects. In addition to nodes for representing simple operations and selection, it introduces nodes to explicitly represent loops. Like the VDG, the VSDG is implicitly in SSA form, and nodes are solely dependent on required operands, avoiding a sequential order of operations. However, the VSDG supports no inter-procedural constructs, and its selection operator is only capable to select between two values based on a predicate. This complicates destruction, as selection nodes must be combined to express conditionals. Even worse, the VSDG represents all nodes as a flat graph, which simplifies optimizations [98], but has a severe effect on evaluation semantics. Operations with side-effects are no longer guarded by predicates, and care must be taken to avoid duplicated evaluation of these operations. In fact, for graphs with stateful computations, lazy evaluation is the only safe strategy [111]. The restoration of a program with an eager evaluation semantics complicates destruction immensely, and requires a detour

over the PDG to arrive at a unique CFG [111]. Zaidi *et al.* [233, 234] adapted the VSDG to spatial hardware and sidestepped this problem by introducing a predication-based eager/dataflow semantics. The idea is to effectively enforce correct evaluation of operations with side-effects by using predication. While this seems to circumvent the problem for spatial hardware, it is unclear what the performance implications would be for conventional processors.

The RVSDG solves the VSDG's eager evaluation problem by introducing regions into the graph. These regions permit to model control flow constructs as nested nodes, and permit to guard operations with side-effects. This avoids any possibility of duplicated evaluation, and in turn simplifies RVSDG destruction. Moreover, nested nodes permit to explicitly encode the hierarchical structure of a program into the graph, further simplifying optimizations.

4.2. Graphics Processing Units

Graphics Processing Units (GPUs) were originally designed to offload the CPU from performing graphics computations, but found their way into general-purpose programming due to the introduction of programming models such as CUDA or OpenCL. Their single instruction, multiple thread (SIMT) execution model combines multi-threading, MIMD, SIMD, and instruction-level parallelism to accelerate data-parallel kernels [81].

In CUDA, developers must decide upon kernel invocation on the total number of threads as well as a partition of these threads into *thread blocks* as follows:

```
kernel<<<nBlocks, nWarps>>>(...parameters...)
```

where `nBlocks` is the number of thread blocks and `nWarps` the number of threads, called *warps* in CUDA terminology, per block. Upon execution, the thread blocks are assigned by the GPU scheduler to *streaming multiprocessors* (SM) and the SIMD instructions from the warps run on individual execution units in lock-step. A vertical cut of a warp's instruction stream, corresponding to a single element executed by a core, is called a *CUDA thread*.

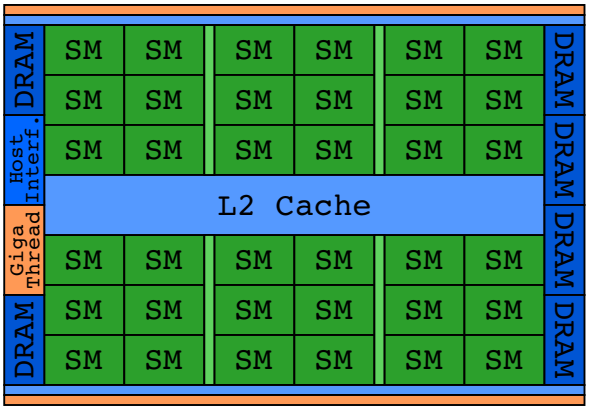


Figure 4.2.: Simplified block diagram of a GPU.

Figure 4.2 shows a simplified block diagram of a GPU. Green depicts execution units, blue register, caches, and memory, and orange scheduling and dispatch units. As shown, a GPU consists of multiple SMs[1], a shared L2 cache, as well as DRAM support and an interface to the host. The Giga Thread scheduler is responsible to distribute thread blocks to the different SMs.

Dropping down one more level of detail, Figure 4.3 shows a simplified block diagram of an SM. It consists of data and instruction caches, registers, as well as a warp scheduler that selects and issues instructions from a ready warp to the execution units, *e.g.*, cores, load/store (LD/ST), or special function units (SFUs). Thus, GPUs consist of two levels of hardware schedulers: a thread block scheduler that assigns thread blocks to SMs, and a warp scheduler within an SM that schedules instructions from warps. The warp scheduler can pick any ready instruction from any warp as warps are independent from another. The scheduler includes a scoreboard of ready instructions, since memory instructions have variable latency due to cache or TLB misses. The basic assumption is that enough threads are available and multi-threading can be used to hide the memory latency of stalled instructions and increase the utilization of SMs.

[1] The number of SMs varies between different microarchitectures.

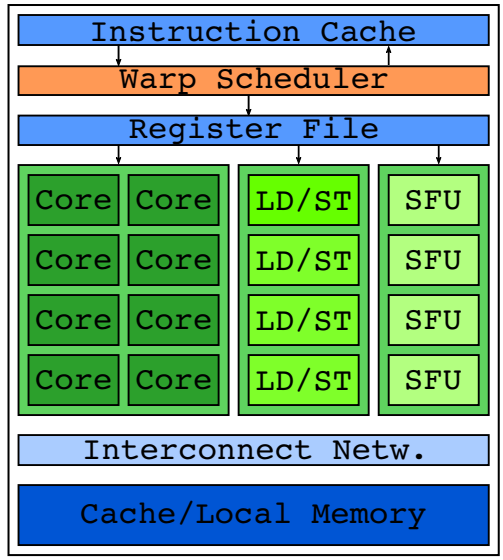
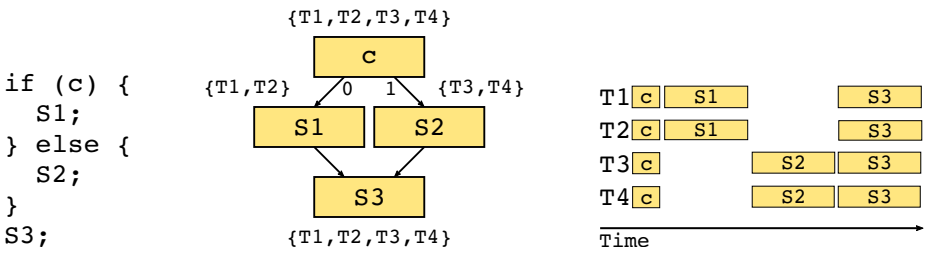


Figure 4.3.: Simplified block diagram of an SM.

4.2.1. Control Flow Divergence on GPUs

On GPUs, all threads of a warp execute the same instruction in lock-step. Upon the execution of a conditional branch, however, it could happen that a warp's threads diverge and individual threads have to follow different execution paths. GPUs handle such divergent threads by executing the individual paths sequentially, masking out threads that do not take a path. This causes performance degradation as the individual execution units of an SM are under-utilized.

Figure 4.4 illustrates the problem of control flow divergence for a warp of four threads. Figure 4.4b shows the CFG of the code in Figure 4.4a. Upon execution of the conditional branch instruction in basic block c, the threads in the warp diverge. Thread $T1$ and $T2$ continue execution at basic block $S1$, whereas thread $T3$ and $T4$ continue execution at basic block $S2$. The threads only reconverge before executing basic block $S3$. Figure 4.4c shows a possible execution schedule, illustrating the sequentialization of the $S1$'s and $S2$'s execution. It shows the under-utilization of the execution pipeline, with thread



(a) Code (b) Control Flow Graph (c) Execution schedule

Figure 4.4.: Control flow divergence.

$T3$ and $T4$ idling while thread $T1$ and $T2$ execute, and similarly, thread $T1$ and $T2$ idling while $T3$ and $T4$ execute. This under-utilization of the execution pipeline due to divergent threads is a major performance bottleneck on GPUs [56, 77, 226, 9].

4.3. Grain Graphs

Grain graphs are an OpenMP performance visualization method that emerged as an alternative to runtime system or thread oriented visualizations. Instead of depicting program execution from a runtime system perspective, grain graphs visualize task and parallel for-loop chunk instances, collectively called *grains*, in a fork-join perspective that is familiar to developers. Grains are annotated with source code locations and are highlighted if they suffer performance issues. The familiar perspective along with the grains' annotations simplifies OpenMP performance analysis, as it permits programmers to immediately connect performance issues to program structure.

4.3.1. Structure

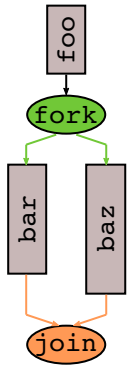
Grain graphs are acyclic directed graphs that consist of nodes representing the creation, synchronization, and computation of task and parallel for-loop chunk instances, as well as edges representing the dependencies between these events. Figure 4.5 shows the different nodes and edges for example OpenMP

```
#pragma omp task
{ /* foo */
  #pragma omp task
  { /* bar */ }
  ...
  #pragma omp task
  { /* baz */ }
  ...
  #pragma omp taskwait
}
```

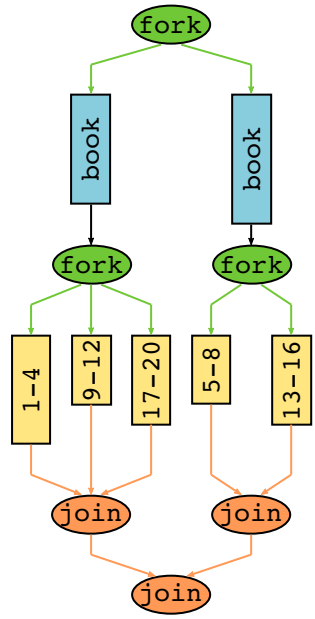
(a) OpenMP task program.

```
#pragma parallel for \
schedule(static,4) \
num_threads(2)
for (int i = 1; i <= 20; i++)
{
  ...
}
```

(b) OpenMP parallel for-loop program.



(c) Grain graph for program in Figure 4.5a.



(d) Grain graph for program in Figure 4.5b

Figure 4.5.: OpenMP programs and corresponding grain graphs. Examples taken from [135].

programs. A grain graph can contain five different node types: fork, join, fragment, book-keeping, and chunk nodes, as well as three different edge types: creation, synchronization, and continuation. Fork and join nodes represent the creation and synchronization of tasks, respectively, and fragment nodes the execution of tasks between these two events. Creation edges connect fork nodes with a child's fragment node, synchronization edges connect the fragment node of children with the join node of the parent, and continuation edges connect fragment nodes to fork or join nodes, and denote the continuation of execution after the spawning of or synchronization with children, respectively.

Figure 4.5c shows the graph with these nodes and edges for the code in Figure 4.5a. In the code, the task foo creates the two tasks bar and baz, performs computations between their creation, and synchronizes with its child tasks. The grain graph in Figure 4.5c shows these events and contains three fragment nodes (grey), *i.e.*, one for each task, one fork node (green) that spawns the two child tasks bar and baz, as well as one join node (orange) that synchronizes the parent task foo with bar and baz.

In code with parallel-for loops, book-keeping nodes denote the computation of splitting an iteration space into chunks, and chunk nodes represent the computation performed by the set of iterations in a chunk. Figure 4.5d shows the graph with these nodes for the code in Figure 4.5b. In the code, the iterations of the for loop are distributed among two threads, where each thread executes four iterations of the loop at a time. The grain graph in Figure 4.5d reflects these events and contains 5 chunk nodes (yellow), *i.e.*, one chunk node for every four iterations, and two book keeping nodes (blue), *i.e.*, one for each of the two threads.

4.3.2. Problem Diagnosis

Grain graph nodes are annotated with the source code locations of the events they represent, as well as with performance metrics measured during profiling and derived post profiling. Profiled metrics include a grain's execution time, cache miss ratio, memory latency, as well as event timings, such as grain creation and synchronization. These metrics are used to compute derived metrics, such as global and local critical paths, work deviation, instantaneous par-

allelism, memory hierarchy utilization, load balance, scatter, and parallel benefit.

Metrics are visually encoded in the graph to ease problem diagnosis. For example, the length of a fragment, book-keeping, or chunk node correlates with the length of its execution time, and its fill color reflects problem severity.

Programmers diagnose problems by switching between different views. Each view represents a different performance problem, and encodes this problem in the graph by dimming non-problematic grains and highlighting problematic grains with a fill color that correlates to problem severity. Grains are inferred as problematic if the value of the corresponding metric crosses a sensible threshold. Figure 4.6 shows the low parallel benefit view of a grain graph, highlighting grains in red that suffer from parallelization overhead as their execution time is too short. These nodes should effectively be executed sequentially to reduce this overhead.

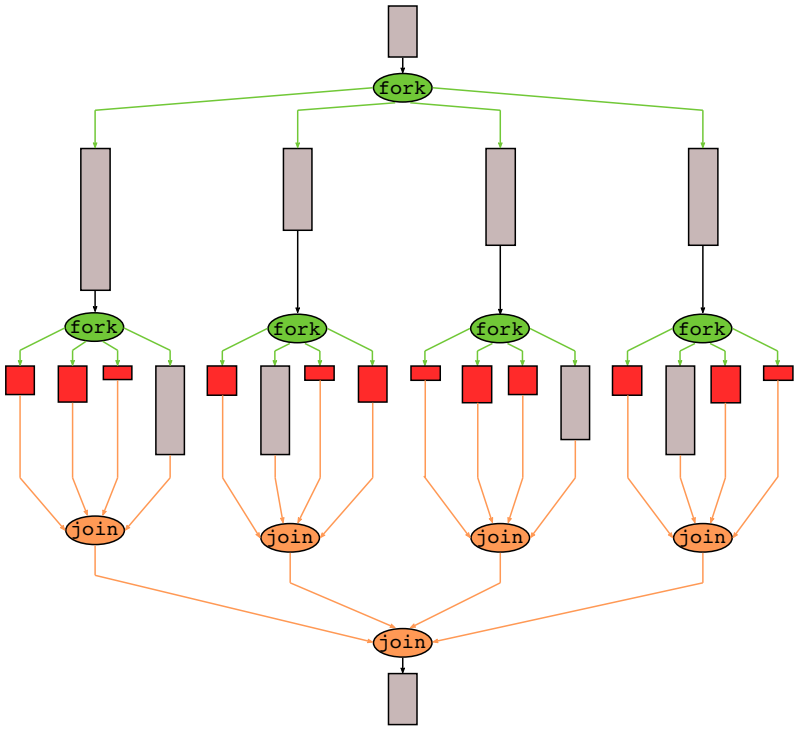


Figure 4.6.: A grain graph with red nodes that exhibit low parallel benefit.

5. Concluding Remarks

The end of technology scaling left the extraction and exploitation of parallelism as the only way forward to improve performance. Chip designers abandoned single-core processor scaling and embraced heterogeneous architectures as viable alternatives. CMPs, GPUs, and special-purpose accelerators emerged and permitted further performance improvements without exceeding power budgets.

This architectural shift, however, was not transparent to software. Sequential programs experience no automatic performance gains running on these systems and must be parallelized to take advantage of the provided structures. This parallelization must statically detect, expose, and exploit concurrent computations inherent in applications, and can either happen explicitly or automatically, *i.e.*, concurrent computations are extracted by developers or tools, respectively. The challenges of explicit extraction lie within the exploitation of parallelism, while automatic extraction already faces challenges in detecting and exposing parallelization opportunities. Specifically, developers face challenges in program correctness as well as performance and application portability, while tools provide insufficient analyses and miss transformations to detect the parallelism, as well as employ inherently sequential IRs. This thesis' research addressed three of these challenges.

Its first part addressed the detection and encoding of concurrent computations in automatic approaches. We presented the Regionalized Value State Depedence Graph (RVSDG) as a new IR for compilers. The RVSDG is a data-flow centric IR that elides most of the control flow from the original program. It is capable of representing an entire translation unit as a hierarchical acyclic graph, enforces desirable properties, such as SSA, explicitly encodes important structures, such as loops, and is capable of exposing a program's concurrent computations. We further devised RVSDG construction and destruction algorithms. RVSDG construction enables the generation of RVSDGs from

programs with any kind of complex control flow, while RVSDG destruction permits the recovery of arbitrarily complex intra-procedural control flow. Together, these algorithms enabled the creation of jlm, a prototype compiler that uses the RVSDG for optimizations. Jlm consumes and produces LLVM IR and can be used as a drop-in replacement for LLVM's optimization stages. This opens up the way for further research in terms of parallelism detection and exploitation.

The second part addressed application portability issues on GPUs by mitigating the effect of branch divergence. We proposed a control flow restructuring method that converts unstructured to structured control flow by inserting predicates and early reconvergence points to reduce the impact of divergence. The method effectively eliminates repeated code execution on GPUs and potentially improves performance.

The third part addressed performance portability on CMPs. We presented an aggregation method for large grain graphs to simplify the performance analysis of OpenMP programs. The method hierarchically groups related nodes, reducing an entire graph to a single node. The aggregated graph can then be navigated by progressively uncovering nodes with performance issues, while hiding unrelated graph regions. This enhances productivity by speeding up the visual analysis of large grain graphs and enabling programmers to understand problems in highly-parallel OpenMP programs with less effort than before.

The developed techniques and methods of the three addressed challenges pave the way for further research. The rest of this section discusses some of this research and attempts an outlook into the future.

5.1. Future Work

This section discusses future research for the individual directions of the four articles. Specifically, Section 5.1.1 discusses future RVSDG research, Section 5.1.2 the impact of redundant execution on GPUs, and Section 5.1.3 presents research ideas on OpenMP performance analysis using grain graphs.

5.1.1. Regionalized Value State Dependence Graph

The RVSDG is an IR that enables the encoding of concurrent computations, as its state and value edges explicitly model the dependencies between operations and/or higher-level program structures, such as loops or functions. This section outlines ideas for the exposure of these computations and the subsequent exploitation of the exposed parallelism. It further presents other research challenges that are orthogonal to code parallelization.

Parallelization Detection

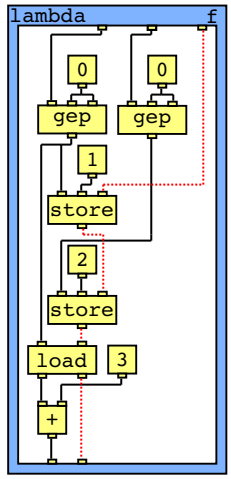
The exposure of independent computations is accomplished by relaxing the overly conservative execution order of the original sequential program. Simple graph rewriting techniques are used to encode analyses results in the graph by splitting state edges. This renders unrelated computations independent from each other and exposes opportunities for automatic parallelization. The remainder of this section presents ideas to relax the execution schedule of sequential input programs.

Separating Unrelated Side-Effects: The current, and naive, implementation uses a single state to sequentialize all operations with side-effects throughout construction. This is overly conservative as different computations can have mutually exclusive side-effects. For example, the side-effect of a nonterminating loop is unrelated to a load that is not dereferencable. These side-effects can be modeled by separate states. This would already result in the exposure of more parallelism after construction as, for example, loops with no memory operations would become independent from other loops with memory operations.

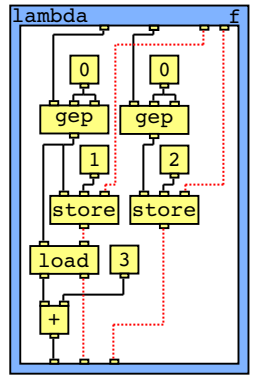
Invariant Edge Redirection: The RVSDG's explicit representation of higher-level constructs as structural nodes combined with the explicit encoding of dependencies enables the modeling of the interrelations between conditionals, loops, and functions. RVSDG construction or optimizations direct edges into structural nodes to serve as inputs for nested computations, but the values represented by these edges might not be modified. This renders these edges invariant with respect to the structural node and the users of the corresponding structural node's output can be diverted. A canonical example are

```
typedef struct s{
  unsigned int i;
} s;

unsigned int
f(s * p, s * q)
{
  p->i = 1;
  q->i = 2;
  return p->i + 3;
}
```

(a) Code



(b) RVSDG after construction



(c) RVSDG after encoding

Figure 5.1.: Encoding alias information in the RVSDG

two terminating loops that do not modify memory. These loops would be sequentialized after construction due to the state edge that is routed through them. This state edge is invariant and can be redirected to render these loops independent from each other.

Alias Analysis: Alias analysis divides a program's memory into disjoint sets of locations that cannot alias one another. These alias classes can be encoded in the RVSDG with the help of state edges as shown in Figure 5.1. The RVSDG in Figure 5.1b shows the code of Figure 5.1a after RVSDG construction. It contains a single state edge (red dotted line) that sequentializes the loads and stores to preserve the semantics of the original program. This sequentialization might be too conservative. If alias analysis can determine that pointers p and q never point to the same storage location, then it is possible to represent these disjoint locations with two distinct state edges. Figure 5.1c shows the RVSDG after the encoding of this non-alias information. Function f has now two states: one for pointer p and q. The respective load and store operations are now sequentialized with the corresponding state edges, which enables further optimizations. The load follows now directly a store with the same address value and both states are directly connected without another in-

tervening node. This permits to replace the output of the load with the value of the store, namely one, which in turn permits the constant folding of the add operation. Thus, the encoding of non-alias information in the RVSDG helps to expose concurrent computations, and may enable other optimizations.

Parallelization Exploitation

Independent computations are explicitly represented in the RVSDG after analyses and transformations relaxed the original sequential execution order. The dependencies between operations are encoded as value and state edges, reducing the detection of parallel computations to simple dependency checks between RVSDG nodes. The remainder of this section discusses some implications for known parallelization methods and ideas for exploiting the RVSDG's properties in other compiler passes.

SLP Vectorization: SLP Vectorization identifies isomorphic scalar operations in straight-line code and packs them into vector operations. This is accomplished by finding seed operations in a basic block and following the data dependence graph from these seed operations to form groups of operations that are vectorizable [157]. In the RVSDG, SLP vectorization could be performed by forming layers of isomorphic nodes in a region. The transformation could be directly performed in the graph as regions are acyclic and all dependencies between nodes are explicitly expressed using edges. Node layering is a well-known problem in graph visualization [16] and inspiration could be drawn from these algorithms.

TLP Extraction: By design, the RVSDG is a concurrent IR as it enables the explicit representation of concurrent computations. A region's structural nodes and function calls can be executed concurrently as long as they do not share any dependencies. TLP is therefore implicitly ingrained in the RVSDG and automatically exposed by analyses and graph transformations, such as invariant edge redirection or alias analysis, as they encode their results as edges in the graph. This TLP parallelism could be explicitly supported by dedicated RVSDG parallel control constructs similarly to the Tapir compiler IR [181], enabling the compiler to optimize across these parallel control constructs.

Other Challenges

Aside from the challenges in code parallelization, the RVSDG offers other research opportunities that arise from its properties. The remainder of this section elaborates on four of these research opportunities.

Register Allocation: Register allocation assigns a large number of program variables to a small number of registers based on the live ranges of variables. In the RVSDG, variable live ranges are not yet fixed, as edges specify only a partial ordering on the nodes within a region. This permits to combine code motion with register allocation [97]. The nodes of a region can be organized into layers such that nodes within a layer maximize the number of used registers, but never exceed them. This would satisfy the register constraints of instructions within each layer, and the next phase then would try to satisfy the constraints globally by performing graph coloring and inserting spills. The effective interleaving of register allocation and code motion mitigates a well-known phase order problem in compilers. Moreover, as the RVSDG explicitly represents loops, the register allocator could be made loop-aware and try to avoid spills in the innermost loops by performing better allocations for these regions.

Predicate Continuation Form: Article B2 introduces predicate continuation form (PCF) as an RVSDG normal form. Throughout destruction, PCF enables the extraction of complex intra-procedural control flow even though the RVSDG supports only two control flow constructs. The article demonstrates the perfect reconstruction of control flow by using predicate control flow recovery (PCFR) to extract a function's original intra-procedural control flow directly after RVSDG construction. In this case, PCFR is applicable as construction generates RVSDGs in PCF and no optimizations are performed between construction and destruction. However, optimizations may restructure the graph such that PCF is lost, impeding the application of PCFR. Article B2 only hints at a method to convert any RVSDG into PCF, but provides no formal algorithm. Such an algorithm needs to be devised before PCFR can be incorporated into any compiler that uses the RVSDG.

Sequentialization: In contrast to the CFG, the RVSDG enforces only necessary dependencies between computations, resulting in a partial execution order.

Even after instruction selection and register allocation, which restrict execution order further by introducing more state edges, computations are not in a total execution order required for code generation. Sequentialization is the process of generating such a total execution order from a given partial execution order. The question that arises is whether there exist differences between total execution orders for different processors (out-of-order vs. in-order) in terms of performance, or whether the results are the same or very similar for any topological node order? Moreover, if there are differences between execution orders, it begs the question whether there is an underlying pattern that could potentially be exploited by a heuristic?

Abstraction Level Encoding: The RVSDG's general nature permits it to encode operations of various abstraction levels. For example, Reissmann *et al.* [164] use the RVSDG to encode GHC's core language, a high-level IR for Haskell, whereas article B1 uses it to encode LLVM IR, a low-level IR for C-based languages. This flexibility permits the RVSDG to hopefully be used throughout the entire compilation process; preferably starting from the abstract syntax tree to retain as much source language information as possible all the way to the final assembly code. The compiler back-end could then be implemented as successive RVSDG transformations, lowering the RVSDG by slowly introducing more and more architecture details.

5.1.2. GPU Divergence

In May 2017, NVIDIA announced the first GPUs with its new microarchitecture, codenamed Volta, that introduces a new SIMT warp execution model with independent thread scheduling [219]. This new model enables the concurrent execution of any thread, regardless of warp, by maintaining the execution state per thread instead of per warp [142]. Figure 5.2 illustrates this change. The predecessors of Volta maintain scheduling resources, such as program counter and call stack, per warp and therefore divergent threads loose concurrency until they all reconverge. In contrast, the Volta SIMT model maintains the execution state per thread and therefore enables the reconvergence of threads at sub-warp granularity. It is the convergence optimizer's responsibility to detect active threads from the same warp and group them to maximize parallel efficiency. This SIMT execution model change permits Volta

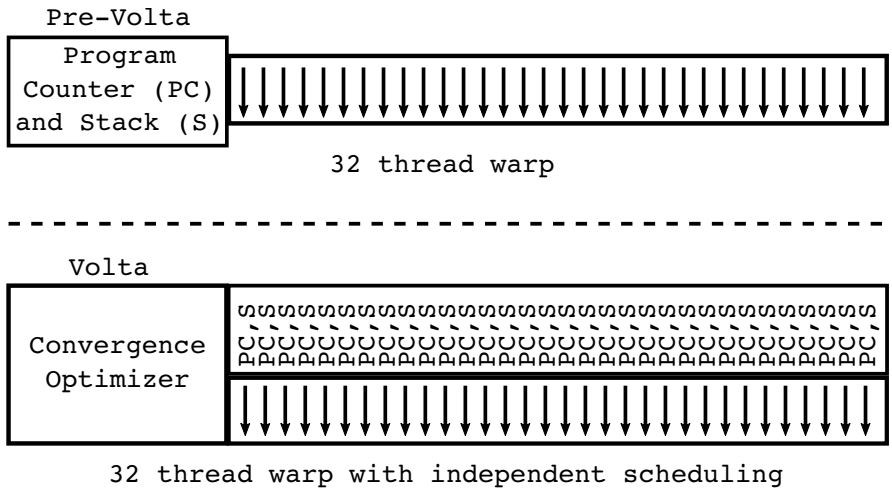


Figure 5.2.: Pre-Volta and Volta thread scheduling architecture. Figure taken from [142].

GPUs to avoid the redundant execution of instructions more efficiently at a fine-grained level in hardware.

5.1.3. Grain Graphs

Aside from the work that introduced grain graphs [131], research in this area culminated in work that

1. examines the suitability of the OpenMP Tools API (OMPT) [65] as an alternative for generating the necessary profile information [106].
2. proposes extensions to OMPT to fully support the generation of grain graphs [107].
3. presents an aggregation method to simplify the visual analysis of large grain graphs [168, 169].

However, all this work is based on a collection of scripts and proof-of-concept implementations as well as third party graph viewers, such as yEd [232], which

were not deliberately developed for OpenMP performance analysis. This complicates further research efforts and the practical usefulness of grain graphs. Thus, a dedicated graph viewer deliberately developed for OpenMP performance analysis using grain graphs is needed. Such a viewer would aid future research into extending grain graphs for OpenMP 4+ and into classifying and ranking problems according to their impact on performance. Moreover, the viewer might be an enabler for the research community in OpenMP performance analysis and be commercially viable.

5.2. Outlook

This section attempts a brief outlook into the near future by elaborating on two challenges: the limits of parallelism inherent in programs and the power limitations of modern processors. These two challenges will be the driving force for future research in the field, and it is vital to address them head-on to enable future increases in system performance.

5.2.1. Parallelism Limits

The end of Dennard scaling also brought an end to technology scaling as a means to improve performance. Chip designers turned to multi-core architectures for exploiting data- and task-level parallelism to further increase performance. While these architectures improve performance for applications that contain such parallelism abundantly, they provide no benefit for sequential applications. Ultimately, parallel performance is governed by Amdahl's law [7], which asserts the theoretical limits of achievable speedup and scalability for applications with insufficient parallelism.

$$S(f,n) = \frac{1}{(1-f) + \frac{f}{n}} \tag{5.1}$$

Amdahl's law is shown in equation 5.1, where S is the achievable speedup, n the number of processors, f the fraction of a program that is parallelizable (ignoring scheduling overhead), and consequently $1-f$ a program's sequential

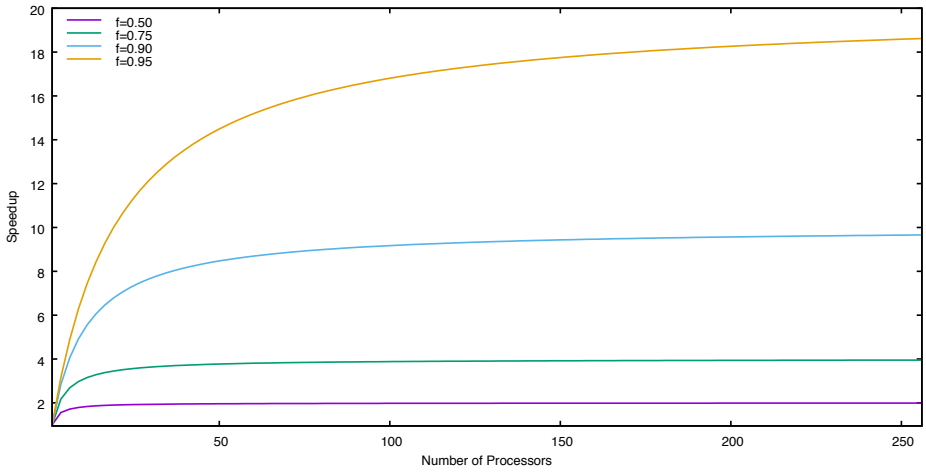


Figure 5.3.: Speedup for various fractions f of Amdahl's law.

part. Figure 5.3 shows the achievable speedup S for different fractions f, illustrating the scalability for applications with various degrees of parallelizable parts. The figure clearly shows that the scalability of an application is dominated by its sequential part. Even with abundant parallelism, *e.g.*, $f = 0.95$, the theoretical achievable speedup with 256 processors is only 18.6 with ever more diminishing returns for an increasing number of processors. Amdahl's law assumes a fixed problem size, and according to Gustafson's law [76], it is possible to mitigate the dominance of the sequential part by increasing problem size. Instead of allocating an increasing number of processors to solve the same problem in shorter time, problem size can be increased such that a larger problem can be solved in the same amount of time. Gustafson's law assumes a sufficient parallelizable part and exploits the scaling of this part with increased problem size. However, even Gustafson's law does not help for applications that consist of a significant sequential part and can effectively not be parallelized. For example, a very low degree of parallelism can be observed in consumer applications, where the number of cores that can be profitably used is around two to three [22]. For such applications, the continued exploitation of ILP seems to be the only foward. This can either be accomplished by avoiding the complexity, power, and memory wall, or by exploiting the ILP from multiple flow of controls.

A promising direction to exploit ILP without running into the different walls is to avoid the dynamic detection of information that is already available at compile time. Hardware structures could be simplified by extending the processor's hardware/software interface to convey more information. For example, it might be possible to reduce the resource intensive and power hungry memory disambiguation structures by extending the instruction set architecture to encode the (in-)dependence of memory operations. Another possibility might be the introduction of instruction slices to enable local forwarding within the execution pipeline instead of utilizing the global and power intensive register file.

Another direction is the ILP exploitation from multiple flows of control, as noted by Lam *et al.* [105] and Mak *et al.* [123]. These studies affirmed ILP limitations from a single flow of control, but also recognized that the exploitable ILP can be increased by an order of magnitude if multiple flows of control could be leveraged. This, however, would require to transform programs at higher abstraction levels to permit the parallelization of independent coarse-grained structures, such as loops or functions. The RVSDG would be a natural candidate IR for such a task, as it explicitly encodes loops and functions, but also permits the encoding of various abstraction levels, as mentioned in Section 5.1.1. It would simplify the exposure of independent flows of control from these abstraction levels, and permit the subsequent exploitation by explicitly encoding their independence in the graph.

Both directions exemplify the importance of compilers for future performance gains, and regardless of the chosen direction, future systems will increasingly rely on software solutions and stronger compilers to uncover the required information for higher performance.

5.2.2. Power Limits

In addition, modern systems are increasingly power constraint, which limits the usable on-chip resources that can be active at any time. The result is *dark silicon* [69], *i.e.*, an ever increasing chip area that cannot be used at any time, and the *utilization wall*, *i.e.*, an underutilization of chip resources due to power constraints. A consequence of the utilization wall is that even

if applications have abundant parallelism, it cannot be exploited without exceeding the given power budget. Dark silicon and the utilization wall are a manifestation of performing computations inefficiently, and their emergence elevated energy-efficiency to an essential requirement for the design of future systems.

A promising direction to improve energy efficiency is specialization, *i.e.*, the design of hardware for specific application domains. This approach trades generality for performance by sacrificing overall performance across application domains for improved performance in specific domains. This increases energy-efficiency for the accelerated domains as more computations can be performed within the given power budget.

One example of specialization are GPUs. They devote more compute resources to accelerate data-parallel applications by sacrificing resources that improve sequential program performance, rendering them more energy-efficient for data-parallel application domains. Another example are accelerators. These architectures are designed for a specific problem (domain) and therefore perform an even bigger tradeoff in terms of generality vs. performance/energy-efficiency as GPUs. These two examples illustrate that specialization can happen at various degrees. While GPUs can still be used for general compute workloads, accelerators are tailored and limited towards the specific workloads they were designed for.

The trend of specializing architectures towards specific application domains will continue as long as power budgets are limited and the demand for performance increases. The future will bear ever more specialized hardware and the programmability gap will widen further. The need for improved energy-efficiency and high developer productivity will challenge established abstractions, assumptions, and methods, and will force us to rethink of how we optimize programs and design the underlying systems. Going forward, this will potentially result in a more holistic approach of combined hardware and software design, where optimizations are performed over the complete system stack.

Part B.

Regionalized Value State Dependence Graph

B1. RVSDG: An Intermediate Representation for Optimizing Compilers

Nico Reissmann, Jan Christian Meyer, and Magnus Själander

Unpublished Manuscript

Abstract. Intermediate Representations (IRs) are a crucial part of every modern compiler as they provide the abstractions to implement optimizations and analyses. A good IR exposes structures and enforces invariants that are important to these optimizations in order to facilitate their implementation. Data centric IRs have emerged as promising candidates for the implementation of optimizations and analyses, but have not been evaluated for feasibility and usability in a compiler as previous work provided no practical implementations.

We present the Regionalized Value State Dependence Graph (RVSDG) IR for optimizing compilers. The RVSDG is a data flow centric IR where nodes represent computations, edges represent computational dependencies, and regions capture the hierarchical structure of programs. It represents programs in demand-dependence form, implicitly supports structured control flow, and models entire programs within a single IR. We provide a complete specification of the RVSDG, construction and destruction methods, as well as exemplify its utility by presenting Dead Node and Common Node Elimination optimizations. We implemented a prototype compiler and evaluate it in terms of performance, code size, compilation time and representational overhead, as well as compare it to a classical CFG-based compiler. Our results indicate that the RVSDG can serve as a competitive IR in optimizing compilers while reducing complexity.

B1.1. Introduction

Intermediate representations (IRs) are at the heart of every modern compiler. These data structures represent programs throughout compilation, connect individual compiler stages, and provide abstractions to facilitate the implementation of analyses, optimizations, and program transformations. A good IR highlights and exposes program properties that are important to the transformations in a specific compiler stage. This reduces the complexity of optimizations and simplifies their implementation.

Data flow centric IRs, such as the Value (State) Dependence Graph (V(S)DG) [224, 98, 111], have emerged as a promising class of IRs for optimizing compilers. These IRs are based on the observation that many optimizations require data flow rather than control flow information, and shift the focus to explicitly expose data instead of control flow. They represent programs in demand-dependence form, encode structured control flow, and explicitly model data flow between operations. This raises the IR's abstraction level, permits simple and powerful implementations of data flow optimizations, and helps to expose the parallelism inherent in programs [111, 98, 197].

This shift in focus from explicit control flow to only structured and implicit control flow requires more sophisticated construction and destruction methods [224, 111, 196]. Specifically, the effective benefits of improved analyses and optimizations require V(S)DG destruction to produce efficient intra-procedural control flow. Bahmann *et al.* [13] describe the first method that permits the complete recovery of the original control flow from a procedure represented in a demand-dependence graph. The paper presents the *Regionalized Value State Dependence Graph* (RVSDG) and shows that the RVSDG's restricted control flow constructs do not limit the complexity of the recoverable control flow.

However, their focus lay on the intra-procedural RVSDG constructs as well as the restructuring and recovery of control flow. They omitted inter-procedural constructs, their construction and destruction, as well as the handling of intra-procedural dependencies during construction. They also lacked a RVSDG implementation that would permit them to quantitatively and qualitatively evaluate the IR. This paper complements the missing RVSDG constructs, its complete construction and destruction, and in addition demonstrates the RVSDG's

feasibility and practicality as an IR for optimizations. Specifically, we make the following contributions:

1. A complete RVSDG specification, including intra- and inter-procedural constructs.

2. A complete description of RVSDG construction and destruction, augmenting the previously proposed algorithms with the construction and destruction of inter-procedural constructs, as well as the handling of intra-procedural dependencies during construction.

3. A presentation of Dead Node Elimination (DNE) and Common Node Elimination (CNE) optimizations to demonstrate the RVSDG's utility. DNE combines dead and unreachable code elimination, as well as dead function removal. CNE permits the removal of redundant computations by detecting congruent operations.

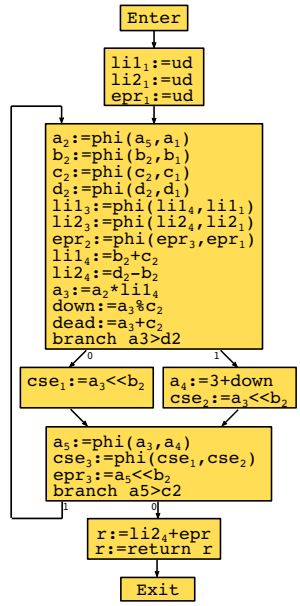
4. A publicly available [165] prototype compiler that implements the discussed concepts. It consumes and produces LLVM IR, and is to our knowledge the first optimizing compiler that uses a demand dependence graph as IR.

5. An evaluation of the RVSDG in terms of performance and size of the produced code, as well as compile time and representational overhead. We further provide a comparison to LLVM, demonstrating that the properties and invariants of the RVSDG raise the abstraction level of the IR and reduce implementation cost.

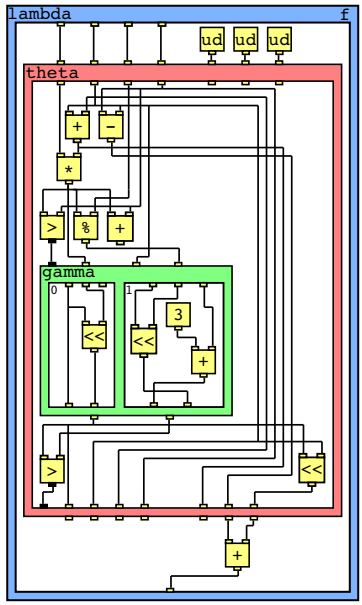
Our results show that the RVSDG has no inherent impediment that prevents it from producing competitive code, suggesting that it can serve as the IR in a compiler's optimization stage. This paves the way for more research into the RVSDG and further exploration of its properties and their effect on optimizations and analyses, as well as its usability for dataflow and parallel architectures.

```
int
f(int a, int b, int c, int d)
{
  int li1, li2;
  int cse, epr;
  do {
    li1 = b+c;
    li2 = d-b;
    a = a*li1;
    int down = a%c;
    int dead = a+d;
    if(a > d) {
      int acopy = a;
      a = 3+down;
      cse = acopy<<b;
    } else {
      cse = a<<b;
    }
    epr = a<<b;
  } while(a > cse);
  return li2+epr;
}
```

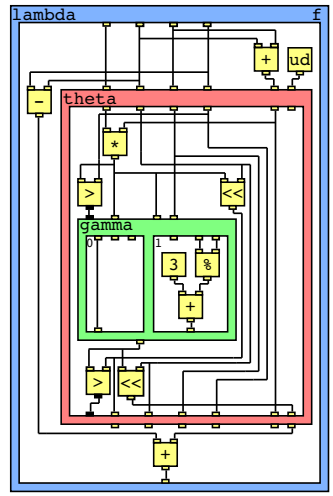
(a) Code



(b) CFG in SSA form



(c) Unoptimized RVSDG

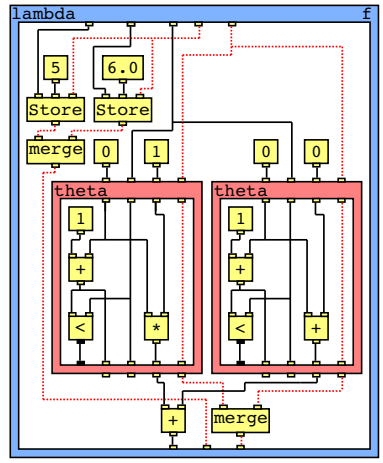


(d) Optimized RVSDG

Figure B1.1.: Simplifying conventional compilation

```
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 of Code B1.2a

Figure B1.2.: Exposing concurrent computations

B1.2. Motivation

Figure B1.1a shows a function with a simple loop and a conditional, and Figure B1.1b shows the corresponding Control Flow Graph (CFG) in Static Single Assignment (SSA) form [52]. The CFG in SSA form is the predominant IR for optimizations in modern imperative language compilers [199]. 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 [172, 223].

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. For example, six of the ten most invoked passes in LLVM are helper passes only performing such tasks. They amount to 23% of all pass invocations (see Section B1.7.5). The lack of enforced invariants complicates the implementation of optimizations and analyses, increases engineering effort, unnecessarily prolongs compilation time,

and leads to compiler bugs [116, 117, 118].

In contrast, the RVSDG exposes program structure and enforces invariants beneficial for optimizations and analyses. Figure B1.1c shows the RVSDG corresponding to Figure B1.1a. It is an acyclic demand-dependence graph where nodes represent simple operations or control flow constructs, and edges the dependencies between computations (see Section B1.3.1). In Figure B1.1c, 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, simplifying the implementation of transformations [224, 98, 111]. 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 [43]. The representation of intra- and inter-procedural control flow as nodes permits to encode an entire translation unit in a single representation, and avoids additional data structures, such as the call graphs, or passes, such as loop detection and normalization.

RVSDG properties (see Section B1.3.2) combined with its explicit data flow enable simple and powerful optimizations (see Section B1.6). Figure B1.1d shows the optimized RVSDG of Figure B1.1c, 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`, 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 B1.1c. No additional data structures or helper passes are required.

Moreover, the RVSDG's explicit representation of program states enables it to encode the relations of side-effecting operations in the graph. The code in Figure B1.2a is used to illustrate this concept. 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. In contrast, the RVSDG permits the encoding of such information directly in the graph, as shown in Figure B1.2b. 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, since they are represented explicitly as nodes and share no dependencies.

In summary, the RVSDG raises the IR abstraction level by enforcing desirable properties, such as SSA form, explicitly encoding important structures, such as loops, and relaxing 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, such as the CFG, and further helps to expose parallelism in programs.

B1.3. The Regionalized Value State Dependence Graph

This section introduces the RVSDG. Specifically, Section B1.3.1 defines the RVSDG and provides further examples for its usage, while Section B1.3.2 details its properties.

B1.3.1. Definition

A Regionalized Value State Dependence Graph (RVSDG) is an acyclic hierarchical multigraph consisting of nested regions. A region $\mathscr{R} = (A, N, E, R)$ represents a computation with argument tuple A, nodes N, edges E, and result tuple R. A node can be either *simple, i.e.,*, it represents a primitive operation, or *structural, i.e.,*, it contains regions. Each node $n \in N$ has a tuple of inputs I and outputs O. In case of simple nodes, they correspond to arguments and results of the represented operation, whereas for structural nodes, they map to arguments and results of the contained regions. For nodes $n_1, n_2 \in N$, an edge $(g, u) \in E$ connects either output $g \in O_{n_1}$ or argument $g \in A$ to either input $u \in I_{n_2}$ or result $u \in R$ of matching type. We refer to g as the *origin* of an

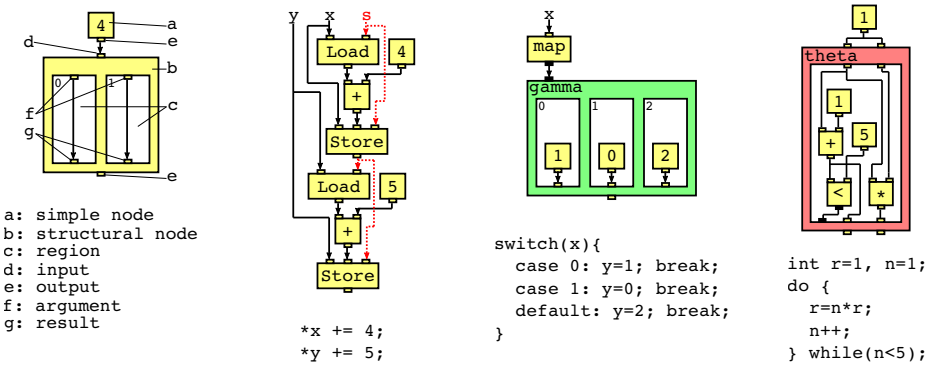
edge, and to u as the *user* of an edge. Every input or result is the user of *exactly one* edge, whereas outputs or arguments can be the origins of multiple edges. All inputs or results of an origin are called its *users*. The corresponding node of an origin is called its *producer*, whereas the corresponding node of a user is called *consumer*. Correspondingly, the set of nodes of all users of an origin are referred to as its *consumers*. The types of inputs and outputs are either *values*, representing arguments or results of computations, or *states*, used to impose an order on operations with side-effects. A node's *signature* are the types of its inputs and outputs, whereas a region's signature are the types of its arguments and results. Figure B1.3a depicts the introduced notation.

Throughout this paper, we use n, e, i, o, a, and r with sub- and superscripts to denote individual **n**odes, **e**dges, **i**nputs, **o**utputs, **a**rguments, and **r**esults, respectively. We use u and g to denote an edge's **u**ser and ori**g**in, respectively. An edge e from user u to origin g is also denoted as $e : (u, g)$, or short (u, g).

Nodes

Simple nodes model primitive operations such as addition, subtraction, load, and store. They have an operator associated with them, and a node's signature must correspond to the signature of its operator. Simple nodes map their input value tuple to their output value tuple by evaluating their operator with the inputs as arguments, and associating the results with their outputs. Figure B1.3b illustrates the use of simple nodes as well as value and state edges. Solid lines represent value edges, whereas dashed lines represent state edges. Nodes have as many value inputs and outputs as their corresponding operations demand. The ordering of the store nodes is preserved by sequentializing them with the help of a state edge.

Structural nodes contain regions and can model structural program behavior such as the conditional or repeated evaluation of computations. We present five different kind of structural nodes: γ-nodes, which represent conditionals, θ-nodes, which represent tail-controlled loops, λ-nodes for procedures and functions, ϕ-nodes for mutually recursive function environments, and ω-nodes for translation units. The rest of this section discusses each structural node in detail and illustrates their usage.



(a) Notation (b) Simple nodes (c) γ-node (d) θ-node

Figure B1.3.: Notation as well as examples for the usage of simple, γ- and θ-nodes.

Gamma-Nodes: A γ-node models a decision point and contains regions $\mathscr{R}_0, ..., \mathscr{R}_k \mid k > 0$ of matching signature. Its first input is a *predicate*, which determines the region under evaluation. It evaluates to an integer v with $0 \leq v \leq k$. The values of all other inputs are mapped to the corresponding arguments of region $\mathscr{R}_v$, $\mathscr{R}_v$ is evaluated, and the values of its results are mapped to the outputs of the γ-node.

γ-nodes represent conditionals with symmetric control flow splits and joins, such as `if-then-else` or `switch` statements without fall-throughs. Figure B1.3c shows a γ-node. It contains three regions: one for each case, and a default region. The map node takes the value of x as input and maps it to zero, one, or two, determining the region under evaluation. This region is evaluated and its result is mapped to the γ-node's output.

We define the *entry variable* of a γ-node as a pair of an input and the arguments the input maps to during evaluation, as well as the *exit variable* of a γ-node as a pair of an output and the results the output could receive its value from:

Definition 1. *The pair* $ev_l = (i_l, A_{l-1})$ *is the* l*-th* entry variable *of a* γ*-node with* k *regions. It consists of the* l*-th input and tuple* $A_{l-1} = \{a_{l-1}^{\mathscr{R}_0}, ..., a_{l-1}^{\mathscr{R}_k}\}$ *with the*

$l-1$-th argument from each region. We refer to the set of all entry variables as EV.

Definition 2. *The pair $ex_l = (R_l, o_l)$ is the l-th* exit variable *of a γ-node with k regions. It consists of a tuple $R_l = \{r_l^{\mathscr{R}_0}, ..., r_l^{\mathscr{R}_k}\}$ of the l-th result from each region and the l-th output they would map to. We refer to the set of all exit variables as EX.*

Theta-Nodes: A θ-node models a tail-controlled loop. It contains one region that represents the loop body. The length and signature of its input tuple equals that of its output, or the region's argument tuple. The first region result is a *predicate*. Its value determines the continuation of the loop. When a θ-node is evaluated, the values of all its inputs are mapped to the corresponding region arguments and the body is evaluated. When the predicate is true, all other results are mapped to the corresponding arguments for the next iteration. Otherwise, the values of the results are mapped to the corresponding outputs. The loop body of an iteration is always fully evaluated before the evaluation of the next iteration starts. This avoids problems such as "deadlocks" between computations of the loop body and the predicate. Moreover, it results in well-defined behavior for non-terminating loops that update external state.

θ-nodes permit the representation of `do-while` loops. In combination with γ-nodes, it is possible to model head-controlled loops, *i.e.*,, `for` and `while` loops. Thus, employing tail-controlled loops as basic loop construct enables us to express more complex loops as a combination of basic constructs. This normalizes the representation and reduces the complexity of optimizations as there exists only one construct for loops. Another benefit of tail-controlled loops is that their body is guaranteed to execute at least once, enabling the unconditional hoisting of invariant code with side-effects.

Figure B1.3d shows a θ-node with two loop variables, n and r, and an additional result representing the predicate. When the predicate evaluates to true, the results for n and r of the current iteration are mapped to the region arguments to continue with the next iteration. When the predicate evaluates to false, the loop exits and the results are mapped to the node's outputs. We define a *loop variable* as a quadruple that represents a value routed through a θ-node:

Definition 3. *The quadruple $lv_l = (i_l, a_l, r_{l+1}, o_l)$ is the l-th* loop variable *of a θ-node. It consists of the l-th input i_l, argument a_l, and output o_l, as well as the $l+1$-th result of a θ-node. We refer to the set of all loop variables as LV.*

Lambda-Nodes: A λ-node models a function and contains a single region representing a function's body. It features a tuple of inputs and a single output. The inputs refer to external variables the λ-node depends on, and the output represents the λ-node itself. The region has a tuple of arguments comprised of a function's external dependencies and its arguments, and a tuple of results corresponding to a function's results.

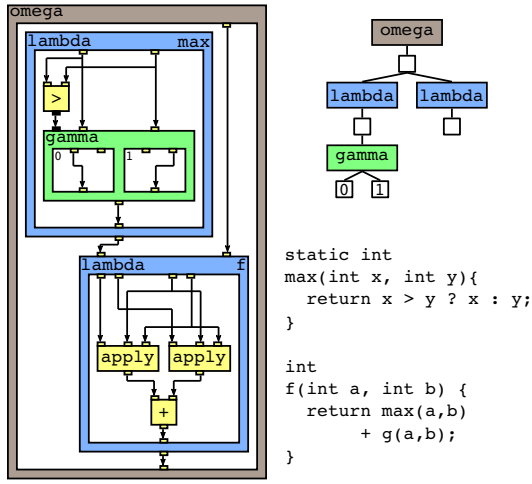
An $apply$-node represents a function invocation. Its first input takes a λ-node's output as origin, and all other inputs represent the function arguments. In the rest of the paper, we refer to an $apply$-node's first input as its *function input*, and to all its other inputs as its *argument inputs*. Invocation maps the values of a λ-node's input k-tuple to the first k arguments of the λ-region, and the values of the function arguments of the $apply$-node to the rest of the arguments of the λ-region. The function body is evaluated and the values of the λ-region's results are mapped to the outputs of the $apply$-node.

Figure B1.4a shows an RVSDG with two λ-nodes. Function f calls functions g and max with the help of $apply$-nodes. The function max is part of the translation unit, while g is external and must be imported (see the paragraph about ω-nodes for more details). We further define the *context variables* of a λ-node. A context variable provides the corresponding input and argument for a variable a λ-node depends on.

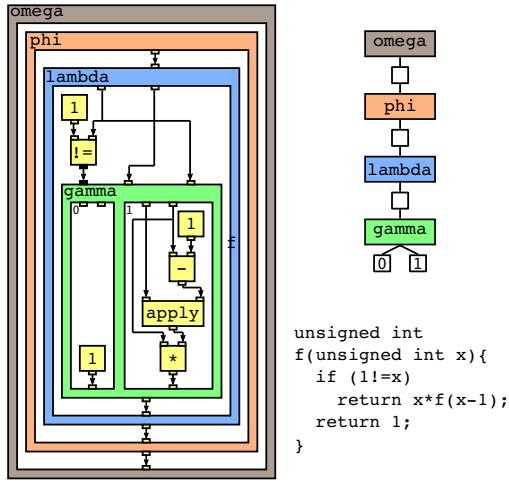
Definition 4. *The pair $cv_l = (i_l, a_l)$ is a λ-node's l-th* context variable. *It consists of the l-th input and argument. We refer to the set of all context variables as CV.*

Definition 5. *The λ-node connected to a function input is the* callee *of an $apply$-node, and an $apply$-node is the* caller *of a λ-node. We refer to the set of all callers of a λ-node as CLL.*

Phi-Nodes: A ϕ-node models an environment with mutually recursive functions, and contains a single region with λ-nodes. Each single output of these λ-nodes serves as origin to a single result in the ϕ-region. A ϕ-node's outputs expose the individual functions to callers outside the ϕ-region, and must



(a) RVSDG with λ-nodes



(b) RVSDG with a ϕ-node

Figure B1.4.: Example for the usage of λ- and ϕ-nodes, as well as corresponding region trees.

therefore have the same arity and signature as the results of the ϕ-region. The first input of an $apply$-node from *outside* the ϕ-region takes these outputs as origin to invoke one of the functions.

The inputs of a ϕ-node refer to variables that the contained functions depend on and are mapped to corresponding arguments in the ϕ-region when a function is invoked. In addition, a ϕ-region has arguments for each contained function. An $apply$-node from inside a ϕ-region takes these as origin to its function input.

ϕ-nodes permit a program's mutually recursive functions to be expressed in the RVSDG without the introduction of cycles. Figure B1.4b shows an RVSDG with a ϕ-node. The function f calls itself, and therefore needs to be in a ϕ-node to preserve the RVSDG's acyclicity. The region in the ϕ-node has one input, representing the declaration of f, and one output, representing the definition of f. The ϕ-node has one output so that f can be called from outside the recursive environment.

We define *context variables* and *recursion variables*. Context variables provide corresponding inputs and arguments for variables the λ-nodes from within a ϕ-region depend on. Recursion variables provide the argument and output an $apply$-node's function input connects to.

Definition 6. *The pair $cv_l = (i_l, a_l)$ is the l-th* context variable *of a ϕ-node. It consists of the l-th input and argument. We call the set of all context variables CV.*

Definition 7. *For a ϕ-node with n context variables, the triple $rv_l = (r_l, a_{l+n}, o_l)$ is the l-th* recursion variable. *It consists of the l-th result and $l+n$-th argument of the ϕ-region as well as the l-th output of the ϕ-node. We refer to the set of all recursion variables as RV.*

Omega-Nodes: An ω-node models a translation unit. It is the top-level node of an RVSDG and has no inputs or outputs. It contains exactly one region. This region's arguments represent entities that are external to the translation unit and therefore need to be imported. Its results mark all exported entities in the translation unit. Figure B1.4a and B1.4b illustrate the usage of ω-nodes. The ω-region in Figure B1.4a has one argument, representing the import of function `g`, and one result, representing the export of function `f`. The ω-region in Figure B1.4b has only one export for function `f`.

Edges

Edges connect node outputs or region arguments to a node input or region result, and are either value typed, *i.e.*,, represent the flow of data between computations, or state typed, *i.e.*,, impose an ordering on operations with side-effects. State edges are used to preserve the observational semantics of the input program by ordering its side-effecting operations. Such operations include memory writes, loads that may cause segmentation faults, as well as exceptions.

In practice, a richer type system permits further distinction between different kind of values or states. For example, different types for fixed-point and floating-point values helps to distinguish between these arithmetics, and an additional type for functions permits to correctly specify the output types of λ-nodes as well as the function input of $apply$-nodes.

B1.3.2. RVSDG Properties

This section summarizes inherent RVSDG properties that offer advantages for optimization implementations in compilers.

Program Coverage: Structural nodes permit the representation of an entire translation unit by modeling control flow constructs, such as conditionals, loops, and functions. The RVSDG therefore provides a unified framework for optimizations, avoiding the need for multiple IRs to model different aspects of a program. All inter- and intra-procedural dependencies between operations are explicit, enabling the unification of traditionally distinct inter- and intra-procedural optimizations as shown in section B1.6.

Flexibility: The RVSDG can model constructs at different abstraction levels. subsection B1.3.1 presents architecture independent constructs, suitable to represent higher level constructs, such as Haskell's Core IR [164]. It is also possible to represent machine instructions as simple nodes and architecture specific subroutines as structural nodes, where inputs and outputs are annotated with specific registers.

Normalization: The RVSDG models all source language constructs as nodes. Simple nodes represent expressions, γ-nodes represent conditionals, θ-nodes

represent loops, and λ-nodes represent functions and procedures. Reducing different source language constructs to a single IR construct restricts the number of cases an optimizing compiler must handle, and thus simplifies its implementation [98]. Moreover, RVSDG encoding adds no sequencing to functionally pure expressions, while operations with side-effects are sequentialized to preserve program semantics. This renders nodes referentially transparent and explicitly exposes parallel operations, as exemplified in Section B1.2.

SSA: The RVSDG is in strict SSA form [52] by definition, as edges connect each input to only one output. This ensures that each variable is assigned exactly once, and defined before its usage. Strict SSA form is automatically created through RVSDG construction and in contrast to other IRs (see Section B1.7.5), transformations, such as jump threading or live-range splitting, cannot destroy this form in the RVSDG. This renders SSA restoration passes [43] irrelevant.

Structure: The RVSDG explicitly exposes program structure as γ-, θ-, λ-, and ϕ-nodes. These nodes can contain regions with more structural nodes, forming a *region tree* that represents an entire translation unit's structure. The upper right of Figure B1.4a and Figure B1.4b show the region tree encoded in the corresponding RVSDGs. The region tree is similar to the Program Structure Tree (PST) [99]. This explicit representation of loops and their relationship eliminates the need to identify and normalize them for optimizations as in other IRs [3].

Ease of Use: An IR must be efficiently traversable and manipulable to enable efficient implementations of optimizations. The encoded region tree and the RVSDG's acyclicity enable simple traversals of entire programs by recursively visiting subregions of structural nodes and traversing simple nodes in topological order. If only structural nodes are of importance, the procedure reduces to a tree traversal. Transformations are expressed by traversing the graph, marking nodes and edges, graph rewriting, copying nodes between regions, and redirecting edges. section B1.6 presents two powerful optimizations based on these simple graph transformations.

B1.4. Construction

RVSDG construction is responsible for mapping all constructs, concepts, and abstractions of an input language to the RVSDG. The mapping is language-specific and depends on the language's concrete features. For example, languages with possibly unstructured control flow, such as C or C++, cannot be mapped directly to the RVSDG and require the CFG as a stepping stone, while other languages, such as Haskell, permit a direct construction [164]. In order to stay language independent, we present RVSDG construction with an Inter-Procedure Graph and CFG as input.

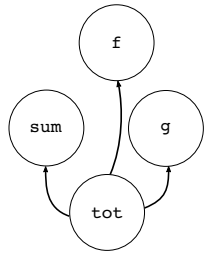
We split RVSDG construction in two phases: *Inter-Procedural Translation* (Inter-PT) and *Intra-Procedural Translation* (Intra-PT). Inter-PT translates functions and inter-procedural dependencies, creating λ- and ϕ-nodes, while Intra-PT translates intra-procedural control and data flow, generating a function's body. Both use a common symbol table to map functions and CFG variables to the corresponding arguments or outputs in the RVSDG. The creation of arguments or outputs triggers updates to this table, but we omit them in the algorithm descriptions to avoid unnecessary cluttering.

B1.4.1. Inter-Procedural Translation

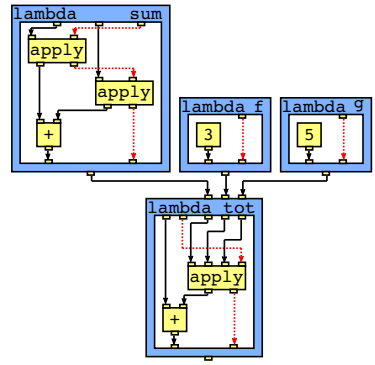
Inter-PT converts all functions from the *Inter-Procedure Graph* (IPG) of a translation unit to λ-nodes. The IPG is an extension of a call graph and captures all static dependencies between functions, incorporating not only those originating from (direct) calls, but also those from other references within a function. In the IPG, an edge from node $n1$ to node $n2$ exists, if the body of the function corresponding to $n1$ references the function represented by $n2$. Figure B1.5b shows the IPG for the code in Figure B1.5a. The code consists of four functions, with function `sum` performing two indirect calls. The corresponding IPG consists of four nodes and three edges. All edges originate from node tot, as it is the only function that explicitly references other functions, *i.e.*, `sum` for a direct call, and `f` and `g` to pass as argument. No edge originates from node `sum`, as the corresponding function does not explicitly reference any other functions, and the functions for the indirect calls are provided as arguments.

```
int
f(){ return 3; }

int
g(){ return 5; }

int
sum(int(*x)(), int(*y)()) {
  return x() + y();
}

int
tot(int z) {
  return z + sum(f, g);
}
```

(a) Code



(b) IPG



(c) RVSDG

Figure B1.5.: Inter-Procedural Translation

The RVSDG puts two constraints on the translation from an IPG. Firstly, mutually recursive functions are required to be created within ϕ-nodes to preserve the RVSDG's acyclicity. Secondly, Inter-PT must respect the calling dependencies of functions to ensure that λ-nodes are created before their $apply$-nodes. In order to embed mutually recursive functions into ϕ-nodes, we need to identify the strongly connected components (SCCs) in the IPG. We consider an SCC *trivial*, if it consists only of a single node with no self-referencing edges. Otherwise, it is *non-trivial*. Moreover, a trivial SCC might not have a CFG associated with it, and is therefore defined in another translation unit.

Algorithm I outlines the RVSDG construction from an IPG. It finds all SCCs and converts trivial SCCs to individual λ-nodes, while the λ-nodes created from non-trivial SCCs are embedded in ϕ-nodes. This satisfies the first constraint. The second constraint is satisfied by processing SCCs in topological order, creating λ-nodes before their $apply$-nodes. The identification and ordering of SCCs can be performed in a single step with Tarjan's algorithm [206], which returns the identified SCCs in reverse topological order. Figure B1.5c shows the RVSDG after the application of Algorithm I to the IPG in Figure B1.5b.

In addition to function arguments, Algorithm I conservatively assumes that all functions are stateful and adds an additional state argument and result to the λ-regions. This state is used to sequentialize operations with side-effects, and nodes representing such operations consume this state and produce another

Algorithm I: INTER-PROCEDURAL TRANSLATION

Compute all SCCs in an IPG and process them in topological order of the directed acyclic graph formed by the SCCs as follows:

1. **TRIVIAL SCC***:*
   a) **WITH CFG***: Begin a λ-node by adding all context variables, function arguments, and an additional state argument to the λ-region. Translate the CFG with Intra-PT as explained in Section B1.4.2, and finish the λ-node by adding the function results and the state result to the λ-region. If a function is exported, add a result to the ω-region and connect the λ-node's output to it.*
   b) **WITHOUT CFG***: Add a ω-region argument for the external function.*
2. **NON-TRIVIAL SCC***: Begin a ϕ-node by adding all functions as well as context variables to the ϕ-region. Translate each function in the SCC according to* TRIVIAL SCC *without exporting them. Finish the ϕ-node by adding all function outputs as results to the ϕ-region. If a function is exported, add a result to the ω-region and connect the ϕ-node's output to it.*

one for the next node. This translates operations with side-effects according to the total order specified by the original program, and ensures correct observable behavior. After construction, alias analysis can be used to relax this conservatively sequentialized order. Pure functions contain no operations that use the added state and would therefore only pass it through, *i.e.*,, the origin of the state result would be the argument of the λ-region. This helps to distinguish pure functions from others and optimize them accordingly.

B1.4.2. Intra-Procedural Translation

This phase translates intra-procedural control and data flow. One of the RVSDG's benefits for optimizations is its limited support of control flow constructs. It only provides γ- and θ-nodes to model structural behavior and therefore requires the restructuring of a function's control flow to make it amenable for the RVSDG. This is the task of control flow restructuring (CFR), which restructures a function's control flow so that it only consists of tail-controlled loops and conditionals with symmetric control flow splits and joins. After CFR, a control tree is constructed from the CFG and its nodes are annotated with the corresponding data dependencies. Finally, the RVSDG is created by traversing the control tree and translating its nodes to RVSDG nodes.

Control Flow Restructuring

CFR converts a CFG so that it only contains tail-controlled loops as well as conditionals with properly nested splits and joins. This phase is only necessary for languages that support a wider range of control flow constructs, such as C or C++, but can be skipped for languages with more limited control flow, such as Haskell or Scheme. CFR consists of two interlocked phases: loop restructuring and branch restructuring. Loop restructuring transforms all loops to tail-controlled loops, while branch restructuring ensures conditionals with symmetric control flow splits and joins. These tail-controlled loops and conditionals are later converted to θ- and γ-nodes, respectively.

We omit an extensive discussion of CFR as it is detailed in Bahmann *et al.* [13]. In contrast to node splitting [237], CFR avoids the possibility of exponential code blowup [37] by facilitating predicate and branch insertions. It does not require a CFG in SSA form as this form is automatically established throughout construction, and its time and space requirements are linear in practice.

Structural Analysis

A restructured CFG can only consist of three different single-entry/single-exit (SESE) control flow regions:

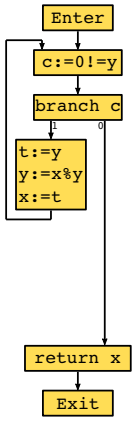
- **Linear Region**: A linear subgraph where the entry node and all intermediate nodes have only one outgoing edge, and the exit node as well as all intermediate nodes have only one incoming edge.
- **Branch Region**: An SESE subgraph with the entry and exit node representing the control flow split and join, respectively, and each alternative consisting either of a single node, or an edge from the entry to the exit node.
- **Loop Region**: A single node where an edge originates and targets this node.

These regions can be identified by performing an interval [130] or structural [183] analysis, resulting in a control tree [130] with basic blocks as leaves and abstract region nodes as branches. A linear node has all nodes of the linear subgraph as children, with the subgraph's entry and exit node as the left and right most child, respectively. A branch node's first child is the subgraph's entry node, and all other children are the nodes of each alternative. An empty alternative, *i.e.*,, an edge from the subgraph's entry to exit node, is represented as an empty basic block. A loop node has only the node representing the loop's body as single child. Figure B1.6a shows Euclid's algorithm as a CFG, and Figure B1.6b shows the same CFG after CFR, which restructured the head-controlled loop to a tail-controlled loop. The left of Figure B1.6c shows the corresponding control tree for the restructured CFG.

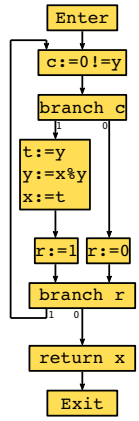
Symbolic Translation

The discovery of individual control flow regions in a CFG allows for a direct translation to an RVSDG. The control tree nodes for branches and loops are directly mapped to γ- and θ-nodes, and the children of linear nodes are translated individually. However, in order to create these nodes, their data dependencies are required for the creation of inputs and outputs. This can be achieved by traversing the nodes of the control tree and annotating them with the set of variables that are demanded. For branch nodes, we have to keep track of two sets: entry and exit set, whereas for loop nodes we only keep track of one set, since a θ-node's input signature matches its output signature.

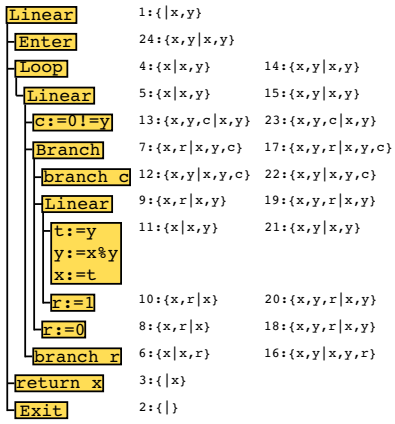
Algorithm II uses demand set D to keep track of variables while traversing the control tree. It adds and removes variables throughout traversal and sets the corresponding demand sets for all nodes. For branch nodes, the entry set is computed by taking the union of the demand sets from all alternatives in addition to the demand set of the entry node. For loop nodes, a reprocessing of the node's subtree is required, if the node's initial demand set is unequal to the demand set after processing. This is due to the requirement that a θ-node's inputs and outputs must have the same signature. The right of Figure B1.6c shows the traversal of the control tree. The notation $n : \{D_{before} | D_{after}\}$ summarizes the visit of each node with n denoting the order of visit, D_{before} the variables in D before processing a node's children, and D_{after} the variables in



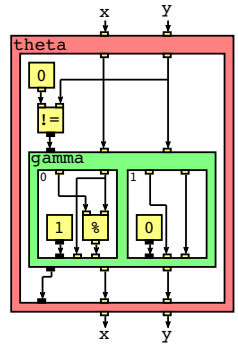
(a) CFG



(b) Restructured CFG



(c) Annotated control tree



(d) RVSDG

Figure B1.6.: Intra-Procedural Translation

Algorithm II: DEMAND ANNOTATION

Process the control tree nodes with the empty demand set D as follows:

- **LINEAR NODE**: *Recursively process the children bottom-up. The node's demand set equals the top most child's demand set.*
- **BRANCH NODE**: *Set D as the node's exit set. Recursively process all children, except the first child, bottom-up with D. Set D to the union of all the children's demand sets, and recursively process the first node with D. The node's entry set equals the first node's demand set.*
- **LOOP NODE**: *Set D as the node's demand set and process its child. If D is unequal to the node's demand set after processing the child, set D as the node's demand set and reprocess the child.*
- **BASIC BLOCK**: *Process the node's operations bottom-up, and remove every variable from D that is defined, as well as add every variable to D that is used. Set D as the node's demand set.*

Algorithm III: SYMBOLIC TRANSLATION

Process the control tree nodes as follows:

- **LINEAR NODE**: *Recursively process the node's children top-down.*
- **BRANCH NODE**: *Recursively process the node's first child. Begin a γ-node with inputs according to the node's entry set. Create subregions by recursively processing the node's other children. Finish the γ-node with outputs according to node's exit set.*
- **LOOP NODE**: *Begin a θ-node with inputs according to its demand set. Create its region by recursively processing its child. Finish the θ-node with outputs according to its demand set.*
- **BASIC BLOCK**: *Process the node's operations top-down and create simple nodes in the RVSDG.*

D after processing a node's children. Thus, in Figure B1.6c, the traversal starts at the root node with D being empty, continues to the exit node, then to the basic block with the `return` statement where x is added to D, and continues further to the loop node, and so forth. After demand annotation, the control tree is translated according to Algorithm III. It processes each node in the control tree creating γ- and θ-nodes for all branch and loop nodes, respectively. Figure B1.6d shows the resulting RVSDG nodes.

B1.5. Destruction

The destruction stage reestablishes control flow by extracting an IPG from an RVSDG as well as generating CFGs from individual λ-regions. Inter-Procedural Control Flow Recovery (Inter-PCFR) creates an IPG from λ-nodes, while Intra-Procedural Control Flow Recovery (Intra-PCFR) extracts control flow from γ- and θ-nodes and generates basic blocks with corresponding operations for primitive nodes. A λ-region without γ- and θ-nodes is trivially transformed into a linear CFG, whereas λ-regions with these nodes require the construction of branches and/or loops. Bahmann *et al.* [13] explored two different approaches for CFG generation: *Structured Control Flow Recovery (SCFR)* and *Predicative Control Flow Recovery (PCFR)*. SCFR uses the region hierarchy within a λ-region to recover control flow, while PCFR generates branches for predicate producers and follows the predicate consumers to the eventual destination. Both schemes reestablish evaluation-equivalent CFGs, but differ in the recoverable control flow. SCFR recovers only control flow that resembles the structural nodes in λ-regions, *i.e.*,, control flow equivalent to `if-then-else`, `switch`, and `do-while` statements, while PCFR can recover arbitrary complex control flow, *i.e.*,, control flow that is not restricted to RVSDG constructs. PCFR reduces the number of static branches in the resulting control flow [13], but might also result in undesirable control flow for certain architectures, such as graphic processing units [167]. For the sake of brevity, we omit a discussion of Intra-PCFR as the algorithms are extensively described by Bahmann *et al.* [13].

B1.5.1. Inter-Procedural Control Flow Recovery

Inter-PCFR recovers an IPG from an RVSDG. IPG nodes are created for λ-nodes as well as arguments of the ω-region, while IPG edges are inserted to capture the dependencies between λ-nodes. Algorithm IV starts by creating IPG nodes for all arguments of the ω-region, *i.e.*,, all external functions. It continues by recursively traversing the region tree, creating IPG nodes for encountered λ-nodes and IPG edges for their dependencies. For the region of every λ-node, it invokes Intra-PCFR to create a CFG using either SCFR or PCFR.

Algorithm IV: INTER-PROCEDURAL CONTROL FLOW RECOVERY

1. *Create IPG nodes for all function arguments of the ω-region.*
2. *Process all nodes of the ω-region in topological order as follows:*
   - **λ-NODES**: *Create an IPG node, and mark it exported if the λ-node's output has a ω-region's result as user. For every context variable $cv = (i, a)$, add an edge from the λ-node's IPG node to the corresponding IPG node of the producer of i. Create a CFG from the λ-node's subregion and attach it to the IPG node.*
   - **ϕ-NODES**: *For every argument of the ϕ-region, create an IPG node for the corresponding λ-node and add IPG edges from this node to the corresponding IPG nodes of the context variables. Create a CFG from every λ-node's subregion and attach it to the IPG node. Mark the IPG node as exported if the corresponding ϕ-node's output has a ω-region's result as user.*

B1.6. Optimizations

The properties of the RVSDG make it appealing for optimizing compilers. Many optimizations can be expressed as simple graph traversals, where subgraphs are rewritten, nodes are moved between regions, nodes or edges are marked, or edges are diverted. In this section, we present Dead and Common Node Elimination optimizations that exploit the RVSDG's properties to unify traditionally distinct transformations.

B1.6.1. Dead Node Elimination

Dead Node Elimination (DNE) is a combination of dead and unreachable code elimination, and removes all nodes that do not contribute to the result of a computation. Dead nodes are generated by unreachable and dead code from the input program, as well as by other optimizations such as Common Node Elimination. An operation is considered dead code when its results are either not used or only by other dead operations. Thus, an output of a node is dead, if it has no users or all its users are dead. We consider a node to be dead, if all its outputs are dead. It follows that a node's inputs are dead, if the node itself is dead. We call all inputs, outputs, or nodes that are not dead alive.

The implementation of DNE consists of two phases: mark and sweep. The mark phase identifies all outputs and arguments that are alive, while the sweep

phase removes all dead entities. The mark phase traverses RVSDG edges according to the rules in Algorithm V. If a structural node is dead, the mark phase skips the traversal of its subregions as well as all of the contained computations, as it never reaches the node in the first place. The mark phase is invoked for all result origins of the ω-region.

The sweep phase performs a simple bottom-up traversal of an RVSDG, recursively processing subregions of structural nodes as long as these nodes are alive. A dead structural node is removed with all its contained computations. The RVSDG's uniform representation of all computations as nodes permits DNE to not only remove simple computations, but also compound computations such as conditionals, loops, or even entire functions. Moreover, its nested structure avoids the processing of entire branches of the region tree if they are dead.

Figure B1.7d shows the RVSDG from Figure B1.7c after the mark phase. Grey colored inputs, outputs, arguments, results, and nodes are dead. The mark phase traverses the graph's edges, marking the γ-node's leftmost output alive. This renders the corresponding result origins of the γ-regions alive, then the leftmost output of the θ-node, and so forth. After the mark phase annotated all outputs and arguments as alive, the sweep phase removes all dead entities.

B1.6.2. Common Node Elimination

Common Node Elimination (CNE) permits the removal of redundant computations by detecting congruent nodes. These nodes always produce the same results, enabling the redirection of their result edges to a single node. This renders the other nodes dead, permitting DNE to remove them. CNE is similar to common subexpression elimination and value numbering [5] in that it detects equivalent computations, but since the RVSDG represents all computations uniformly as nodes, it can be extended to conditionals [174], loops, and functions.

We consider two simple nodes n_1 and n_2 congruent, or $n_1 \cong n_2$, if they represent the same computation, have the same number of inputs, *i.e.*,, $\left|I_{n_1}\right| = \left|I_{n_2}\right|$, and the inputs $i^k_{n_1}$ and $i^k_{n_2}$ are congruent, or $i^k_{n_1} \cong i^k_{n_2}$, for all $k = [0..\left|I_{n_1}\right|]$. Two inputs are congruent if their respective origins $g^k_{n_1}$ and $g^k_{n_2}$ are congruent, *i.e.*,,

Algorithm V: DEAD NODE ELIMINATION

1. **MARK**: *Mark output or argument as alive and continue as follows:*
   - ω-**REGION ARGUMENT**: *Stop marking.*
   - ϕ-**NODE OUTPUT**: *Mark the result origin of the corresponding recursion variable.*
   - ϕ-**REGION ARGUMENT**: *Mark the input origin if the argument belongs to a context variable. Otherwise, mark the output of the corresponding recursion variable.*
   - λ-**NODE OUTPUT**: *Mark all result origins of the λ-region.*
   - λ-**REGION ARGUMENT**: *Mark the input origin if the argument is a dependency.*
   - θ-**NODE OUTPUT**: *Mark the θ-node's predicate origin as well as the result and input origin of the corresponding loop variable.*
   - θ-**REGION ARGUMENT**: *Mark the input origin and output of the corresponding loop variable.*
   - γ-**NODE OUTPUT**: *Mark the γ-node's predicate origin as well as the origins of all results of the corresponding exit variable.*
   - γ-**REGION ARGUMENT**: *Mark the input origin of the corresponding entry variable.*
   - **SIMPLE NODE OUTPUT**: *Mark the origin of all inputs.*
2. **SWEEP**: *Process all nodes in reverse topological order and remove them if they are dead. Otherwise, process them as follows:*
   - ω-**NODE**: *Recursively process the ω-region. Remove all dead arguments.*
   - γ-**NODE**: *For all exit variables $(R,o) \in EX$ where o is dead, remove o and all $r \in R$. Recursively process the γ-regions. For all entry variables $(i,A) \in EV$ where all $a \in A$ are dead, remove all $a \in A$ and i.*
   - θ-**NODE**: *For all loop variables $(i,a,r,o) \in LV$ where a and o are dead, remove o and r. Recursively process the θ-region. Remove i and a.*
   - λ-**NODE**: *Recursively process the λ-region. For all context variables $(i,a) \in CV$ where a is dead, remove a and i.*
   - ϕ-**NODE**: *For all recursion variables $(r,a,o) \in RV$ where a and o are dead, remove o and r. Recursively process the ϕ-region. Remove a. For all context variables $(i,a) \in CV$ where a is dead, remove a and i.*

$g^k_{n_1} \cong g^k_{n_2}$. By definition, the origins of inputs are either outputs of simple or structural nodes, or arguments of regions. Origins from simple nodes are only equivalent when their respective producers are computationally equivalent, whereas for the other cases, it must be guaranteed that they always receive the same value.

The implementation of CNE consists of two phases: mark and divert. The mark phase identifies congruent simple nodes, while the divert phase diverts all edges of their origins to a single node, rendering all other nodes dead. Both phases of Algorithm VI perform a simple top-down traversal, recursively processing subregions of structural nodes annotating inputs, outputs, arguments, and results, as well as simple nodes as congruent. For γ-nodes, the algorithm marks only computations within a *single* region as congruent and performs no analysis between regions. In the case of θ-nodes, computations are only congruent when they are congruent before and after the loop execution, *i.e.*,, the inputs and results of two loop variables must be congruent.

Figure B1.7b shows the RVSDG for the code in Figure B1.7a, and Figure B1.7b the RVSDG after CNE. Two of the four multiplications take the same inputs and are therefore congruent to each other. Thus, their result edges are redirected and they become dead. DNE can then remove both multiplications as shown in Figure B1.7d.

For simple nodes, the algorithm marks all nodes within a region that are congruent to a node n. In order to avoid costly traversals of all nodes for every node n, the mark phase takes the candidates from the users of the origin of n's first input. If there is another input from a simple node n' with the same operation and number of inputs among them, the other inputs from both nodes can be compared for congruence. Moreover, a region must store constant nodes, *i.e.*, nodes without inputs, separately from other nodes so that the candidate nodes for constants are available. For commutative simple nodes, the inputs should be sorted before their comparison.

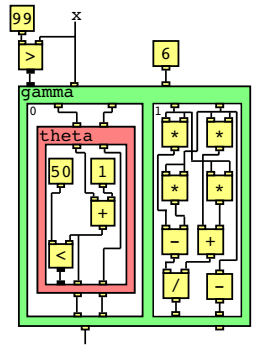
The presented algorithm only detects simple nodes as congruent within a region. For γ-nodes, congruence can also exist between nodes of different γ-regions and extending the algorithm would eliminate these redundancies. Another extension would be to permit congruence detection for structural nodes to implement conditional fusion [174] and loop fusion [125]. In the case of γ-nodes, it is sufficient to ensure that two nodes have congruent predicates, whereas

Algorithm VI: COMMON NODE ELIMINATION

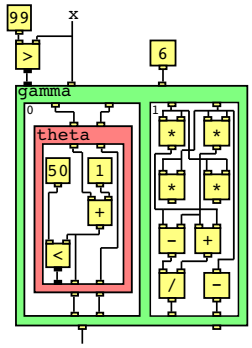
1. **MARK**: *Process all nodes in topological order as follows:*
   - **SIMPLE NODES**: *Denote this node as n. Mark n as congruent to all nodes n' which represent the same operation and where $|I_n| = |I_{n'}| \wedge i_n^k \cong i_{n'}^k$ for all $k = [0..|I_n|)$. Mark all outputs $o_n^k \cong o_{n'}^k$ for all $k = [0..|O_n|)$.*
   - **γ-NODE**: *For all entry variables $ev_1, ev_2 \in EV$ where $i_{ev_1} \cong i_{ev_2}$, mark $a_{ev_1}^k \cong a_{ev_2}^k$ for all $k \in [0..|A_{ev_1}|)$. Recursively process the γ-regions. For all exit variables $ex_1, ex_2 \in EX$ where $r_{ex_1}^k \cong r_{ex_2}^k$ for all $k \in [0..|R_{ex_1}|)$, mark $o_{ex_1} \cong o_{ex_2}$.*
   - **θ-NODE**: *For all loop variables $lv_1, lv_2 \in LV$ where $i_{lv_1} \cong i_{lv_2} \wedge r_{lv_1} \cong r_{lv_2}$, mark $a_{lv_1} \cong a_{lv_2}$ and $o_{lv_1} \cong o_{lv_2}$. Recursively process the θ-region.*
   - **λ-NODE**: *For all context variables $cv_1, cv_2 \in CV$ where $i_{cv_1} \cong i_{cv_2}$, mark $a_{cv_1} \cong a_{cv_2}$. Recursively process the λ-region.*
   - **ϕ-NODE**: *For all context variables $cv_1, cv_2 \in CV$ where $i_{cv_1} \cong i_{cv_2}$, mark $a_{cv_1} \cong a_{cv_2}$. Recursively process the ϕ-region.*
   - **ω-NODE**: *Recursively process the ω-region.*
2. **DIVERT**: *Process all nodes in topological order as follows:*
   - **SIMPLE NODES**: *Denote this node as n. For all nodes n' which are congruent to n, divert all outputs $o_{n'}^k$ to o_n^k for all $k = [0..|O_n|)$.*
   - **γ-NODE**: *For all entry variables $ev_1, ev_2 \in EV$ where $i_{ev_1} \cong i_{ev_2}$, divert all edges from $a_{ev_2}^k$ to $a_{ev_1}^k$ for all $k \in [0..|A_{ev_1}|)$. Recursively process the γ-regions. For all exit variables $ex_1, ex_2 \in EX$ where $r_{ex_1}^k \cong r_{ex_2}^k$ for all $k \in [0..|R_{ex_1}|)$, divert all edges from o_{ex_2} to o_{ex_1}.*
   - **θ-NODE**: *For all induction variables $lv_1, lv_2 \in LV$ where $a_{lv_1} \cong a_{lv_2} \wedge o_{lv_1} \cong o_{lv_2}$, divert all edges from a_{lv_2} to a_{lv_1} and from o_{lv_2} to o_{lv_1}. Recursively process the θ-region.*
   - **λ-NODE**: *For all context variables $cv_1, cv_2 \in CV$ where $i_{cv_1} \cong i_{cv_2}$, divert all edges from a_{cv_2} to a_{cv_1}. Recursively process the λ-region.*
   - **ϕ-NODE**: *For all context variables $cv_1, cv_2 \in CV$ where $i_{cv_1} \cong i_{cv_2}$, divert all edges from a_{cv_2} to a_{cv_1}. Recursively process the ϕ-region.*
   - **ω-NODE**: *Recursively process the ω-region.*

```
int y = 6;
if (99 > x) {
  z = (x*x)-(y*y)
    / (y*y)+(x*x);
  w = -y;
} else {
  do {
   x++;
  } while(50 < x);
  z = x;
  w = y;
}
```

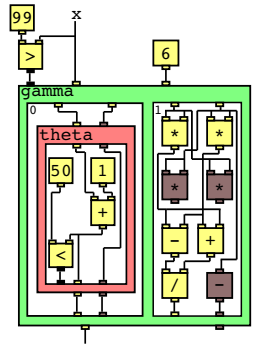
(a) Code



(b) RVSDG



(c) After CNE



(d) After DNE mark

Figure B1.7.: Dead and Common Node Elimination

for θ-nodes it would be necessary to permit congruence detection between different θ-regions to ensure that their predicates are the same.

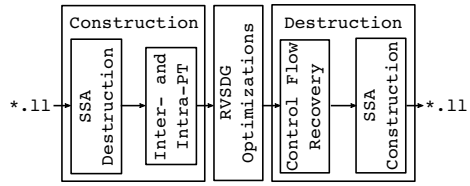
B1.7. Implementation and Evaluation

This section's goal is to demonstrate that the RVSDG has no inherent impediment that prevents it from producing competitive code and that it can serve as the IR in a compiler's optimization stage. The goal is not to outperform mature compilers, such as LLVM or GCC. This would require a significant engineering effort, which is outside the scope of this article. In light of this goal, we evaluate the RVSDG in terms of performance and size of produced code, as well as compilation time and representational overhead. We further provide a qualitative comparison between the RVSDG and the CFG-based implementation of LLVM.

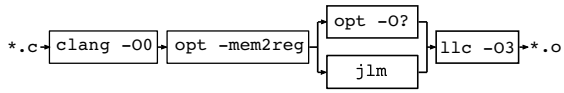
B1.7.1. Evaluation Setup

We have implemented *jlm*, a publicly available [165] prototype compiler that uses the RVSDG for optimizations. Its compilation pipeline is outlined in Figure B1.8a. Jlm takes LLVM IR as input, constructs an RVSDG, transforms and optimizes this RVSDG, and destructs it again to LLVM IR. The SSA form of the input is destructed before RVSDG construction proceeds with Inter- and Intra-PT. This additional step is required due to the control flow restructuring phase of Intra-PT. Destruction discovers control flow by employing SCFR before it constructs SSA form to output LLVM IR. Jlm's LLVM IR support is currently limited to function, integer, floating point, pointer, array, and structure 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 [158] to evaluate the RVSDG's usability and efficacy. This benchmark suite provides structurally small benchmarks, and therefore reduces the implementation effort for the construction and destruction phases, as well as the number and complexity of optimizations.



(a) Jlm compilation pipeline.



(b) Evaluation setup.

Figure B1.8.: Jlm's compilation pipeline and evaluation setup.

Figure B1.8b outlines our evaluation setup. We use clang 4.0.1 [45] 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 the following optimizations in addition to DNE (subsection B1.6.1) and CNE (subsection B1.6.2):

- **Inlining** (ILN): Inlines functions that are not exported and only called once.
- **Invariant Value Redirection** (INV): Redirects invariant values from θ- and γ-nodes. For γ-nodes, the users of an exit variable's output can be redirected to the origin of an entry variable's input, if the origin for all the results of the exit variable are the corresponding arguments of the entry variable. For θ-nodes, a loop variable is invariant if the origin of its result is the argument of the loop variable.
- **Node Push Out** (PSH): Moves all invariant nodes out of γ- and θ-regions. For γ-nodes, all nodes without side-effects are moved, exposing them to other optimizations such as CNE. For θ-nodes, a node is invariant if all its operands are invariant.

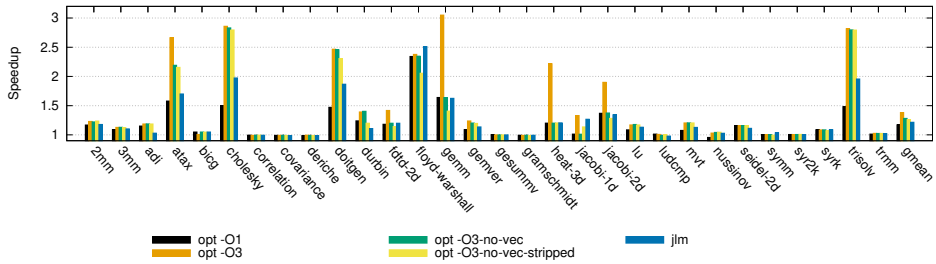


Figure B1.9.: Speedup relative to `O0` at different optimization levels.

- **Node Pull In** (PLL): Moves all nodes that are only used in one γ-region into the γ-node. This ensures their conditional execution, while avoiding code bloat.

- **Node Reduction** (RED): Performs simplifications such as constant folding or strength reduction. This performs similar reductions as LLVM's redundant instruction combinator (`-instcombine`), albeit by far not as many.

- **Loop Unrolling** (URL): Unrolls all inner loops by a factor of four. Higher factors gave no significant performance improvements in return for the increased code size.

- $\theta-\gamma$ **Inversion** (IVT): Inverts γ- and θ-nodes where both nodes have the same predicate origin. This replaces the loop containing a conditional with a conditional that has a loop in its then-case.

We use the following optimization order: ILN INV RED DNE IVT INV DNE PSH INV DNE URL INV RED CNE DNE PLL INV DNE. 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.

B1.7.2. Performance

Figure B1.9 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 B1.9 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 the output files and found the following causes:

- **adi**: Jlm fails to eliminate load instructions from the two innermost loops. These loads have loop-carried dependences with a distance of one to store instructions in the same loop, and can be eliminated by propagating the stored value to the users of the load's output. The LLVM pass that performs this optimization is loop load elimination (`-loop-load-elim`). If this transformation is performed by hand on the two loops, then jlm achieves the same performance as `O1`.

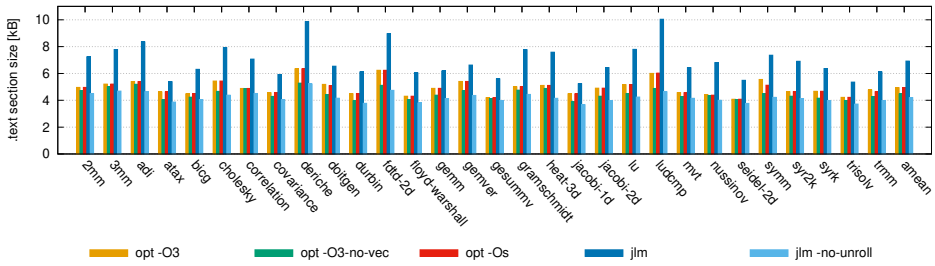


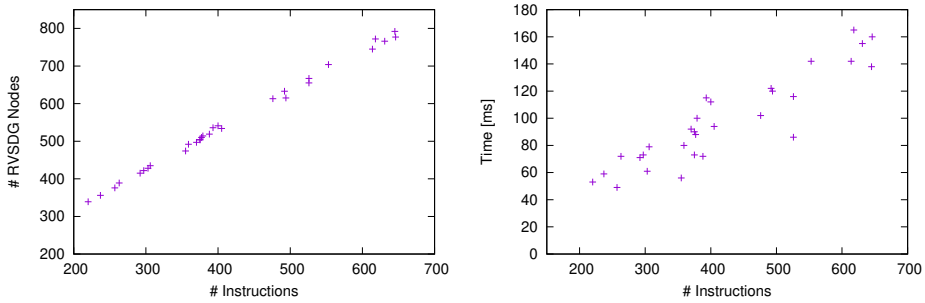
Figure B1.10.: Code size at different optimization levels.

- **durbin**: Jlm fails to transform a loop that copies values between arrays to a `memcpy` intrinsic. This impedes LLVM's code generator to produce better code. The LLVM pass responsible for this transformation is the loop-idiom pass (`-loop-idiom`). If the loop is replaced with a call to `memcpy`, then jlm achieves better performance than `O1`.
- **ludcmp**: Jlm is worse than `O1`, but equal to `O3-no-vec`. This suggests that the difference is due to missing vectorization.
- **seidel-2d**: Similarly as for adi, jlm fails to eliminate load instructions from the innermost loop. If the load elimination is performed by hand, then jlm achieves the same performance as `O1`.

Figure B1.9 shows that it is feasible to produce competitive code using the RVSDG, but also that more optimizations and analyses are required in order 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 in order to compete with mature compilers at more complex benchmarks.

B1.7.3. Code Size

Figure B1.10 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



(a) Representational overhead.

(b) Compilation times.

Figure B1.11.: Compilation overhead of jlm.

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.

B1.7.4. Compilation Overhead

Figure B1.11 shows the overhead in terms of IR size and time for the RVSDG. Figure B1.11a shows the representational overhead by relating the number of

instructions in the LLVM module to the number of RVSDG nodes after construction, whereas Figure B1.11b relates the number of instructions in the LLVM module to the time spent on RVSDG construction, optimizations, and RVSDG destruction.

Figure B1.11a shows a clear linear relationship for all cases, confirming the observations by Bahmann *et al.* [13] that the RVSDG is feasible in terms of space requirements. Figure B1.11b also indicates a linear dependency, but with larger variations for similar input sizes. This variation can be attributed to the fact that construction, destruction, as well as optimizations are also compounded by input structure. Structural differences in the inter-procedure and control flow graphs lead to runtime variations in RVSDG construction and destruction, as well as different runtimes for optimizations. For example, the presence of loops in a translation unit determines whether loop unrolling is performed, while their absence avoids the runtime overhead for this optimization completely. Overall, Figure B1.11 suggests that the RVSDG is feasible as an IR for optimizing compilers in terms of compilation overhead.

B1.7.5. Comparison to LLVM

LLVM invokes 77 different analyses and optimization passes at optimization level `O3`. Many of them are invoked repeatedly, resulting in a total of 199 invocations. Table B1.1 shows the ten most invoked passes as well as the count for SSA Restoration. In total, these passes comprise 99 invocations or ca. 50% of all invocations. In this section, we detail each of these passes and compare them to a (potential) RVSDG implementation:

- **(Basic) Alias Analysis**: The basic alias analysis pass performs a stateless alias analysis, while the other pass forwards alias results to subsequent optimizations. The RVSDG can help to reduce dependencies between the alias analysis pass and other transformations, as its state edges permit to encode alias information directly in the graph. For example, two stores that never alias can be rendered independent by redirecting their state origin to the same output (see Figure B1.2b). This encodes the no-alias information in the graph, and enables other optimizations to perform their transformation without an additional representation for

Table B1.1.: Ten most invoked LLVM passes at optimization level `O3`.

Optimization	Flag	# 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

alias information. However, further research is required before any conclusive answers can be drawn.

- **Dominator Tree Construction**: The Dominator Tree Construction pass finds the CFG's forward dominator information. Dominator trees are an essential part of CFG-based compilers, as they permit simple traversal of cyclic CFGs and are used in a myriad of transformations, such as loop detection, SSA (re-)construction, and code hoisting. Its reconstruction is required as optimizations, such as jump threading or loop unrolling, alter CFG structure. The computation of dominators is unnecessary in an RVSDG setting. The dependencies between nodes already encode such information, and the RVSDG's acyclic structure as well as its region tree permit efficient traversal. Moreover, the RVSDG is always in SSA form and requires no SSA restoration. Although many CFG-based optimizations of LLVM use dominator information, none of the corresponding optimizations in jlm need such information.

- **Scalar Evolution Analysis**: The scalar evolution pass analyzes and categorizes scalar expressions in loops, recognizes induction variables, and represents them as abstract expressions. It is primarily used to support

induction variable substitution and strength reduction. The RVSDG seems to offer no simplifications for this analysis, and a similar implementation would be required in an RVSDG setting.

- **Natural Loop Canonicalization**: The Natural Loop Canonicalization pass canonicalizes loops to simplify and improve subsequent analyses, such as Loop Invariant Code Motion (LICM). The transformation inserts a single pre-header block to ensure a single entry edge into the loop and basic blocks to ensure that all exit blocks are dominated by the loop header. It also guarantees that loops have exactly one back edge after the transformation. Natural Loop Canonicalization is invoked before every major loop optimization to ensure a canonicalized loop form. This pass is irrelevant in an RVSDG setting as loops are already in this form. The RVSDG's θ-nodes represent tail-controlled loops, *i.e.*, `do-while` loops, which have exactly one incoming edge and one back edge, and its exit block is dominated by the loop.
- **Redundant Instruction Combinator**: The Redundant Instruction Combinator pass combines and simplifies instruction sequences, and performs algebraic simplifications. This pass corresponds to the Node Reduction (RED) pass in jlm.
- **Loop-Closed SSA Form (Verifier)**: The Loop-Closed SSA (LCSSA) Form pass transforms loops by inserting phi instructions at the end of loops for all values that are still live after a loop. Its corresponding verifier pass is always executed before the LCSSA pass and its sole purpose is to indicate when to run LCSSA verification. The LCSSA Form pass canonicalizes loops to simplify other loop optimizations. Continuous reconstruction of this form is required, as it is not explicitly enforced by the IR. These passes are irrelevant in an RVSDG setting as loops are always in this form. The explicit representation of loops as nodes indicates their boundaries, and the loop variables of θ-nodes mark all values that are live across a loop boundary.
- **CFG Simplifier**: The CFG Simplifier pass removes dead basic blocks, merges linear subgraphs, eliminates phi instructions from basic blocks with a single predecessor, removes empty basic blocks, and performs a myriad of other simplifications. This pass normalizes the CFG, and removes code artifacts from other optimizations, such as the LCSSA pass.

The transformations performed by the CFG Simplifier are manifold, and are either unnecessary in an RVSDG setting, or are performed by other optimization passes such as DNE, PSH, or RED. Examples of the former are linear subgraph merging or empty basic block removal, while examples of the latter are dead basic block removal (DNE) or code hoisting (PSH).

- **Natural Loop Information**: The Natural Loop Information pass identifies loops and determines their nesting structure as well as depth. This pass is the starting point of every loop transformation as it identifies the loops in the CFG. This pass is unnecessary in an RVSDG setting as loops are explicitly represented and their nesting structure is encoded in the region tree.

- **SSA Restoration**: Compilers using a CFG in SSA form as IR need to restore this form every time optimizations change the CFG structure. In LLVM, SSA restoration is not performed as an individual pass, but ad hoc as part of individual optimizations. Many of these optimizations, but not all, use the SSAUpdater class to restore SSA form. We performed a simple search on the LLVM source for `SSAUpdate.h`, and found twelve transformations that include this file. This indicates that SSA restoration is performed regularly in LLVM. In contrast, SSA restoration is irrelevant in the RVSDG, as the graph is by definition in SSA form and is not valid without it.

If SSA Restoration is excluded, then six of the ten most invoked passes of LLVM are not at all, or only partially required in the RVSDG. These passes are Dominator Tree Construction, Natural Loop Canonicalization, Loop-Closed SSA Form (Verifier), CFG Simplifier (partial), and Natural Loop Information. Most of these passes are only helper passes. They extract structure from the IR, establish invariants, or simplify it to a canonical form such that actual optimizations can be performed. Excluding the CFG Simplifier pass and SSA Restoration, these passes comprise 45 invocations, or ca. 23% of the total invocations. By design, the RVSDG provides already the information and invariants established by these passes, and therefore can omit their implementation as well as avoid the associated compile time overhead. This simplifies the complexity of the compiler and makes optimizations more resilient as they are less

dependent on previously executed helper passes. In contrast, LLVM must regularly invoke these passes to (re-)discover loops, simplify and canonicalize the IR, and establish invariants, as its CFG does not provide the relevant abstractions.

B1.8. Related Work

A cornucopia of IRs has been presented in the literature in order to better expose desirable program properties for optimizations. Here, we discuss the properties of the more prominent ones, highlighting their strengths and weaknesses.

The Control Flow Graph (CFG) [4] is the predominant IR for imperative languages [199]. It explicitly represents the possible control flow paths of programs, simplifying certain analyses, such as loop identification or irreducibility detection, as well as enabling simple target code generation. Its translation to SSA form [52] additionally improves the efficiency of data flow optimizations [223, 172]. However, many classical optimizations are based on data flow rather than control flow information. The CFG complicates these optimizations by representing data flow implicitly and imposing a total order upon its instructions, potentially requiring the construction of other IRs, such as the data flow graph [55]. Optimizations, such as jump threading or loop unrolling, break its SSA property, requiring SSA reconstruction [43]. Moreover, the CFG only represents a single procedure and another IR, such as the call graph, needs to be employed to express inter-procedural dependencies. Thus, while the CFG is very easy to construct and destruct, it exposes only limited program information required for efficient analyses and optimizations. Program structure, such as loops and their nesting, as well as SSA form need to be explicitly discovered, and in case of SSA form also maintained. The CFG is neither complete, *i.e.*,, other IRs are necessary, nor does it provide a strong normalization effect with its total instruction order.

The Program Dependence Graph (PDG) [71, 87] combines control and data flow within a single representation. It features two major edge types to represent data and control flow, while nodes represent statements, predicates, or regions, which group all nodes with the same control dependencies. Thus,

if the control dependency of a region is fulfilled, then all its children could potentially be executed in parallel. Region nodes relax the total ordering imposed by the CFG, and combined with the unified representation of data and control dependencies, certain optimizations, such as vectorization [17], are simplified. An extension to incorporate collection of procedures was presented as the System Dependence Graph (SDG) by Horwitz *et al.* [88]. However, the PDG suffers from aliasing and side-effect problems, because it supports no clear distinction between data held in registers or memory [98]. Moreover, the PDG does not enforce SSA form and therefore requires additional effort to maintain it. It features a large number of different edge types, five in Ferrante *et al.* [71] and four in Horwitz *et al.* [87], increasing maintenance costs during and after transformations to ensure its invariants. Thus, the PDG is superior to the CFG in that it is more normalizing and complete. It relaxes the CFG's total instruction order through region nodes and the SDG enables the representation of an entire program. Its alias and side-effect problems however complicate or can even preclude construction altogether [98], and its abundance of edge types negatively affects transformability. Program structure and SSA form still need to be discovered and maintained.

The Value Dependence Graph (VDG) [224] abandons the explicit representation of control flow. It only expresses the flow of values using ports. Nodes represent simple operations, γ-nodes the selection between values, and λ-nodes represent functions. The VDG expresses loops as recursive functions, using a single construct to express both concepts. It is implicitly in SSA form, since each value corresponds to exactly one port. A node depends solely on its data dependencies, and therefore only on the values needed for its operation, avoiding the total ordering imposed by the CFG. However, modeling only data flow raises a significant problem with the preservation of program semantics, as the "evaluation of the VDG may terminate even if the original program would not..." [224]. Another issue is its representation of loops as functions. On the one hand, this enables loop optimizations for functions and *vice versa*, but on the other hand, it might complicate these optimizations, as they have to handle both without being able to distinguish between them.

The Value State Dependence Graph (VSDG) [98, 111] addresses the termination problem of the VDG by introducing state edges. These edges are used to model the sequential execution of operations with side-effects. In addition to nodes for simple operations and γ-nodes, it introduces θ-nodes to explicitly

represent loops. Like the VDG, the VSDG is implicitly in SSA form, and nodes are solely dependent on required operands, avoiding the total order of instructions imposed by the CFG. The VSDG supports no functions, and therefore is not able to represent an entire program. Its γ-nodes only select between two values based on a predicate. Thus, γ-nodes must be combined throughout destruction in order to express conditionals with multiple values. Moreover, all nodes are represented in a flat graph, which simplifies optimizations [98], but has a severe effect on evaluation semantics. Operations with side-effects, which originally resided in different control flow paths, are no longer guarded by a conditional. Instead, γ-nodes are used to choose the correct value based on the conditional's predicate, and care must be taken to avoid duplicated evaluation of operations with side-effects. In fact, for graphs with stateful computations, lazy evaluation seems to be the only safe strategy [111].

B1.9. Conclusion

This paper presents a complete specification of the RVSDG IR for optimizing compilers. We complement the construction and destruction algorithms from previous work, and show the RVSDG's efficacy as an IR for analyses and optimizations by presenting Dead Node and Common Node Elimination. We implemented jlm, a publicly available [165] compiler that uses the RVSDG for optimizations, and evaluate it in terms of performance, code size, compilation time, and representational overhead, as well as compared it to a widely used CFG-based compiler. The results suggest that the RVSDG combines the abstractions of data centric IRs with the CFG's advantages to optimize and generate efficient control flow. This makes the RVSDG an appealing IR for optimizing compilers. A natural direction for future work is to extend jlm to support the entire LLVM IR, exploring how features such as exceptions can be efficiently mapped to the RVSDG. Another research direction would be to extend the number of optimizations and their heuristics in jlm to a competitive level with CFG-based compilers. This would provide further information about the number of necessary optimizations, their complexity, and consequently the required engineering effort.

B2. Perfect Reconstructability of Control Flow from Demand Dependence Graphs

Helge Bahmmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer

Published in
ACM Transactions on Architecture and Code Optimization (TACO)

Abstract. Demand-based dependence graphs (DDGs), such as the (Regionalized) Value State Dependence Graph ((R)VSDG), are intermediate representations (IRs) well suited for a wide range of program transformations. They explicitly model the flow of data and state, and only implicitly represent a restricted form of control flow. These features make DDGs especially suitable for automatic parallelization and vectorization, but cannot be leveraged by practical compilers without efficient construction and destruction algorithms. Construction algorithms remodel the arbitrarily complex control flow of a procedure to make it amenable to DDG representation, whereas destruction algorithms reestablish control flow for generating efficient object code. Existing literature presents solutions to both problems, but these impose structural constraints on the generatable control flow, and omit qualitative evaluation.

The key contribution of this paper is to show that there is no intrinsic structural limitation in the control flow directly extractable from RVSDGs. This fundamental result originates from an interpretation of loop repetition and decision predicates as computed continuations, leading to the introduction of the *predicate continuation* normal form. We provide an algorithm for constructing RVSDGs in predicate continuation form, and propose a novel destruction

algorithm for RVSDGs in this form. Our destruction algorithm can generate arbitrarily complex control flow, and we show this by proving that the original CFG an RVSDG was derived from can apart from over-specific detail be reconstructed perfectly. Additionally, we prove termination and correctness of these algorithms. Furthermore, we empirically evaluate the performance, the representational overhead at compile time, and the reduction in branch instructions compared to existing solutions. In contrast to previous work, our algorithms impose no additional overhead on the control flow of the produced object code. To our knowledge, this is the first scheme that allows the original control flow of a procedure to be recovered from a DDG representation.

B2.1. Introduction

The main intermediate representation (IR) for imperative languages in modern compilers is the Control Flow Graph (CFG) [199]. It explicitly encodes *arbitrarily complex* control flow, admitting unconstrained control flow optimizations and efficient object code generation. Translation to static single assignment (SSA) form [52] additionally enables efficient data flow optimizations [223, 172], but the implicit data flow and specific execution order of CFGs restrict their utility by complicating these optimizations unnecessarily [111].

Data flow centric IRs have developed from the insight that many classical optimizations are based on the flow of data, rather than control. DDGs such as the (Regionalized) Value State Dependence Graph ((R)VSDG) [98] show promising code quality improvements and reduced implementation complexity. These IRs *implicitly* represent control flow in a structured form, which allows for simpler and more powerful implementations of data flow optimizations such as common subexpression and dead code elimination [197, 98, 111]. DDGs require a construction algorithm to translate programs from languages with extensive control flow support, and a destruction algorithm to extract the control flow necessary for object code generation.

Johnson [98] constructs the VSDG of a C program from its abstract syntax tree (AST), and excludes goto and switch statements to obtain a reducible subset of the language. Stanier [196] extends VSDG to irreducible control flows, constructing it from the CFG by structural analysis [183]. Several patterns are

matched in the CFG and converted with per-pattern strategies, before individual translations are connected in the VSDG. Irreducible graphs are handled by node splitting, which can lead to exponential code blowup [37]. Johnson's VSDG destruction [98] adds state edges until execution order is determined. Resulting CFGs follow the control flow structure of their VSDG, potentially increasing code size and branch instruction count compared to the original program. Lawrence [111] translates the VSDG to a *program dependence graph (PDG)* [71] by encoding a lazy evaluation strategy, transforms the PDG to a restricted, duplication-free form, and converts it to a CFG. Introducing the PDG to refine a VSDG into exactly one CFG substantially complicates destruction, but the overhead of the resulting control flow is not quantified. To our knowledge, previous work omits quantitative analysis, or inherently features code size growth and/or suboptimal control flow recovery. No proposed algorithm has been analyzed with respect to optimality.

In this paper, we show that the RSVDG representation does not impose any structural limit on control flows obtainable from it. Interpreting loop repetition and decision predicates as computed continuations, we introduce the *predicate continuation form* as a normal form, and propose algorithms for RSVDG construction and destruction. The construction handles complex control flow without node splitting, which avoids code blowup. Destruction uses the predicate continuation form to extract control flow. Multiple CFGs map to the same RVSDG because they contain inessential detail information (evaluation order of independent operations, variable names; cf. Section B2.3) that is irretrievably lost during RVSDG construction. This leads to degrees of freedom in refining an RVSDG into a specific CFG where "arbitrary" choices lead to viable results. We show that there are choices under which our destruction algorithm perfectly reconstructs the original CFG an RVSDG has been generated from (cf. Theorem B2.5.8). This leads to the insight that our algorithm is universal in the sense that it can generate *arbitrary* control flow. We prove termination and correctness of our algorithms, experimentally evaluate performance and representational overhead, and compare the reduction of branch instructions to previous work. For practical programs, we observe that processing time and output size empirically correlate linearly with input size. This suggests that our algorithms are fit for field application. Thus, we demonstrate that control flow optimizations can be lifted to DDG representations, enabling the use of a single IR for data and control flow optimizations.

This reduces use of the CFG to a step in DDG generation for languages with complex control flow.

The paper is organized as follows: Section B2.2 introduces terminology and definitions. Section B2.3 describes a destruction algorithm that produces a CFG from an RVSDG in predicate continuation form. Section B2.4 develops a construction algorithm that restructures a CFG and translates it to an RVSDG. Section B2.5 proves algorithm termination and correctness, and that our destruction scheme can always recover the original CFG. We empirically evaluate our algorithms using CFGs from SPEC2006 benchmarks in Section B2.6. Section B2.7 discusses related work, and Section B2.8 concludes our work.

B2.2. Terminology and Definitions

This section provides necessary terminology and definitions. It defines the CFG and RVSDG, as well as restricted subsets which are used throughout the paper.

B2.2.1. Control Flow Graph

A *control flow graph* **C** is a directed graph consisting of vertices representing statements, and arcs representing transitions between statements. If a vertex has multiple outgoing arcs, we assign a unique index to each. Statements take the following form:

- $v_1, v_2, \ldots, v_k$:=$expr$ designates an assignment statement. The $expr$ must evaluate to a k-tuple of value and/or states[1] which are assigned to the variables on the left.
- `branch` $expr$ designates a branch, where $expr$ evaluates to an integer that matches an outgoing arc index. Execution resumes at the destination statement of this arc.

[1] We allow multiple states to be alive at the same time here and in our later RVSDG definition if the states are independent. Use cases include e.g. modelling disjoint memory regions after aliasing analysis.

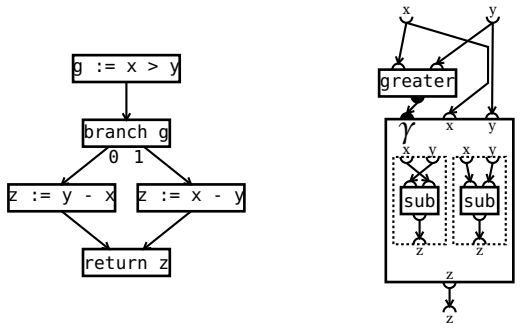


Figure B2.1.: CFG (left) and RVSDG (right) representations of the same function, computing $x - y$ if $x > y$ and $y - x$ otherwise.

- `null` designates a null operation. Its operational semantics is to perform no operation, we insert it for structural reasons[2].

All vertices except for `branch` statements must have at most one outgoing arc. Essentially, the CFG represents a single procedure of a program in imperative form: From a given start vertex, evaluate all statements of a vertex in sequential order by strictly evaluating the expression of each statement, updating the variable state on each assignment statement, and follow alternative arcs according to `branch` statements.

Definition 8. *A CFG is called* closed *iff it has a unique* entry *and* exit *vertex with no predecessors and successors, respectively. A CFG is called* linear *if it is closed and each vertex has at most one predecessor and one successor.*

Definition 9. *The* dynamic execution trace *of a closed CFG for given arguments is the sequence of traversed vertices and outgoing arcs, starting from the entry vertex.*

Figure B2.1 contains an example CFG. We insert a `return` statement at the end of a procedure in a CFG to clarify its result (its operational semantics is the same as `null`). Bold capital letters like **C** denote CFGs. Letters v, a, V, and

[2] It can be regarded as a "dummy" assignment operation, but we want to distinguish such structural statements from original program statements.

A denote vertices, arcs, vertex sets and arc sets, respectively. Without loss of generality, we assume that all CFGs are in *closed* form. We also consider CFGs of a more constrained shape:

Definition 10. *A closed CFG is called* structured *if its shape can be contracted into a single vertex by repeatedly applying the following steps:*

1. *If v' is unique successor of v, and v is unique predecessor of v', then remove arc $v \to v'$ and merge v and v'.*
2. *If v has only successors $v_0, v_1, \ldots$ and possibly v', v' has only predecessors $v_0, v_1, \ldots$ and possibly v, and each of $v_0, v_1, \ldots$ has only a single predecessor and successor, then remove all arcs $v \to v_i$, $v_i \to v'$, $v \to v'$ and merge all vertices.*
3. *If v has an arc pointing to itself and only one other successor, remove the arc $v \to v$.*

Structured CFGs are a subset of reducible CFGs. They correspond to procedures with only structured control flow in the following sense: Within each structured CFG we can identify subgraphs that are structured CFGs themselves[3], and replace them with a single vertex[4] such that the parent graph ends up in one of three possible shapes:

1. **Linear**: A linear chain of vertices (possibly a single vertex).
2. **Branch**: A symmetric control flow split and join, i.e an "`if/then/else`" or "`switch/case` without fall-through" construct where each alternative path is either a linear chain or a single arc.
3. **Loop**: A tail-controlled loop, i.e. a "`do/while`" loop where the loop body itself is a linear chain or a single arc.

Applying this recursively, we can regard each structured CFG as a *tree* of such subgraphs which we call *regions*. The CFG shown in Figure B2.1 is structured. We use an overline marker to designate structured CFGs: $\overline{\mathbf{C}}$.

[3] ignoring a potential single "repetition" arc from the exit to entry vertex of a subgraph
[4] This corresponds to contraction using the rules in Definition 10.

Definition 11. *The* structure multigraph[5] *of a closed CFG is formed by taking the original CFG and replacing all vertices with a single predecessor and successor by a direct arc from the predecessor to successor. The* projection of a dynamic execution trace *of a closed CFG is then obtained by omitting all vertices/arcs with a single predecessor and successor and projecting the remainder. Closed CFGs* $\mathbf{C}$ *and* $\mathbf{C}'$ *are* structurally equivalent *if there is an isomorphism between the two structure multigraphs such that it maps projected execution traces for the same input arguments to each other.*

Intuitively, structural equivalence means that two CFGs have the same dynamic branching structure for all possible arguments.

Definition 12. *An arc* $v_1 \to v_2$ dominates *a vertex* v *if either*

1. $v_2 \neq v$ *and both* v_1 *and* v_2 *dominate* v*, or*
2. $v_2 = v$ *and* v_1 *dominates* v*.*

The dominator graph of arc a *in* $\mathbf{C}$ *is the subgraph* $\mathbf{S}$ *with vertices* V *dominated by* a*.*

Intuitively, the dominator graph of an arc a is the subgraph in $\mathbf{C}$ where every path from the entry vertex to every vertex in this subgraph must pass through arc a.

B2.2.2. Regionalized Value State Dependence Graph

An RVSDG R is a directed acyclic hierarchical multigraph consisting of nodes representing computations, and edges[6] representing the forwarding of results of one computation to arguments of other computations. Each node has a number of typed *input* ports and *output* ports corresponding to parameters and results, respectively. An edge connects an input to exactly one output of matching type. The types of inputs and outputs may either be *values* or

[5] We want to preserve arcs as present in the original graph, but due to the removal of vertices we may end up with multiple distinguishable arcs between two vertices. The result may not be a graph, but a multigraph.

[6] We use the terms arc/vertex in the context of CFGs and edge/node in the context of RVSDG to assist in telling the different representations apart.

represent the *state* of an external entity involved in a computation. The distinguishing property is that values are always assumed to be copyable. The types of the input/output tuple of a node are called its *signature*. We model arguments and results of an RVSDG as free-standing ports.

The RVSDG supports two different kinds of nodes: *simple* and *complex* nodes. Simple nodes represent primitive operations such as addition, subtraction, load and store. They map a tuple of input values/states to a tuple of output values/states, and the node's arity and signature must therefore correspond to the represented operator. Complex nodes contain other RVSDGs, such that we can regard an RVSDG as a *tree* of such subgraphs which we call *regions*. We require two kinds of complex nodes in this paper:

- *Gamma nodes* represent decision points. Each γ-node contains two or more RVSDGs with matching result signatures. A γ-node takes as input a *decision predicate* that determines which of its regions is to be evaluated. Its other inputs are mapped to arguments of the contained RVSDGs, and its outputs are mapped to the results of the evaluated subgraph (cf. Figure B2.1).
- *Theta nodes* represent tail-controlled loops. Each θ-node contains exactly one RVSDG representing the loop body. Matching arguments/inputs and results/outputs of this region and the θ-node are used to represent the evolution of values/states through loop iterations. An additional *loop predicate* as result of the loop body determines if the loop should be repeated (cf. left of Figure B2.2).

The RVSDG represents a single procedure of a program in demand-dependence form. Arguments and results of the root region map to arguments and results of the procedure. The semantics of an RVSDG is that evaluation is demand-driven but strict, i.e. all arguments of a node are evaluated before the node itself is evaluated according to the following rules:

- *Simple nodes:* After evaluation of all arguments, apply the operator represented by the node to their values, and associate the results with the node outputs.
- *Gamma nodes:* After evaluation of all arguments (including the decision predicate), the predicate value dictates which sub region is chosen for

evaluation, mapping arguments of the gamma node to arguments of the sub region. The chosen region is evaluated and its results are associated with the outputs of the $\gamma-$node.

- *Theta nodes:* After evaluation of all arguments of the theta node, associate their values with the arguments of the sub region. Then evaluate the repetition predicate and all results from the loop body. If the value of the repetition predicate is non-zero, associate result values from this loop iteration with argument values for the next loop iteration and repeat evaluation. If the value of the repetition predicate is zero, associate the results of the last repetition with the outputs of the $\theta-$node.[7]

As explained in Section B2.3, an RVSDG is equivalent to a structured CFG, and since every CFG can be made into a structured one by the algorithm in Section B2.4, the topmost region of an RVSDG allows the representation of an entire procedure. We model the arguments and results of this region as free-standing ports.

An RVSDG must obey some additional structural restrictions to satisfy the non-copyability of states. We omit a thorough discussion of how these constraints can be formulated and maintained during transformations. Instead, we only require that it satisfies the following sufficient condition: All states are used linearly, i.e. in each region, a state output is connected to at most one state input.

Figure B2.1 illustrates a sample function in CFG and RVSDG representation: A γ-node selects the computation results from either of two embedded subgraphs, depending on its predicate input. By convention, the subgraphs embedded into a γ-node correspond from left to right to predicate values $0, 1, \ldots$. Predicate input/output ports are identified as filled sockets in diagrams.

We expect a richer type system to allow not just a distinction between values and states, but between different kind of values and/or states, e.g. different value types for fixed-point and floating point numbers. For this paper, we only

[7] Note that this definition requires that the evaluation of the loop body for one iteration is finished before the next iteration can commence. This avoids several problems such as "deadlocks" between computation of the loop body and the predicate. It also provides well-defined behavior for non-terminating loops that keep updating external state.

assume the existence of a *predicate* type that allows the enumeration of alternatives. We call a node that has at least one predicate output a *predicate def node*, and a node with at least one predicate input a *predicate use node*. This allows us to define a more strongly normalized form of RVSDGs:

Definition 13. *An RVSDG is in* predicate continuation form *iff*

- *Any predicate def node has exactly one successor*
- *Only one predecessor of any node may be a predicate def node*
- *Any predicate output must be connected to at most one predicate input, and that in turn must belong to a γ- or θ-node, or to the output of a region*

Any RVSDG can easily be converted to this form by a combination of node splitting/cloning (to satisfy the requirement that any predicate def has at most one use), merging multiple independent computations into a single node (to satisfy the requirement that at most one predecessor of a node is a predicate def node), and possibly inserting nodes to convert back and forth between predicates and other kinds of values (to satisfy the last requirement). Creating this normal form is essentially a technicality that we will not discuss in further detail, since all RVSDGs occurring in this paper are in this form by construction.

B2.3. Extracting Control Flow from the RVSDG

In this section, we present algorithms for extracting a CFG from an RVSDG. The dependence information contained within regions serves as basis for control flow construction: It describes a partial ordering of primitive operations that any control flow recovery must adhere to in order to yield an evaluation-equivalent program.

Any γ- and θ-node-free RVSDG can be trivially transformed into a linear CFG. In the presence of γ- or θ-nodes, the resulting CFG includes branching and/or looping constructs. We explore two different approaches: *Structured control flow recovery (PCFR)* uses the hierarchical *structure* of regions contained within γ- and θ-nodes to construct control flow. *Predicative control flow recovery (PCFR)* determines control flow by *predicate* assignments.

Since the RVSDG representation is more normalizing and lacks some detail of the CFG, we must employ some auxiliary algorithms:

- EVALORDER: The exact evaluation order within an RVSDG region is underspecified. Each topological order of nodes corresponds to a valid evaluation order, and one particular order must be chosen. We add only one additional constraint to the topological order: corresponding predicate def and use nodes must be adjacent if the given RVSDG is in predicate continuation form.
- VARNAMES: The RVSDG is implicitly in SSA form. We therefore need to find names for variables in the CFG representation such that boundary conditions for loops and control merge points are satisfied. This requires a strategy for computing an interference-free coloring of SSA variables such that same-colored SSA names can be assigned the same name in a CFG. Additionally, the insertion of copy operations may be necessary.

We consider algorithms for these problems mostly outside the scope of this paper, and assume that they are given. It is trivial to formulate an algorithm satisfying EVALORDER, e.g. some variant of depth first traversal. We briefly discuss VARNAMES in Section B2.3.1. Our control flow recovery algorithms are parameterized over these algorithms, and we discuss how they relate to perfect CFG recovery in Section B2.5.

B2.3.1. Copy insertion and coloring

We can formulate the conversion from an RVSDG to a CFG as a two step process: first insert ϕ expressions at control join points to obtain a CFG in SSA form, and then eliminate these ϕ expressions by inserting copy operations in the predecessors of ϕ's basic block [52]. This process succeeds as long as no critical arcs[8] are present in a CFG [29]. According to our definition of structured CFGs, only the loop repetition and exit arcs are critical: Structured "`if/then/else`" or "`switch/case` without fall-through" as per Definition 10 cannot produce critical arcs, leaving only those arcs additionally permitted

[8] An arc from v_1 to v_2 is critical iff v_1 has multiple successors and v_2 has multiple predecessors.

through loop constructs as possibly critical. These in turn are amenable to simplistic copy insertion: choose a name for each loop variant variable, and indiscriminately assign to it on the entry, repetition, and exit path of the loop.

In order to simplify the presentation, we lift part of this process to the RVSDG representation: After determining a topological order of nodes, we insert placeholder `copy` operations into the RVSDG. These `copy` operations permit the algorithms in the following sections to directly convert an RVSDG into an SSA-free CFG, and are replaced by parallel copy operations during this conversion[9]. Having made these observations, the following rules insert a sufficient number of copy operations:

Algorithm CopyInsertion

- *For each γ-node, insert a* `copy` *operation copying each result value as the last operation in each of the alternative subregions.*
- *For each θ-node, insert a* `copy` *operation copying each loop variant value twice: Once, just before the θ-node (loop entry), and once at the last operation in the subregion (loop exit/repetition).*

This algorithm inserts many redundant copy operations. Subsequently, coalescing based on an interference graph as per [26] can eliminate many or even all of these copy operations.

B2.3.2. Structured Control Flow Recovery

The naive approach to generating control flow is to treat γ- and θ-nodes as black boxes from the outside, and represent more fine-grained control flow on the inside. We call this approach *structured control flow recovery (SCFR)*, and formulate it as follows:

Algorithm StructuredControlFlow

Sequentially process all nodes of a given RVSDG. For each node, insert vertices into the CFG according to the following rules, and add arcs between them for continuation:

[9] The evaluation semantics of these `copy` operations within the RVSDG is that they just pass through all argument values as results without change.

1 *For each simple node (including* `copy` *nodes), insert an assignment statement evaluating an expression equivalent to the semantics of the node into the CFG.*

2 *For each γ-node, recursively process the subregions into separate CFGs. Insert a* `branch` *statement corresponding to the decision predicate of the γ-node, and fan out to the entry points of the sub-CFGs. Rejoin control flow from the exits of the sub-CFGs with the next vertex to be added to the CFG.*

3 *For each θ-node, recursively process the subregion into a separate CFG. Link to the entry of the sub-CFG, and add a* `branch` *statement after the exit. This corresponds to the loop predicate of the θ-region, and either repeats the sub-CFG, or continues with the next vertex to be added.*

If necessary, add a `null` *vertex to rejoin any remaining control splits, or provide a loop exit point.*

Figure B2.2 shows an example RVSDG on the left and the CFG recovered by this procedure in the center. The resulting CFG is *always* structured, and is similar to those produced by Johnson [98]. Compared to the original CFG from which the RVSDG was constructed, this may result in a substantial overhead in terms of code size and branch instructions.

Definition 14. *Let* **R** *be an RVSDG, then denote the control flow graph produced by algorithm* StructuredControlFlow *as:*

$$\textsc{Scfr}(\mathbf{R}, \textsc{EvalOrder}, \textsc{VarNames})$$

B2.3.3. Predicative Control Flow Recovery

An alternative approach is to interpret the predicate computations inside the RVSDG to determine control flow: Instead of generating `branch` vertices for γ- and θ-constructs themselves, we generate a `branch` vertex for predicate def nodes, and follow the predicate use nodes to the eventual destination. This requires an RVSDG in *predicate continuation form*, as defined in Section 13. The constraints of this form ensure that there is a topological order such that there is no node between predicate def/use pairs. The algorithm EvalOrder is assumed to always produce such an order, but note that CopyInsertion may

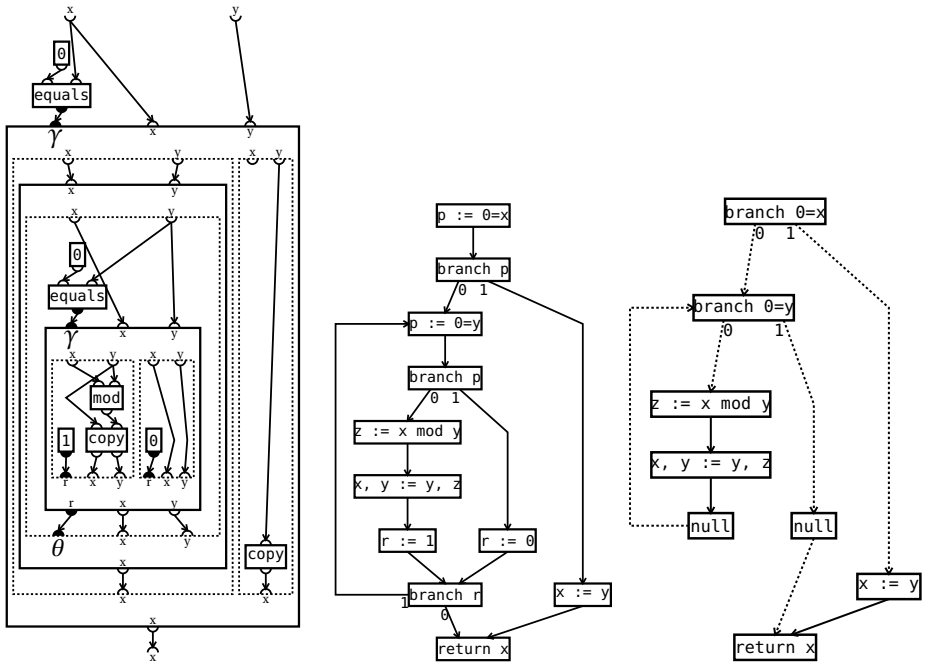


Figure B2.2.: Euclid's algorithm in different representations. Left: Representation of the algorithm as RVSDG. Center: CFG generated from RVSDG on the left by SCFR. Right: CFG recovered from RVSDG on the left by PCFR. The solid arcs are generated in the first pass by PREDICATIVECONTROLFLOWPREPARE, the dotted arcs are generated in the second pass by PREDICATIVECONTROLFLOWFINISH.

insert `copy` nodes afterwards. We introduce *predicative control flow recovery (PCFR)* as the two-pass process consisting of PREDICATIVECONTROLFLOWPREPARE and PREDICATIVECONTROLFLOWFINISH described below. The first pass generates straight-line basic blocks, the second pass is responsible for connecting them. The rightmost diagram of Figure B2.2 illustrates the recovery process and the generated CFG.

The first step is quite similar to STRUCTUREDCONTROLFLOW, but produces a disconnected graph. It lacks all the `branch` vertices introduced in the other algorithm:

Algorithm PREDICATIVECONTROLFLOWPREPARE

Sequentially process all nodes of a given RVSDG according to the chosen topological order. For each node, insert vertices into the CFG according to the following rules. Add an arc to each vertex from the previously generated vertex unless *the previous node in the chosen topological order is a predicate def node:*

1 *For each simple node (including* `copy` *nodes), insert an assignment statement evaluating an expression equivalent to the semantics of the node into the CFG.*

2 *For each γ-node, recursively process subregions. Add the resulting sub-CFGs to the parent CFG* without *adding continuation arcs to the entry points.*

3 *For each θ-node, recursively process the subregion. Add the resulting sub-CFG to the parent CFG* without *adding any continuation arcs either to the entry or from the exit.*

4 *For each simple node that is a predicate definition, enumerate all of its possible predicate output value combinations. In case there is only one, keep track of this predicate constant. Add a* `branch` *statement with a predicate identifying the effective predicate output combination depending on its input values. In case there is only one predicate value combination, i.e. a predicate constant, create a* `null` *statement instead. Keep track of the destination(s).*

Record a mapping of the vertices corresponding to each RVSDG node and region.

This leaves a graph with a number of dangling branch statements, which are then resolved in a secondary pass:

Algorithm PREDICATIVECONTROLFLOWFINISH

Process all predicate-def simple nodes. Identify the vertex created for it in the CFG already as origin. *For each possible predicate value combination, traverse to the eventual destination: Start with the subsequent node in topological order (which must be either the corresponding predicate use node* or *an intervening* `copy` *node) and repeatedly apply the following rules until no predicate value is used any more:*

1. *When reaching the origin vertex, add an arc to itself, making it an infinite loop.*

2. *If n is a predicate constant, traverse to its eventual destination as if the process had originally started here.*

3. *If n is a* `copy` *node, insert a vertex with a corresponding assignment statement, and connect to it from the origin vertex. Treat the inserted vertex as the new origin, and continue tracing from the next node in topological order.*

4. *If n is not a predicate use node, terminate search at the corresponding vertex, and connect to it from the origin vertex.*

5. *If n is a γ- or θ-node, trace to the first node within the correct subregion (or the subsequent node in the parent if the subregion is empty).*

6. *If n is the result of a γ-region, traverse to the subsequent place in the parent region.*

7. *If n is the result of a θ-region, determine the value of the loop repetition predicate. If the loop is to be repeated, traverse to the entry of the region. Otherwise traverse to the subsequent place in the parent region.*

Subsequently, prune all vertices that are not reachable from the entry point and short-circuit all `null` *vertices.*

Note that the above tracing process inserts exactly one linear path per predicate-def node and possible predicate value combination generated by it. Particularly, if there is no `copy` node encountered during tracing, then it inserts exactly one arc. Noting that `copy` nodes are inserted for the purpose of SSA destruction, we observe that the additionally inserted vertices correspond to arc splits in other approaches [29].

Definition 15. *Let* **R** *be an RVSDG, then denote the control flow graph produced by predicative control flow recovery as:*

PCFR(**R**, EVALORDER, VARNAMES)

B2.4. Transforming CFGs to RVSDGs

We observed that Algorithm STRUCTUREDCONTROLFLOW converts an RVSDG into a structured CFG. There is a corresponding direct transformation from any structured CFG to an RVSDG by utilizing the decomposition into regions following Definition 10: After logically contracting all **Branch** and **Loop** subregions into single vertices, we can recursively convert each **Linear** region into an RVSDG as follows:

- Set up a symbol table to map each variable name in the CFG to its definition place in the RVSDG. All initially defined variables will be marked as *parameters*.
- Process all vertices in topological order by the following rules:
  - For an assignment statement, generate a simple node in the RVSDG with an operation equivalent to its right hand side. Update the symbol table.
  - If the vertex represents a **Branch** subregion, take note of the branch predicate. Recursively process each alternative path. Afterwards generate a γ-node that uses the predicate and all variables required in the subregions as input according to the symbol table. Update the symbol table with all variables assigned to in any of the alternate paths to use the new value/state defined by the γ-node.

  - If the vertex represents a **Loop** subregion, recursively process the loop body. Take note of the predicate variable controlling repetition within the symbol table used in processing the subregion and generate a corresponding θ-node containing the generated RVSDG. Use the symbol table to determine the initial values/states of all loop variables as input to the θ-node, and update the symbol table to reflect the state after exiting the loop.

- Generate the results of the RVSDG using the symbol table.

Essentially, we perform a symbolic execution of the procedure represented by the CFG. Note, that the algorithm does *not* assume a CFG in SSA form. This form is automatically established during construction, due to an RVSDG being implicitly in it.

Definition 16. *For a structured CFG* $\overline{\mathbf{C}}$*, let the RVSDG resulting from the algorithm above be designated by:* BUILDRVSDG*($\overline{\mathbf{C}}$).

The remainder of this section describes an algorithm that converts an arbitrary CFG into a structured one. In contrast to other approaches, it avoids cloning any existing nodes in the graph. Instead, it introduces fresh *auxiliary predicate variables* which we name `p`, `q` and `r`. These are used on the left hand side of assignment statements and in branch statements. We refer to statements involving one of these variables as *auxiliary assignments* and *auxiliary branches*.

The algorithm consists of two parts: The first identifies and handles loops, while the second operates on acyclic graphs and only processes branch constructs. We use the following notational convention: Original graphs, arcs and vertices are marked with plain letters such as $\mathbf{C}$, a, v, while transformed graphs and newly inserted arcs and vertices are denoted as $\mathbf{C}^*$, a^*, v^*.

Definition 17. *For any closed CFG* $\mathbf{C}$*, let the structured CFG resulting from the algorithm described below be designated by:* RESTRUCTURECFG($\mathbf{C}$)*. Furthermore, denote by*

$$\text{BUILDRVSDG}(\mathbf{C}) := \text{BUILDRVSDG}^*(\text{RESTRUCTURECFG}(\mathbf{C}))$$

the RVSDG built by combining both steps.

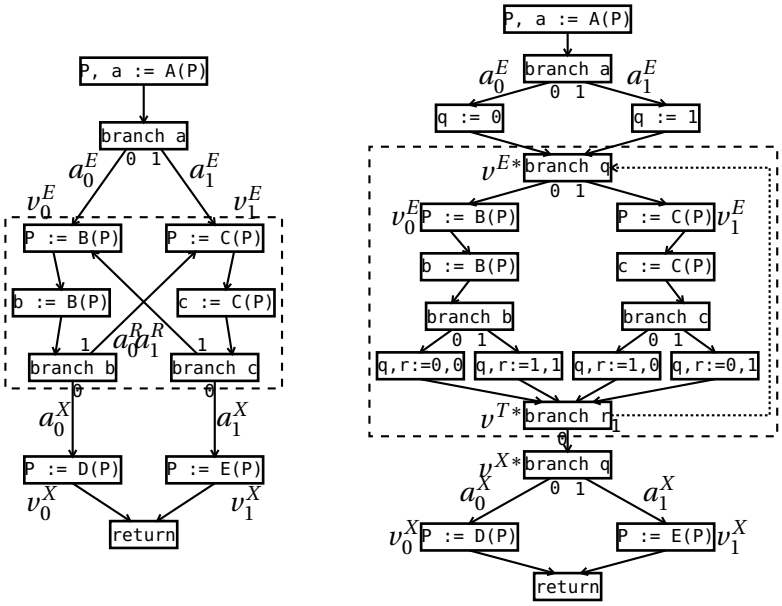


Figure B2.3.: Restructuring of loop control flow. Left: Unstructured control flow graph (loop with two entry and exit paths). The vertices within the dashed box form a strongly connected component. Right: Restructuring the left CFG. The subgraph in the dashed box corresponds to the loop body and is treated as if it were collapsed into a single vertex from the point of view of the outer graph. The dotted arc is the single repetition arc.

B2.4.1. Loop Restructuring

Given a closed CFG **C**, we start by identifying all strongly connected components (SCCs) within **C** using the algorithm described by Tarjan [206]. By necessity, neither the entry nor the exit vertex of **C** are part of any SCC. Each SCC is restructured into a loop with a single head and tail vertex, such that the head vertex is both the single point of entry and starting point for subsequent iterations and the tail vertex is the single vertex controlling repetition and exit from the loop.

For each SCC we identify the following sets of arcs and vertices:

- Entry arcs $A^E = \{a_0^E, a_1^E, \ldots\}$: All arcs from a vertex outside the SCC into the SCC
- Entry vertices $v_0^E, v_1^E, \ldots, v_{m-1}^E$: All vertices which are the target of one or more entry arcs
- Exit arcs $A^X = \{a_0^X, a_1^X, \ldots\}$: All arcs from a vertex inside the SCC pointing out of the SCC
- Exit vertices $v_0^X, v_1^X, \ldots, v_{n-1}^X$: All vertices which are the target of one or more exit arcs
- Repetition arcs $A^R = \{a_0^R, a_1^R, \ldots\}$: All arcs from a vertex inside the SCC to any entry vertex

The left hand side of Figure B2.3 illustrates the arc and vertex sets under consideration.

Unless the loop already meets our structuring requirements, we restructure it by possibly introducing a `branch` statement as single point of entry v^{E*} and single point of exit v^{X*}. They demultiplex to the original entry vertices or exit vertices, respectively. Another `branch` statement is introduced as single control point v^{T*} at the end of the loop. A single *repetition arc* a^{R*} leads from v^{T*} to v^{E*}, and a single *exit arc* a^{X*} leads from v^{T*} to v^{X*}. Two auxiliary predicates q and r are added to facilitate demultiplexing and repetition. The original entry, exit and repetition arcs are all replaced as follows:

- For each entry arc, identify the entry vertex v_k^E it points to. Replace the arc with an assignment `q` $:= k$ that proceeds to v^{E*}, which in turn evaluates k on entry to determine continuation at v_k^E.
- For each repetition arc, identify the entry vertex v_k^E it points to. Replace the arc with an assignment `q, r` $:= k$, `1` and funnel control flow through v^{T*}, which in turn evaluates r to determine whether the loop is to be repeated and subsequently continued at v_k^E.
- For each exit arc, identify the exit vertex v_k^X it points to. Replace the arc with an assignment `q, r` $:= k$, `0` and funnel control flow through v^{T*},

which in turn evaluates r to leave the loop. Subsequently, control flow is demultiplexed by v^{X*} to reach v_k^X.

The result of this process is illustrated on the right hand side of Figure B2.3. The loop body itself is now contained in a subgraph which we denote as $\mathbf{L}^*$. It has exactly one entry vertex and one exit arc from/to the remainder of the whole graph. For all purposes of further processing of $\mathbf{L}^*$, we will treat a^{R*} as absent.

Note that $\mathbf{L}^*$ is not necessarily acyclic: While all repetition arcs A^R were removed, there may still be nested loops within the loop body. Since $\mathbf{L}^*$ (minus a^{R*}) is a closed CFG, the algorithm can be applied recursively on $\mathbf{L}^*$. This produces eventually an evaluation equivalent structured graph $\overline{\mathbf{L}}$, which can be substituted for $\mathbf{L}^*$ in $\mathbf{C}$. We call the result graph after the substitution of all SCCs $\mathbf{C}^*$. For further processing of $\mathbf{C}^*$, we treat all $\mathbf{L}^*$ subgraphs as if each were a single vertex.

Under this interpretation, $\mathbf{C}^*$ is now acyclic and we can apply the algorithm described in Section B2.4.2 to eventually produce a structured CFG $\overline{\mathbf{C}}$. Note, that for the purpose of constructing an RVSDG, there is no need to actually create $\overline{\mathbf{C}}$ or even $\mathbf{C}^*$. Instead, we can recursively build for each $\mathbf{L}^*$ subordinate RVSDGs and then wrap them individually into a θ-node.

B2.4.2. Branch Restructuring

Given an acyclic closed CFG $\mathbf{C}$, we partition the graph into a linear *head* subgraph $\mathbf{H}$, multiple *branch* subgraphs $\mathbf{B}_k$ and a *tail* subgraph $\mathbf{T}$. Iff $\mathbf{C}$ is linear, the partitioning results in zero branch subgraphs and an empty subgraph $\mathbf{T}$. In this case $\mathbf{C}$ is already structured and no further steps are necessary. The branch and tail subgraphs are restructured to closed CFGs, resulting in branch subgraphs with exactly one entry arc from $\mathbf{H}$ and one exit arc to the restructured tail subgraph $\mathbf{T}^*$. The algorithm is then applied recursively to each branch and tail subgraph until we eventually obtain a structured graph $\overline{\mathbf{C}}$.

During partitioning, we first identify $\mathbf{H}$ by searching for the longest linear subgraph from the entry vertex. The last vertex of $\mathbf{H}$ must be a `branch` statement with m outgoing *fan-out arcs* $a_0^F, a_1^F, \ldots, a_{m-1}^F$. We initially identify the branch subgraph $\mathbf{B}_j$ as the dominator graphs of the arc a_j^F. As explained later, we

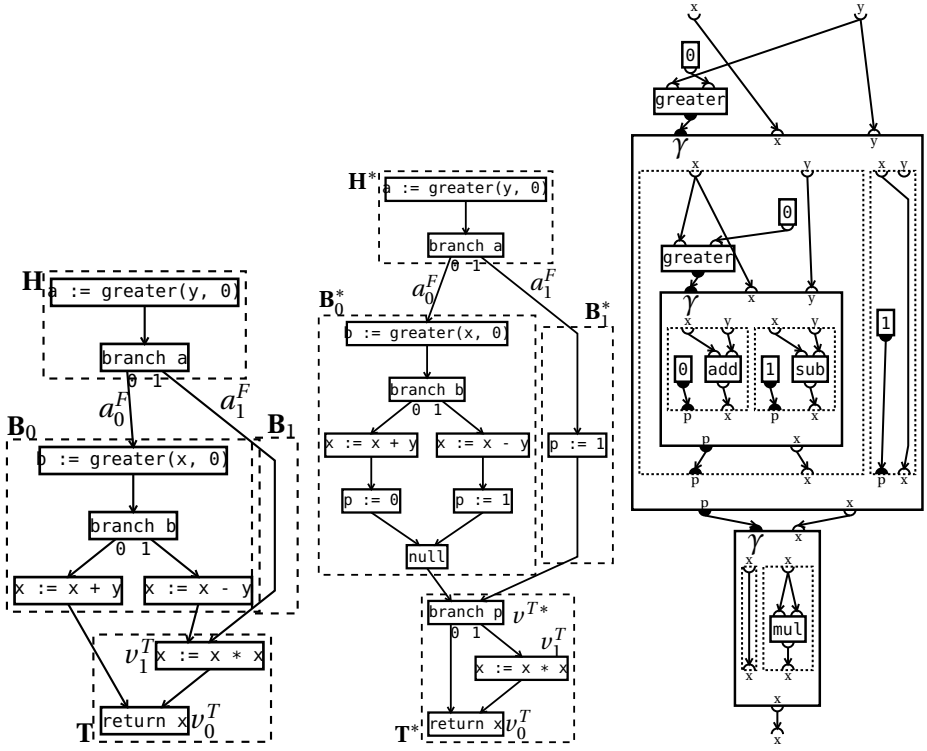


Figure B2.4.: Restructuring of branch control flow. Left: Unstructured control flow graph (inconsistent selection paths). Middle: Restructuring the left CFG. Right: The RVSDG equivalent to the CFGs to the left.

may have to trim the sets of vertices covered by each $\mathbf{B}_j$ slightly (we denote the "pure" dominator subgraphs without trimming as $\mathbf{B}'_j$). Note that some branch subgraphs might also be empty, but we nevertheless keep track of the defining arc a_j^F. The remainder of **C** forms then the tail subgraph **T**. The left diagram of Figure B2.4 illustrates the partitioning.

Let $v_0^T, v_1^T, \ldots, v_{n-1}^T$ be the *continuation points* in the tail subgraph: These are the vertices within **T** with at least one arc from either one of the branch subgraphs, or one of the fan-out arcs $a_0^F, a_1^F, \ldots, a_{m-1}^F$ pointing to them. There must be at least one such continuation point and if there is exactly one, **T** has already the desired structure.

If there are multiple continuation points, then restructure **T** and all $\mathbf{B}_j$ as follows:

- Insert a `branchp` statement as v^{T*} into **T** that demultiplexes to the original continuation points $v_0^T, v_1^T, \ldots, v_{n-1}^T$
- Substitute each arc from any $\mathbf{B}_j$ to every v_k^T with an assignment `p` $:= k$, and funnel control flow through v^{T*}
- If some $\mathbf{B}_j$ is empty, the fan-out arc a_j^F must point to some v_k^T. Substitute this arc with a `p` $:= k$ statement and replace the previously empty branch subgraph with this single vertex.
- If any branch subgraph has more than one exit arc to the tail subgraph, funnel all paths through a single `null` statement.

If any of the inserted `p` $:= \ldots$ statements immediately follows a `q` $:= \ldots$ statement, we fuse these two into a single vertex. This results in a new graph $\mathbf{C}^*$ with subgraphs $\mathbf{H}^*$, $\mathbf{B}_j^*$ and $\mathbf{T}^*$ corresponding to the head, branch and tail subgraphs and $\mathbf{H}^*$ being structurally identical to **H**. The right diagram of Figure B2.4 illustrates the result of this process as well as the notation. The new branch and tail subgraphs are closed, recursively applying the process to them yields structured graphs $\overline{\mathbf{B}}_0, \overline{\mathbf{B}}_1, \ldots \overline{\mathbf{B}}_{m-1}$ and $\overline{\mathbf{T}}$ (in case $\mathbf{B}_i^*$ is empty, set $\overline{\mathbf{B}}_1 := \mathbf{B}_i^*$). These subgraphs can then be substituted in $\mathbf{C}^*$ for the original branch and tail subgraphs, resulting in a structured graph $\overline{\mathbf{C}}$.

As shown in Section B2.4.1, it is not necessary to build the structured CFG $\overline{\mathbf{C}}$ for the purpose of RVSDG construction. Instead, the algorithm recursively builds

RVSDGs. The transformed branch subgraphs are contained within a γ-node, whereas the transformed head and tail subgraphs precede and succeed the γ-node, respectively. This is illustrated in Figure B2.4.

Trimming of Branch Subgraphs

The algorithm above always yields a graph with the desired structure, but may interleave the defs and uses of different auxiliary predicates improperly. This results in an RVSDG that is not in predicate continuation form and we therefore need to trim the vertex sets of the branch subgraphs as follows:

- Determine from the set of continuation points the subset of vertices that are immediate successor of an auxiliary assignment statement.
- For each such continuation point, pull its immediate predecessor from the branch subgraphs into the tail subgraph *unless all* immediate predecessors are in the branch subgraphs.

Figure B2.5 shows a CFG where trimming leads to eviction of a vertex from one of the branch subgraphs and illustrates its effect: It prevents improper interleaving of auxiliary assignments and branches by ensuring that either *all* defs of a predicate are at the end of any branch subgraph or *none* are. The only possible interaction is an assignment/branch on p nested properly within an assignment/branch on q. Note that according to the rules above, the assignments are fused into a single vertex and translated as a single predicate defining node in the RVSDG.

B2.5. Proof of Correctness and Invertibility

In this section, we prove correctness of the presented algorithms and that the proposed RVSDG construction and destruction algorithms are mutually inverse to each other.

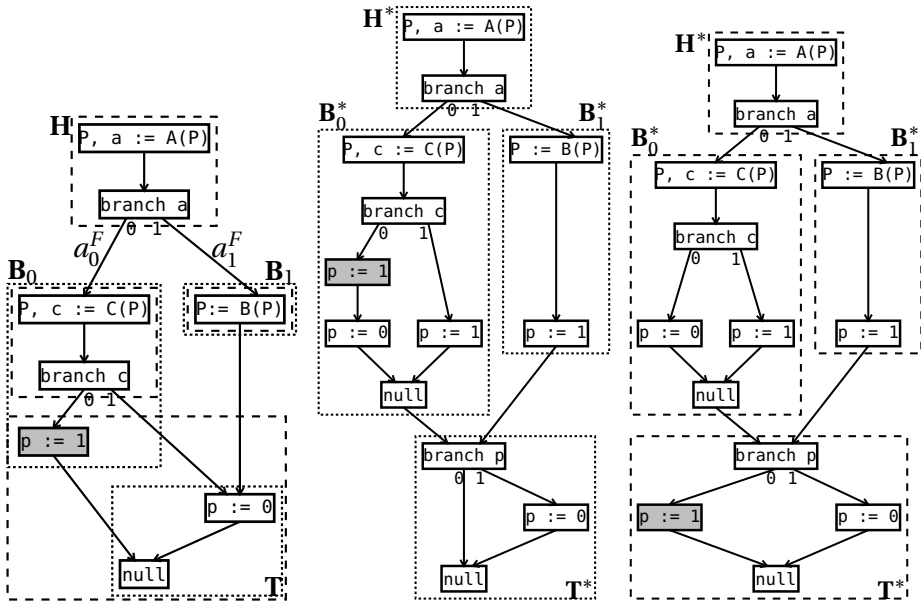


Figure B2.5.: Left: A CFG where the last `null` statement is successor to auxiliary assignment statement `p := 0` and `p := 1`. Only one of these is dominated by any of the fan-out arcs a_0^F and a_1^F and is therefore a *critical assignment*. The dotted boxes illustrate the subgraphs without any trimming, the dashed boxes show the effect of trimming. The critical assignment is shaded. Middle: Effect of applying branch restructuring without trimming. The improper interleaving of operations leads to the critical assignment being "lost". Right: Applying branch restructuring after trimming. No vertex is inserted on a path from the critical assignment to the exit vertex.

B2.5.1. Termination

Definition 18. *A CFG is said to be* pqr-complete *iff: If one predecessor of a vertex is an* auxiliary assignment *statement, then all of its predecessors are.*

Theorem B2.5.1. *For a given acyclic, closed, pqr-complete and non-linear CFG* $\mathbf{C}$*, the partitioning step in Section B2.4.2 yields graphs* $\mathbf{T}, \mathbf{B}_0, \mathbf{B}_1, \ldots, \mathbf{H}$ *such that* $\mathbf{H}, \mathbf{T}$ *and at least one* $\mathbf{B}_i$ *is non-empty.*

Proof. $\mathbf{C}$ is closed, let v^E and v^X be the entry and exit vertices. By construction it is evident that $v^E \in \mathbf{H}$ and $v^X \in \mathbf{T}$, so both are non-empty. The only remaining proof obligation is to show that one $\mathbf{B}_i$ is non-empty. We proceed with a few auxiliary propositions, then show that some $\mathbf{B}'_i$ (dominator graph before trimming) is non-empty, then show the same for some $\mathbf{B}_i$ (dominator graph after trimming).

Let v^B be the uniquely determined last `branch` vertex of $\mathbf{H}$. For each vertex, let $d(v)$ be the depth (length of longest path from entry) of vertex v. Since $\mathbf{H}$ is linear with only v^B having immediate successors outside of $\mathbf{H}$ by construction, it must hold $d(v) < |\mathbf{H}|$ for all $v \in \mathbf{H}$, $d(v) \geq |\mathbf{H}|$ for all $v \notin \mathbf{H}$ and $d(v) < d(v^X)$ for all $v \neq v^X$. Furthermore, if $d(v) \geq |\mathbf{H}|$ for some vertex v, then for any of its immediate predecessors v' it must be that either $d(v') \geq |\mathbf{H}|$ or $v' = v^B$.

We first show that for any vertex v with $d(v) = |\mathbf{H}|$ that $v \in \mathbf{B}'_i$ for some i: v cannot be in $\mathbf{H}$, but all of its predecessors must be. Since $\mathbf{H}$ is linear with only the fan-out arcs $a^F_0, a^F_1, \ldots, a^F_{m-1}$ of its last `branch` vertex pointing to any vertex outside of $\mathbf{H}$, this `branch` vertex must be the single predecessor of v. Since there can be at most one arc between any two vertices, v must be dominated by some arc a^F_i, hence $v \in \mathbf{B}'_i$.

There must be at least one arc v^R with $d(v^R) = |\mathbf{H}|$ because $d(v^X) > |\mathbf{H}|$, therefore at least one $\mathbf{B}'_i$ must be non-empty. Since $v^X \notin \mathbf{B}'_j$ for any j, this in turn means that there must be a vertex $d(v^S) = |\mathbf{H}| + 1$.

Let P be the set of immediate predecessors of v^S. For each $v \in P$ it is either $d(v) = d(v^S) - 1 = |\mathbf{H}|$ or $v = v^B$, and there must be at least one $v^P \in P$ such that $d(v^P) = |\mathbf{H}|$. If none of the elements of P are auxiliary assignments, then $v^P \in \mathbf{B}_i$ since it is not subject to trimming, completing our proof. Otherwise by pqr-completeness all elements of P must be auxiliary assignment vertices. This

means that $v^B \notin P$ (since v^B is a `branch` statement), and therefore $d(v) = |\mathbf{H}|$ for all $v \in P$. This means that each $v \in P$ is also in some dominator subgraph $\mathbf{B}'_i$. This in turn means that they are not removed by trimming and therefore $v^P \in \mathbf{B}_i$ for some i as well. □

Theorem B2.5.2. *The branch restructuring algorithm given in Section B2.4.2 always terminates for a given acyclic, closed and pqr-complete CFG* **C**.

Proof. We prove this by showing that there is a strictly decreasing well-ordered metric for the arguments of the recursive function calls, and that the arguments to these calls satisfy the preconditions of this theorem.

For any **C** let $b(\mathbf{C})$ denote the number of `branch` vertices, and $|\mathbf{C}|$ denote the number of vertices. Define $m(\mathbf{C}) := (b(\mathbf{C}), |\mathbf{C}|)$ with the well-ordering relation

$$m(\mathbf{C}) < m(\mathbf{C}') \leftrightarrow b(\mathbf{C}) < b(\mathbf{C}') \vee (b(\mathbf{C}) = b(\mathbf{C}') \wedge |\mathbf{C}| < |\mathbf{C}'|)$$

and will show that $m(\mathbf{T}^*) < m(\mathbf{C})$ and $m(\mathbf{B}^*_i) < m(\mathbf{C})$.

If $b(\mathbf{C}) = 0$, then the graph is linear and will be returned unchanged without recursion. If $b(\mathbf{C}) \neq 0$ we know by Theorem B2.5.1, that **H**, **T** and at least one $\mathbf{B}_i$ contain at least one vertex of **C** each, and **H** contains a `branch` vertex. This means that $|\mathbf{T}| \leq |\mathbf{C}| - 2$, $b(\mathbf{T}) \leq b(\mathbf{C}) - 1$, since we insert at most one `branch` vertex into the tail subgraph, it follows: $|\mathbf{T}^*| \leq |\mathbf{C}| - 1 \wedge b(\mathbf{T}^*) \leq b(\mathbf{C}) - 1$. We only insert assignment and `null` statements into the branch subgraphs and therefore $b(\mathbf{B}^*_j) \leq b(\mathbf{C}) - 1$.

The head, tail and branch subgraphs of **C** are initially pqr-complete. The only possible insertion into the tail subgraph is one `branch` vertex and into the branch subgraphs are auxiliary assignment statements that fan in to a `null` statement. All of these insertions preserve the initial pqr-complete property. □

Theorem B2.5.3. *The loop restructuring algorithm given in Section B2.4.1 terminates for any given closed pqr-complete CFG* **C**.

Proof. As above, we give a well-ordered metric for the arguments of recursive calls. Let $s(\mathbf{C}) := \sum_{S \in \mathrm{SCC}(\mathbf{C})} |S|$ be the sum of all vertices in any non-trivial strongly connected component. By necessity, $s(\mathbf{C}) \leq |\mathbf{C}| - 2$, since the entry

and exit vertex cannot be in any SCC. Furthermore, the total size is also an upper bound for the size of each SCC individually.

$\mathbf{L}^*$ consists of all vertices of one SCC plus at most two auxiliary vertices. Neither the newly added auxiliary entry and exit vertices can be part of any SCC, nor can the original entry vertices as we remove all repetition arcs. Therefore, $s(\mathbf{L}^*) \leq |\mathbf{L}^*| - 2 - 1 \leq s(\mathbf{C}) - 1$.

The introduced auxiliary assignment statements amount to assignments to `q` which fan into the entry of the loop, and assignments to `q`, `r` which fan to the tail vertex of the loop. Consequently, all graphs processed either by recursion or branch restructuring share this property. By assumption, the auxiliary predicates `p`, `q` and `r` are not used anywhere in any original CFG on which restructuring operates, concluding the proof that the algorithm terminates. □

The theorems above show that the algorithms are structurally recursive and we can prove properties about them inductively.

B2.5.2. CFG Restructuring Correctness and Evaluation Equivalence

Theorem B2.5.4. *For any closed CFG* $\mathbf{C}$*, the restructuring algorithm of Section B2.4 yields a structured CFG* $\overline{\mathbf{C}}$*.*

Proof. We prove by induction over the recursive call tree, first for branch restructuring as per Section B2.4.2: All linear graphs at the call tree leafs are structured by rule 1 of Definition 10. $\mathbf{H}^*$ is linear, by induction hypothesis all of $\overline{\mathbf{B}}_i$ and $\overline{\mathbf{T}}$ are either structured or empty. Contracting all of these subgraphs leaves a graph shaped according to rule 2 which is therefore structured as well.

Loop restructuring as per Section B2.4.1 calls into branch restructuring at its leafs, and by induction hypothesis we can again presume $\overline{\mathbf{L}}$ to be structured. Rule 3 allows removing the repetition arc after contracting the loop body. By applying branch restructuring while treating all loops as indivisible vertices we therefore arrive at a structured CFG again. □

Theorem B2.5.5. *For each arc from v_1 and v_2 in* **C***, there is either a corresponding arc in* $\overline{\mathbf{C}}$ *or a path from v_1 to v_2 that consists entirely of matching* auxiliary assignment *and* auxiliary branch *vertices as well as* null *statements such that control must flow from v_1 to v_2.*

Proof. Evidently, each individual insertion of auxiliary statements during the restructuring process already satisfies the proposition. We only need to show that there is no interaction which disturbs the proper matching of assignment and `branch` statements. The only places where auxiliary vertices can be inserted improperly is during branch restructuring between subgraphs $\mathbf{B}_i$ and **T**. Due to pqr-completeness, the trimming step ensures that all insertions occur in a uniformly nested fashion within other existing auxiliary assignment and `branch` statements. Thus, we only need to prove that there is no path with assignments to `p` without an intervening `branchp` statement. We assert that auxiliary statements only occur in CFGs processed recursively as:

1. `branchp` as the entry vertex of a graph, or
2. `p :=` ... as the exit vertex of a graph, or
3. `p :=` ... as predecessors to a `null` exit vertex

In the first two cases there is nothing to prove since no vertices are inserted before entry or after the exit vertex. In the third case, the trimming step ensures that either all or none of the `p :=` ... statements is contained in the branch subgraphs. In the first case, there is nothing to prove since there are no insertions between vertices in the tail subgraph. In the second case, pqr-completeness ensures that the tail subgraph consists of a single `null` vertex and that there is consequently no auxiliary assignment or branch statement inserted. □

The last theorem not only proves the evaluation equivalence of the original and restructured CFG, but also shows that there is a simple mechanical procedure for recovering the original CFG. The following algorithm is a very restricted form of constant propagation that only considers constant assignments to any of the auxiliary variables:

Algorithm SHORTCIRCUITCFG

Repeatedly apply these transformations:

- *Replace every arc to a* `null` *vertex with an arc to its successor, and remove the vertex from the graph.*
- *Let* $x_1, x_2, \ldots$ *be names of variables and* $expr_1, expr_2, \ldots$ *be expressions for a statement of the form* $x_1, x_2, \ldots := expr_1, expr_2, \ldots$ *followed by a* `branch` x_i *vertex, then substitute* x_i *for* $expr_i$ *in the* `branch` *statement and remove* x_i *and* $expr_i$ *from the assignment statement. If no variables and expressions remain, replace every arc to it with an arc to its successor and remove the assignment statement from the graph.*
- *Let* c *be a constant expression, then replace a* `branch` c *statement with a* `null` *statement with an outgoing arc to the destination of the branch corresponding to* c.

Corollary B2.5.1. *For any closed CFG* $\mathbf{C}$ *the following is required to hold:*

$$\mathbf{C} = \text{SHORTCIRCUITCFG}(\text{RESTRUCTURECFG}(\mathbf{C}))$$

B2.5.3. CFG Reconstruction by PCFR

Theorem B2.5.6. *For any given structured CFG* $\overline{\mathbf{C}}$ *there are oracles for* EVALORDER *and* VARNAMES *such that*

1. VARNAMES *succeeds without introducing* `copy` *operations, and*
2. *the following equation holds:*

$$\text{SCFR}(\text{BUILDRVSDG}^*(\overline{\mathbf{C}}), \text{EVALORDER}, \text{VARNAMES}) = \overline{\mathbf{C}}$$

Proof. Let $\mathbf{R} := \text{BUILDRVSDG}^*(\overline{\mathbf{C}})$. The nodes of each region of $\mathbf{R}$ were constructed by traversing the corresponding part of $\overline{\mathbf{C}}$ in control flow order, and recording just the value and state dependencies of operations. Therefore, a topological order of RVSDG nodes corresponding to the original control flow order exists and can be supplied by an oracle for EVALORDER.

During RVSDG construction, we kept updating a symbol table maintaining the mapping from variable name in the original CFG and output of a node in the RVSDG. Inverting this mapping means that we can recover the assignment of def sites in the RVSDG to names in the original CFG. This yields an

interference-free coloring, showing that VARNAMES can succeed without insertion of `copy` operations and recover the original CFG names.

Together this shows that SCFR perfectly reconstructs $\overline{\mathbf{C}}$. □

Theorem B2.5.7. *For an RVSDG* $\mathbf{R}$ *in predicate continuation form and algorithms* EVALORDER *and* VARNAMES *such that* VARNAMES *succeeds without introduction of* `copy` *nodes, the following identity is required to hold:*

$$\begin{aligned}&\textsc{Pcfr}(\mathbf{R}, \textsc{EvalOrder}, \textsc{VarNames})\\ =\ &\textsc{ShortCircuitCfg}(\textsc{Scfr}(\mathbf{R}, \textsc{EvalOrder}, \textsc{VarNames}))\end{aligned}$$

Proof. Let $\overline{\mathbf{C}} := \textsc{Scfr}(\mathbf{R}, \textsc{EvalOrder}, \textsc{VarNames})$. By assumption, the chosen topological order keeps predicate def and use nodes adjacent, and also VARNAMES does not insert nodes. This ensures that $\overline{\mathbf{C}}$ in turn has assignments to predicate variables immediately succeeded by any `branch` statement(s) using the value – with possibly some intervening `null` statements which are eliminated as the first step of SHORTCIRCUITCFG.

PREDICATIVECONTROLFLOWPREPARE produces the same set of vertices as SCFR, except that predicate defining RVSDG nodes are translated as `branch/null` instead of assignment statements and that no `branch` vertices are generated at the head or tail of γ- or θ-constructs, respectively (see Figure B2.2 for an illustration).

PREDICATIVECONTROLFLOWFINISH evaluates the same expressions for determining the continuation points as SHORTCIRCUITCFG does during `branch` replacement steps, combined with the structural correspondence this shows that both arrive at the same CFG. □

With these preparations, we can formulate and prove the main theorem of our paper:

Theorem B2.5.8. *For each closed CFG* $\mathbf{C}$ *there exist oracles* EVALORDER *and* VARNAMES *such that:*

$$\mathbf{C} = \textsc{Pcfr}(\textsc{BuildRvsdg}(\mathbf{C}), \textsc{EvalOrder}, \textsc{VarNames})$$

.

Proof. Combining the theorems above, we know there exist oracles for EVALORDER and VARNAMES such that we can rewrite:

$$\begin{aligned}
\mathbf{C} &= \textsc{Pcfr}(\textsc{BuildRvsdg}(\mathbf{C}), \textsc{EvalOrder}, \textsc{VarNames}) \\
&= \textsc{ShortCircuitCfg}(\textsc{Scfr}(\textsc{BuildRvsdg}^*(\textsc{RestructureCfg}(\mathbf{C})), \\
&\quad \textsc{EvalOrder}, \textsc{VarNames})) \\
&= \textsc{ShortCircuitCfg}(\textsc{RestructureCfg}(\mathbf{C})) \\
&= \mathbf{C}
\end{aligned}$$

□

The next theorem shows that the degrees of freedom in undoing SSA has no material influence on the control flow structure:

Theorem B2.5.9. *For each closed CFG* **C** *there exists an oracle* EVALORDER *such that for any* VARNAMES *and* $\mathbf{C}' = \textsc{Pcfr}(\textsc{BuildRvsdg}(\mathbf{C}), \textsc{EvalOrder}, \textsc{VarNames})$ *it holds that* $\mathbf{C}'$ *and* **C** *are structurally equivalent (see definition 11).*

Proof. Using the same oracle for EVALORDER as in the previous theorem, we observe that PCFR processes all RVSDG nodes in the same sequence in both cases. This leads to the same order of CFG nodes, but $\mathbf{C}'$ may differ from **C** in two aspects: It may use different variable names and contain several variable copy operations (cf. Section B2.3.1) that are not present in **C**. This still leads to the same structure multigraph and unchanged evaluation semantics, hence the two CFGs are structurally equivalent. □

Exact reconstruction of the original control flow depends on selecting a specific topological ordering of nodes per region. An RVSDG where topological ordering is sufficiently constrained will always faithfully reproduce the original control flow structure. Otherwise, different orders may result in wildly different CFG shapes. Still, we retain as an invariant that the number of branching points does not increase which shows that our destruction algorithms never deteriorates control flow as measured by static branch counts:

Theorem B2.5.10. *For each closed CFG* **C**, *any* EVALORDER, *any* VARNAMES *and* $\mathbf{C}' = \textsc{Pcfr}(\textsc{BuildRvsdg}(\mathbf{C}), \textsc{EvalOrder}, \textsc{VarNames})$ *it holds that* $\mathbf{C}'$ *have the same number of* `branch` *statements.*

Proof. The number of branches generated by PCFR only depends on the number of predicate assignments within the RVSDG, and the number of auxiliary predicate assignments “skipped" by algorithm PREDICATIVECONTROLFLOWFINISH. The former is evidently independent of algorithm choices for EVALORDER and VARNAMES. The independence of the latter is a consequence of EVALORDER keeping predicate defs and uses adjacent. □

B2.6. Empirical Evaluation

This section describes the results of applying PCFR and SCFR to CFGs extracted from the SPEC2006 benchmark suite [82], in order to evaluate practical implications for the number of generated branches, IR size, and compile-time overhead.

B2.6.1. CFG Extraction and Representation

CFGs were extracted from all C benchmarks in the SPEC2006 using *libclang* [10], which resulted in a set of 14321 CFGs. We ensured that all were closed, by eliminating statically unreachable basic blocks. All operations were modeled as having side effects, consuming a single program state, and producing one for the next operation. This was done to ensure a unique topological order for reconstructing the exact CFG.

CFGs were classified as *Linear* per Definition 8, *Structured* per Definition 10, *Reducible* or *Irreducible*. Structured graphs were identified by structural analysis [183], and irreducibility was determined by T1/T2 analysis [3]. Figure B2.6 shows distributions of each class per program. The CFGs were converted to RVSDGs as described in Section B2.4, and restored to CFGs using PCFR and SCFR. All PCFR results were verified to equal their original CFGs.

[10] http://clang.llvm.org/

	bzip2	cactusADM	calculix	gcc	gobmk	gromacs	h264ref	hmmer	lbm	libquantum	mcf	milc	perlbench	sjeng	sphinx3	wrf
Linear	25	632	9	1353	1552	334	81	71	2	26	1	78	329	28	60	80
Structured	34	429	534	1362	616	238	184	124	5	17	2	42	517	43	63	180
Reducible	41	288	577	1582	474	626	301	329	12	58	21	115	394	73	190	175
Irreducible	2	-	-	2	-	-	-	-	-	-	-	-	10	-	-	-
Total	102	1349	1120	4299	2642	1198	566	524	19	101	24	235	1250	144	313	435

Figure B2.6.: Classification of Extracted CFGs

	bzip2	cactusADM	calculix	gcc	gobmk	gromacs	h264ref	hmmer	lbm	libquantum	mcf	milc	perlbench	sjeng	sphinx3	wrf
SCFR	1440	5802	9373	35827	9679	5486	6136	3766	56	305	190	1160	9374	2236	1543	1284
PCFR	1100	5127	8752	31960	8329	5277	5867	3343	55	280	160	1109	8248	1913	1436	1135
Savings %	**30.9**	**13.2**	**7.1**	**12.1**	**16.2**	**4.0**	**4.6**	**12.7**	**1.8**	**8.9**	**18.8**	**4.6**	**13.7**	**16.9**	**7.5**	**13.1**

Figure B2.7.: Relative Static Branch Counts of PCFR to SCFR

B2.6.2. Key Observations

PCFR removes every auxiliary assignment and `branch` statement introduced by CFG restructuring, avoiding any code size increases from node splitting or additional branches to preserve program logic. Figure B2.7 shows its improvement over SCFR, in terms of `branch` statements. Static branch counts are not directly proportional to program run time, but we regard them as indicative of overheads incurred by other methods [95, 217, 198]. We note that graphs generated by SCFR greatly resemble results produced by Johnson's algorithm [98], suggesting that their overheads are comparable.

RVSDG construction as per Section B2.4 can add constructs with no equivalent in the original CFG, depending on the complexity of the original control flow. Consequently, our RVSDG form carries an expected representational overhead. This is quantified in Figure B2.8, which relates the number of CFG

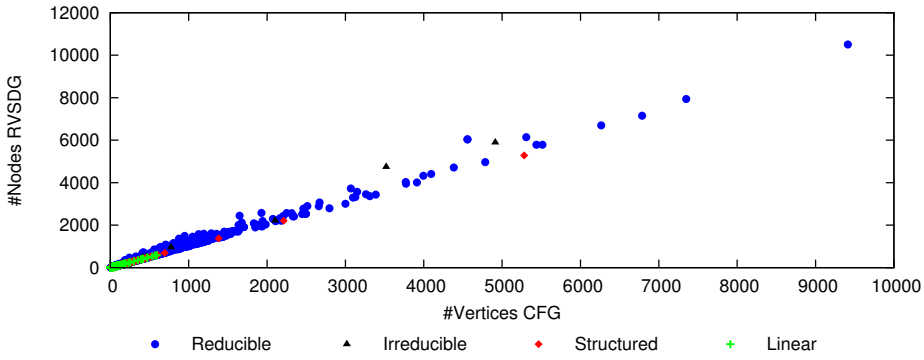


Figure B2.8.: Corresponding CFG and RSVDG Sizes

vertices to RVSDG nodes for each of our CFGs. There is a clear linear relationship for all of the 14321 cases, suggesting that our construction algorithm is practically feasible in terms of space requirements.

Construction and destruction were timed for each CFG, to assess how compile-time overhead grows with graph size. Figure B2.9 shows timings of RVSDG construction versus input size. For the majority of CFGs a linear dependency is recognizable, with a few notable deviations from the expected line. Deviations below the line are either structured or nearly structured graphs. For these graphs, restructuring only discovers hierarchical structure without compounding it, resulting in lower processing time. The 7 deviations above the line are either produced by large `if/then/else` `if` statements, or complex control flow of interacting loops, `switch` and `goto` statements, which produced graphs with hundreds of cascaded equality tests on one side. Through branch restructuring, such imbalanced graphs lead to mostly empty branch subgraphs, and one full. The dominator graph is repeatedly recomputed for the full branch subgraph, and hence, for most vertices in the CFG. This implies a theoretical worst-case quadratic complexity for such cases.

Figure B2.10 shows timings of RVSDG destruction via PCFR versus input size. A linear tendency is visible also here, with a few deviations below and above the line. The deviations below the line are due to graphs with a high degree of linearity. In comparison to the number of total nodes, few predicate def nodes need to be traced to their corresponding use nodes, which results in

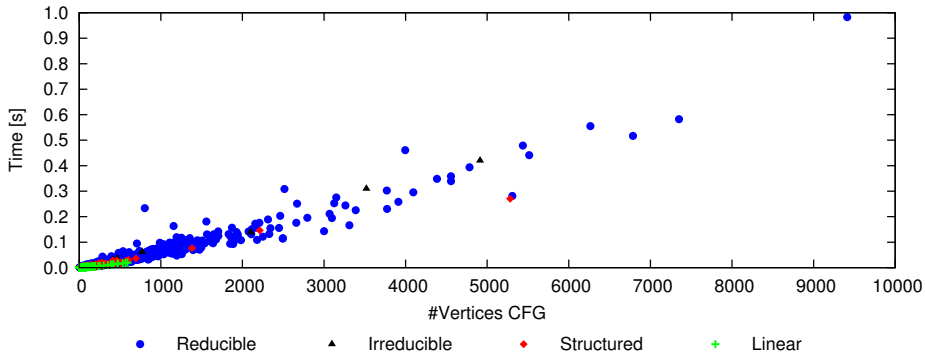


Figure B2.9.: Execution Times for CFG to RVSDG Conversions

low processing time. The few points above the line are again due to extremely unstructured and complex control flow. Considering that PREDICATIVECONTROLFLOWFINISH repeatedly traces through nested RVSDG regions for similar use/def chains starting at different nesting depths, we expect its computational complexity to be worst-case quadratic in terms of input size. Experiments with different compilers suggest that these also exhibit non-linear processing time for similarly complex control flow.

B2.6.3. Summary

The overall conclusion of the empirical study is that PCFR permits CFGs to be reconstructed from their RVSDG representation without producing any control flow overhead for actual benchmark programs. In addition, the IR size and compile-time processing overheads are predominantly linear in terms of input size. We therefore conclude that PCFR enables an RSVDG representation to precisely capture a CFG structure without substantial disadvantages.

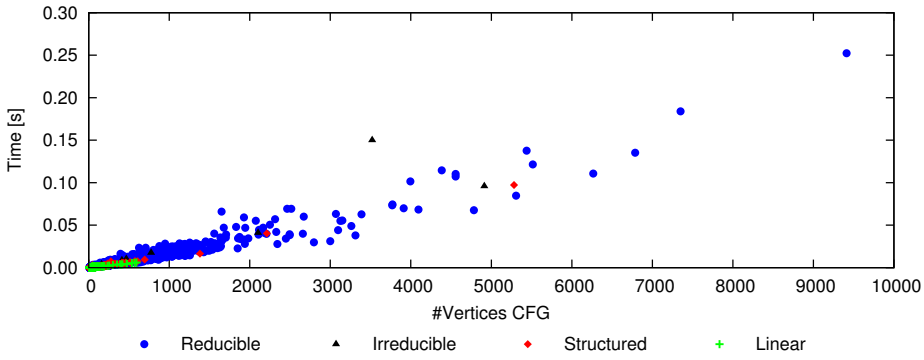


Figure B2.10.: Execution Times for RVSDG to CFG Conversions

B2.7. Related Work

B2.7.1. Intermediate Representations

Many different IRs emerged for simplifying program analysis and optimizations [199]. The RVSDG is related to SSA [52], gated SSA [211] and thinned gated SSA [80]. It shares the same γ function with (thinned) gated SSA and its θ construct is closely related to their μ and η functions. However, all three of these SSA variants are bound to the CFG as an underlying representation. Many optimizations such as constant propagation [223] or value numbering [172] rely on the SSA form to achieve improved results or an efficient implementation.

The Program Dependence Graph (PDG) [71] is a directed graph incorporating control and data flow in one representation. It has statement, predicate expression, and region nodes, and edges represent data or control dependencies. Control flow is explicitly represented through a control dependence subgraph, resulting in cycles in the presence of loops. The PDG has been used for generating vectorized code [17] and partition programs for parallel execution [178].

The Program Dependence Web (PDW) [147] combines gated SSA with the PDG and is suitable for the development of different architecture back-ends. It

allows a direct interpretation under three different execution models: control-, data-, or demand-driven. However, its construction requires 5 passes over the PDG resulting in a time complexity of $O(N^3)$ in the size of the program.

Click's IR [46] is a PDG variant with a Petri net based model of execution. Nodes represent operations, and edges divide into control and data dependence subsets. It was designed for simplicity and speed of compilation, and a modified version is used in the Java HotSpot server compiler [150].

The Value Dependence Graph (VDG) [224] is a directed graph where nodes represent operations and edges dependencies between them. It uses γ-nodes to represent selection, but lacks a special construct for loops. Instead they are expressed as tail-recursive functions with λ-nodes. The significant problem with the VDG is that it fails to preserve the termination properties of a procedure.

B2.7.2. Control Flow Restructuring

Many control flow restructuring algorithms have been proposed to facilitate optimizations. Most work focuses on removing goto statements, or irreducible control flow.

Erosa *et al.* [68] eliminate goto statements in C programs by applying AST transformations. The algorithm works on the language-dependent AST, but is also applicable to other languages [39]. However, it replicates code, and the simple elimination of gotos is insufficient for RVSDG construction.

Zhang *et al.* [238, 237] create single-entry/-exit subgraphs within CFGs. As with Erosa's work, the method suffers from code expansion and produces CFGs not suitable for RVSDG construction.

The algorithm presented by Ammarguellat [8] operates on its own input language, but is closest to our approach in terms of produced control flow. The paper presents an algebraic framework for normalizing control flow to only three structures: single-entry/-exit loops, conditionals and assignments [25]. The algorithm represents a procedure as a set of continuation equations and solves this system with a Gaussian elimination-like method. In contrast to Erosa's and Zhang's work, code is only duplicated for irreducible graphs.

Janssen *et al.* [95] and Unger *et al.* [217] use controlled node splitting to transform irreducible CFGs to reducible ones. Although irreducible control flow is extremely rare in practice [198] and their results are encouraging in terms of code bloat, it is proven that node splitting results in exponential blowup for particular CFGs [37].

B2.8. Conclusion

In this paper, we presented algorithms for constructing RVSDGs from CFGs and vice versa. We proved the correctness of these algorithms, and that the exact, original control flow can be recovered after round-trip conversion. Empirically, we found that our algorithms successfully process CFGs extracted from selected SPEC2006 benchmarks. Processing time and output size correlate linearly with input size, with acceptable deviations for less than 0.01% of test cases. While pathological inputs may cause worse asymptotic behavior, we conclude that the algorithms are applicable to practical programs.

Our main result is that our algorithms obtain control flows which are not limited by any structural properties of the RVSDG. Expressing control flow transformations as predicate computations permits related optimizations to be moved entirely into the RVSDG domain. This provides a means to translate optimizations which depend on established control flow, such as Johnson and Mycroft's VSDG-based code motion and register allocation scheme [97], and may remove the need to perform control flow transformations in any other representation. Ultimately, our work enables a compiler to use a demand-dependence representation up until object code generation, when deciding on a specific flow of control is inevitable.

Part C.

GPU Divergence

C1. Efficient Control Flow Restructuring for GPUs

Nico Reissmann, Thomas L. Falch, Benjamin A. Björnseth, Helge Bahmann, Jan Christian Meyer, and Magnus Jahre

Published in
Proceedings of the 2016 International Conference on High Performance Computing & Simulation (HPCS)

Abstract. The CUDA and OpenCL programming models have facilitated the widespread adoption of general-purpose GPU programming for data-parallel applications. GPUs accelerate these applications by assigning groups of threads to SIMD units, which execute the same instruction for all threads in a group. Individual group threads might diverge and follow different paths of execution. Divergent branches cause performance degradation by under-utilizing the execution pipeline, resulting in a major performance bottleneck. The presence of unstructured control flow in addition to divergent branches causes further degradation, since it results in repeated execution of instructions.

In this paper, we propose a transformation which converts unstructured to structured control flow. It only creates tail-controlled loops, and properly nests all control flow splits and joins by inserting predicates. We implement an additional pass to NVIDIA's CUDA compiler to experimentally evaluate our transformation using synthetic unstructured control flow graphs, as well as kernels in the Rodinia benchmark suite. Our approach effectively eliminates redundant execution and potentially improves execution time for the synthetic unstructured control flow graphs. For the kernels in the benchmark suite, it only adds a minor, average overhead of 2.1% to the execution time of already structured kernels, and reduces execution time for the only unstructured kernel

by a factor of five. The representational overhead at compile-time is linear in terms of instructions.

C1.1. Introduction

Programming models such as CUDA [50] and OpenCL [146] allow developers to port applications to Graphic Processing Units (GPUs) and use their computing power for general purpose processing (GPGPU). GPUs accelerate data-parallel applications by mapping groups of threads to parallel execution units. These thread groups run in lock-step, executing the same instruction in Single Instruction Multiple Data (SIMD) mode. Individual threads in a group can *diverge* by following different paths of execution. Current GPUs handle these divergent branches by executing all paths sequentially, and masking out threads that do not take a path. Divergent threads reconverge at the immediate post-dominator (IPDOM)[1] of the branch instruction [73].

Branch divergence causes performance degradation by under-utilizing the execution pipeline. Moreover, IPDOM is the earliest point of reconvergence for structured control flow graphs (CFGs), but can result in redundant basic block execution with unstructured control flow. Branch divergence is therefore a major performance bottleneck [56, 77, 226, 9], exacerbated by unstructured control flow. The causes of unstructured control flow are programming language constructs such as goto, switch, and break statements, short circuiting operations, and compiler optimizations such as function inlining. Transforming unstructured control flow eliminates the redundant execution caused by divergence, mitigating its performance penalty. Moreover, compilers targeting AMD GPUs represent programs in the AMD IL [1] intermediate representation (IR). In contrast to NVIDIA's PTX [159], it only supports structured control flow, making transformations necessary.

In this paper, we propose a transformation to convert unstructured to structured control flow. It is based on the work from Bahmann *et. al.* [13] and consists of two phases: *loop restructuring* and *branch restructuring*. Loop restructuring converts all cyclic structures to tail-controlled loops, while branch restructuring ensures proper nesting of control flow splits and joins. This trans-

[1] See Section C1.3 for a definition of IPDOM

formation works by adding predicates and branches to CFGs. We modify the algorithm to admit head-controlled loops, and separate the implementation of the loop and branch restructuring phases. This seperation is possible because the insertion order of additional predicates and branches is irrelevant for GPUs. In contrast to previous solutions [61, 60, 226, 227], the use of predication instead of node splitting for restructuring CFGs avoids the risk of exponential code inflation [37].

We implement control flow restructuring as an additional pass to NVIDIA's CUDA compiler, and evaluate it experimentally using synthetic unstructured control flow graphs (CFGs), as well as all kernels in the Rodinia benchmark suite [40]. The synthetic unstructured CFGs demonstrate that our approach effectively eliminates redundant execution and potentially improves execution time for unstructured graphs with branch divergence. We use the Rodinia benchmark suite to evaluate transformation cost in terms of execution time and representational overhead at compile-time. Control flow restructuring adds a minor average overhead of 2.1% to the execution time of already structured kernels, and reduces execution time for the only unstructured kernel by a factor of five. While the overhead for already structured kernels is notable, it is significantly lower than previously reported results [61, 60]. The representational overhead at compile-time is linear in terms of instructions.

The paper is organized as follows: Section C1.2 describes the problem of branch divergence for unstructured control flow. Section C1.3 introduces terminology and definitions, while Section C1.4 describes our algorithm. We empirically evaluate it using synthetic unstructured CFGs and the Rodinia benchmark suite [40] in Section C1.5. Section C1.6 discusses related work, and Section C1.7 concludes and suggests further directions for research.

C1.2. Motivation

The IPDOM of a branch is the earliest possible point of reconvergence in structured CFGs, causing no redundant execution. In unstructured graphs, however, it is possible to introduce earlier points of reconvergence in order to avoid multiple executions of basic blocks. The following pseudocode shows a simple if-then-else statement with a short circuited condition:

```
if (c || d) {
  S1;
} else {
  S2;
}
S3;
```

Figure C1.1a depicts the corresponding CFG. The CFG is unstructured due to splits and joins not being properly nested.

Consider a warp of four threads executing this code segment, with threads $T1$ and $T2$ taking execution path ($c?, S1, S3$), thread T3 execution path ($c?, d?, S1, S3$), and thread $T4$ execution path ($c?, d?, S2, S3$) (see Figure C1.1a). The threads only reconverge before executing basic block $S3$. Thus, the basic block $S1$ would be executed twice, once for threads $T1$ and $T2$, and once for thread $T3$ as shown in the example schedule in Figure C1.1c.

Figure C1.1b depicts the CFG after control flow restructuring. The basic idea is to insert predicate assignments ($p := 0$ and $p := 1$) and branches ($p?$) such that all splits and joins are properly nested, and the resulting CFG is structured. This results in threads $T3$ and $T4$ reconverging at $NULL$ and threads $T1$, $T2$, $T3$, and $T4$ at $p?$, avoiding the duplicated execution of $S1$ as shown in the schedule of Figure C1.1d. The problem of repeated basic block execution compounds in bigger subgraphs, possibly resulting in more than two executions of individual nodes.

Structured graphs do not result in redundant code execution on GPUs, because nested divergent branches always reconverge in the inverse order of their execution, *i.e.* the inner branch reconverges before the outer branch. Our transformation converts kernels to structured graphs which consist only of tail-controlled loops and properly nested control flow splits and joins. For tail-controlled loops, divergent branches reconverge at the loop's epilogue, while divergent splits reconverge at the corresponding join. Thus, our transformation always produces graphs which preclude redundant code execution.

Developers are aware of the potential disadvantages of unstructured control flow for GPUs, and therefore try to avoid it. A compiler supporting control flow restructuring in combination with divergence analysis [47] would allow

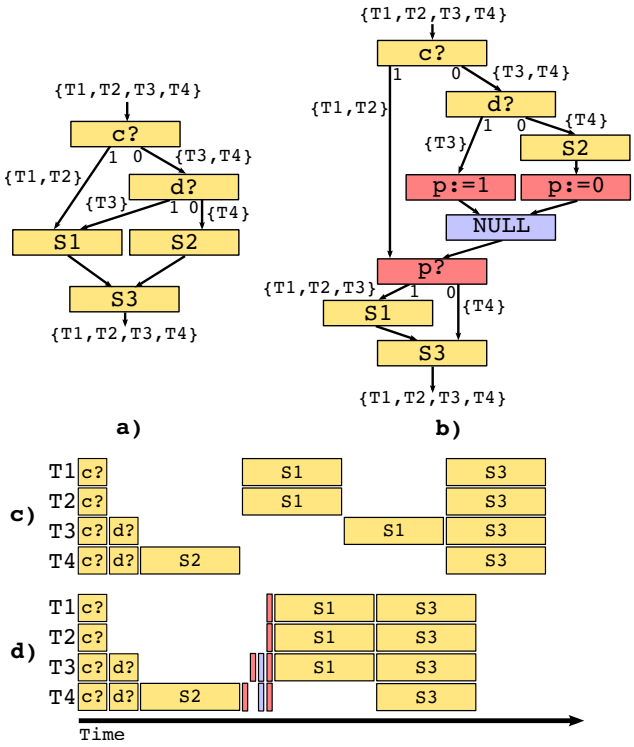


Figure C1.1.: Example illustrating the negative effect of unstructured control flow in the presence of branch divergence. **a)**: CFG for the pseudocode in Section C1.2. **b)**: Control flow restructured CFG from a). **c)**: Possible execution schedule for the CFG in a). **d)**: Possible execution schedule for the CFG in b).

a greater class of programs to be automatically translated into efficient GPU code.

C1.3. Terms and Definitions

A *control flow graph* is a directed graph consisting of nodes containing statements and edges representing transitions between statements. Outgoing edges are numbered with unique consecutive indices starting from zero (although we will omit writing out the index if a node has only one outgoing edge). Statements take the following form:

- $v := expr$ designates an assignment statement. The right hand side expression is evaluated and the result is assigned to the variable named on the left.
- $v?$ designates a branch statement. The variable is evaluated and execution resumes at the node reached through the correspondingly numbered edge.
- Other kinds of statements corresponding to original program behavior (observable side-effects) are allowed as well. We omit their discussion, as they are irrelevant for the control flow behavior discussed in this paper.

Only branch statements may have more than one outgoing edge. Furthermore, we require that each CFG has two designated nodes: The *entry node* without a predecessor, and the *exit node* without a successor. CFGs represent programs in imperative form: Starting at the entry node, successively evaluate each statement, until reaching the exit node.

Nodes are generally denoted by n with sub- and superscripts. Edges denoted by e with sub- and superscripts. An edge from a node n_1 to a node n_2 is written as $n_1 \to n_2$. We call n_1 the edge's *source* and n_2 its *target*.

Definition 19. *A CFG is called* single-entry/single-exit (SESE) *if its shape can be contracted into a single node by repeatedly applying the following steps:*

1. *If n' is unique successor of n, and n is unique predecessor of n', then remove edge $n \to n'$ and merge n and n'.*
2. *If n has only successors $n_0, n_1, \ldots$ and possibly n', n' has only predecessors $n_0, n_1, \ldots$ and possibly n, and each of $n_0, n_1, \ldots$ has only a single predecessor and successor, then remove all arcs $n \to n_i$, $n_i \to n'$, $n \to n'$ and merge all vertices.*
3. *If n has an edge pointing to itself and only one other successor, remove the edge $n \to n$.*
4. *If n has an outgoing edge targeting n' and n' has only one outgoing edge targeting n, then remove $n \to n'$, n', and $n' \to n$.*

Single-entry/single-exit CFGs are a subset of reducible CFGs and can be characterized by allowing only the following constructs:

- Straight line code.
- Properly nested conditionals ("`if/then/else`" or "`switch/case`" statements without fall-throughs).
- Tail-controlled loops ("`do/while`" loops without "`break`" or "`continue`" statements).
- Head-controlled loops ("`for`" or "`while`" loops without "`break`" or "`continue`" statements).

Definition 20. *A CFG is called* tail-structured *if its shape can be contracted into a single node by repeatedly applying rule 1, 2, and 3 from Definition 19.*

Tail-structured CFGs are a subset of SESE CFGs and correspond to programs with only straight line code, properly nested conditionals, and tail-controlled loops.

Definition 21. *A CFG is called* linear *if every vertex has exactly one incoming and outgoing edge.*

Linear CFGs are a subset of tail-structured CFGs and correspond to programs with only straight line code.

Definition 22. *A CFG is called* minimal *if it does not contain any nodes n and n' such that n' is the unique successor of n, and n the unique predecessor of n'. Thus, a minimal CFG contains no linear subgraphs.*

Definition 23. *An edge $n_1 \to n_2$* dominates *node n if*

1. *$n_2 \neq n$ and both n_1 and n_2 dominate n, or*
2. *$n_2 = n$ and n_1 dominates n.*

The dominator graph of edge e is the subgraph of all nodes dominated by e.

Intuitively, the dominator graph of an edge e is the subgraph where every path from the entry node to every node in this subgraph must pass through edge e.

Definition 24. *A node n' is said to be the immediate post-dominator (IPDOM) of another node n iff:*

- *n' post-dominates each immediate successor $n_0, n_1, \ldots$ of n, and*
- *for each other node n'' which also post-dominates each of $n_0, n_1, \ldots$ it holds that either $n'' = n'$ or that n'' post-dominates n'.*

Intuitively, the immediate post-dominator is the earliest point in a CFG where all paths starting at some node n necessarily reconverge.

C1.4. Control Flow Restructuring

Regularization of control flow can be facilitated by node cloning, predication or a combination of both techniques. Here we follow the approach laid out by Bahmann *et al.* [13] using predication only. Assume a CFG with a single entry and exit node. We restructure it using the procedures described below. The approach consists of two phases:

- Loops are detected and (possibly) transformed into a tail-controlled loop.
- Branches are restructured such that branch and join points are symmetric.

C1.4.1. Loop Restructuring

We start by identifying all strongly connected components (SCCs) and process each of them according to the procedure below. By necessity, neither entry nor exit node are part of any SCC. First, identify the following nodes and edges:

- *Entry edges* $e_0^E, e_1^E, \ldots$: All edges from a node outside the SCC into the SCC
- *Entry nodes* $n_0^E, n_1^E, \ldots, n_{k-1}^E$: All nodes that are target of at least one entry edge
- *Exit edges* $e_0^X, e_1^X, \ldots$: All edges from a node inside the SCC out of the SCC
- *Exit nodes* $n_0^X, n_1^X, \ldots, n_{l-1}^X$: All nodes that are target of at least one exit edge
- *Repetition edges* $e_0^R, e_1^R, \ldots$: All edges inside the SCC that have one entry node as target

See Figure C1.2 for illustration. We denote the set of nodes belonging to SCC by L. Initially, L induces the SCC subgraph. The following modifies the original graph, and we updates L as well such that it eventually induces a suitable structured loop subgraph. When we say "create a node within L" (as opposed to just "create a node") in the following, it means: Create the node in the CFG and update L such that it also has this node as member.

1. Pick two unused variables q and r to identify continuation location and loop repetition state, respectively.
2. If there are multiple entry nodes:
   a) Create a branch node b^E within L that evaluates q and continues at e_m^E iff $q = m$.
   b) Replace each entry edge: If the edge originally pointed to e_m^E, create an assignment statement $q := m$, divert the original entry edge to it, and continue control flow to b^E from there.

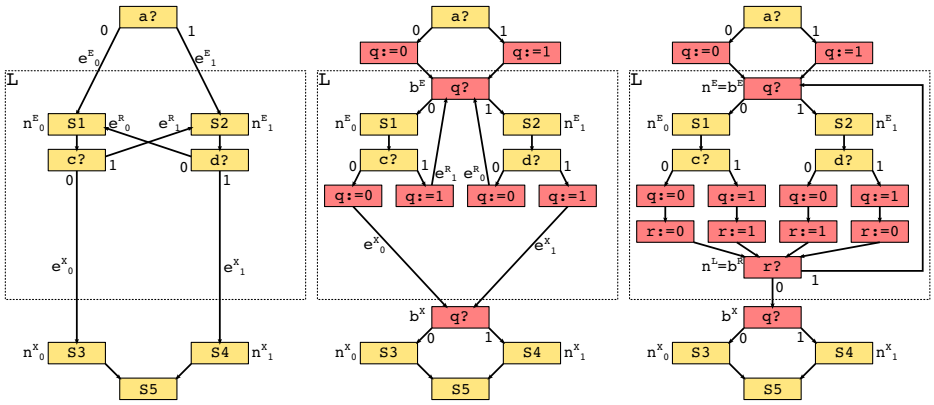


Figure C1.2.: Loop restructuring. **Left**: A CFG with an unstructured loop. Nodes and subgraphs identified by restructuring algorithm are marked as per algorithm description. **Center**: Intermediate state of loop restructuring after remodeling entry/exit control flow. **Right**: Final state after converting the loop to a single repetition and exit edge. Inserted nodes remodeling the original control flow are marked in red.

c) Replace each repetition edge: If the edge originally pointed to e_m^E, create an assignment node $q := m$ within L, divert the original repetition edge to it, and continue control flow to b^E from here. Record the newly recorded edges as repetition edges in lieu of the replaced ones.

After this step there is only one entry node: Either the newly created node b^E or the single original entry node. Denote it by n^E.

3. If there are multiple exit nodes:

   a) Insert a branch node b^X that evaluates q and continues at e_m^X iff $q = m$.

   b) Replace each exit edge: If the edge originally pointed to e_m^X, create an assignment node $q := m$ within L, divert the original repetition edge to it, and continue control flow to b^X.

   After this step there is only one exit node: Either the newly created node b^X or the single original exit node. Denote it by n^X.

4. If there are any two distinct nodes that are origin of either repetition and/or exit edges[2]:

   a) Create a branch node b^R within L evaluating r that continues at n^X if $r = 0$ and at n^E otherwise.

   b) Create an assignment node $r := 0$ within L and an edge from it to b^R. Divert all exit edges to it.

   c) Create an assignment node $r := 1$ within L, create an edge from it to b^R. Divert all repetition edges to it.

Note that the algorithm above does not actually modify the graph if it is already tail-structured. After this processing is complete, L contains two marked nodes:

- n^E is the unique entry node; all edges from outside L into L will have this node as target

[2] Note that this includes the cases of two or more repetition or exit edges

- n^L is the unique last node; there is only one edge leaving L, it originates in n^L

n^E has only one predecessor node within L, the node n^L. The edge $n^L \to n^E$ is the unique repetition edge of this loop. Temporarily remove this repetition edge, keeping track of the two nodes it used to connect. We repeatedly apply this whole loop transformation algorithm for any other SCC in the graph.

After all SCCs have been transformed as above, the resulting graph is acyclic. We process this acyclic graph according to the algorithm in the next section, and then re-insert all repetition edges that were set aside.

C1.4.2. Branch Restructuring

First, construct the "head" subgraph H as follows: Add the entry node to H. If the last node added has exactly one outgoing edge, add it as well as its target node to H. There are now two cases to consider:

- H covers the entire original graph.
- H covers only a portion of the original graph. There is a node b that was added to H last that has at least two outgoing edges.

In the first case the algorithm terminates: The original CFG is linear.

In the second case, record the outgoing edges of b as $f_0, f_1, \ldots, f_{m-1}$. Compute the dominator graphs of each f_k as B_k: This is the set of nodes and their connecting edges reachable from the entry node only through f_k. We call these the "branch" subgraphs. Record the remaining nodes and their connecting edges as the "tail" subgraph T. Some B_k may be empty, the corresponding edge f_k would in this case go directly to some node in T; in this case, create a "dummy" node in B_k and route the path through it. (See left of Figure C1.3 for illustration.)

We denote by $c_0, c_1, \ldots, c_{n-1}$ the *continuation points* in the tail subgraph: These are the nodes with T with at least one edge from either branch subgraph. There must be at least one such continuation point, and if there is exactly one then this branching construct has already a suitable structure. Otherwise, restructure T and B_k as follows:

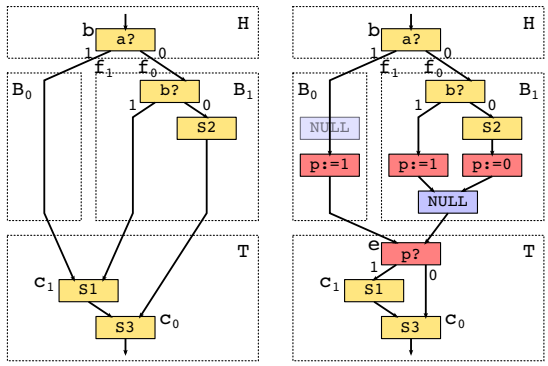


Figure C1.3.: Branch restructuring. **Left**: A CFG with unstructured branches. Nodes and subgraphs identified by restructuring algorithm are marked as per algorithm description. **Right**: The result of restructuring the CFG on the left hand side. Inserted nodes remodeling the original control flow through predication are marked in red. Dummy nodes inserted for structural purposes are marked in blue.

- Choose an unused auxiliary variable, denote it by p.
- Turn T into a graph with a single entry point e: Set up branches such that control resumes at c_k if p evaluates to k on entry.
- Turn each B_j into a graph with a single exit point: Divert edges pointing to c_k into statements that assign k to p, rejoin control flow for this in a single node that then proceeds to e.

Now, all edges leaving any B_j point to e. Finally, if some B_j has multiple exit paths, join all these paths into a single "dummy" node within B_j, and create a single exit edge from this node to e.

Recursively apply the same algorithm to each B_j as well as T. The control flow of the resulting graph is then tail-structured.

In the specification above, we utilized "n-way" branches: a single variable is used to identify one of n possible branch destination points. A subsequent pass can introduce additional auxiliary variables to reduce these constructs to

2-way branches. Additionally, superfluous dummy nodes in straight line code can be eliminated.

C1.4.3. Loop Restructuring with Copying

The loop restructuring algorithm in Section C1.4.1 transforms all loops into tail-controlled loops by inserting additional branches and assignments. Only loops that are already tail-controlled, *i.e.* do-while loops, are not altered. However, programmers express loops commonly as head-controlled loops, *i.e.* for and while loops. Loop restructuring would restructure these loops and introduce additional overhead. Figure C1.4 shows a simple head-controlled loop on the left and its equivalent after loop restructuring in the middle. The algorithm transforms an unconditional branch to a conditional one, and inserts two assignments, one of them being executed every loop iteration. This could potentially lead to overhead in execution time (see Section C1.5).

In order to mitigate the effect of loop restructuring on head-controlled loops, we employ loop inversion [130]. Basically, loop inversion transforms a head-controlled loop to an if-statement that surrounds a tail-controlled loop. Compilers perform this optimization in order to reduce the impact of branches at the expense of code duplication: a head-controlled loop features two branches, one conditional and one unconditional, while a tail-controlled loop features only one conditional branch.

In order to identify head-controlled loops, we inspect the entry, repetition, and exit nodes/edges of an SCC. We consider an SCC head-controlled, if it fulfills the following criteria:

- a single entry edge e_0^E, repetition edge e_0^R, and exit edge e_0^X
- the target of e_0^E, namely n_S^E, is the first node of a linear subgraph S
- the source of e_0^X, namely n_S^X, has two outgoing edges, e_0^X and e^B, and is the last node of subgraph S

The left image in Figure C1.4 illustrates the used notation. The linear subgraph S represents the condition of the loop, with edge e^B leading to the loop's body, and edge e_0^X exiting it.

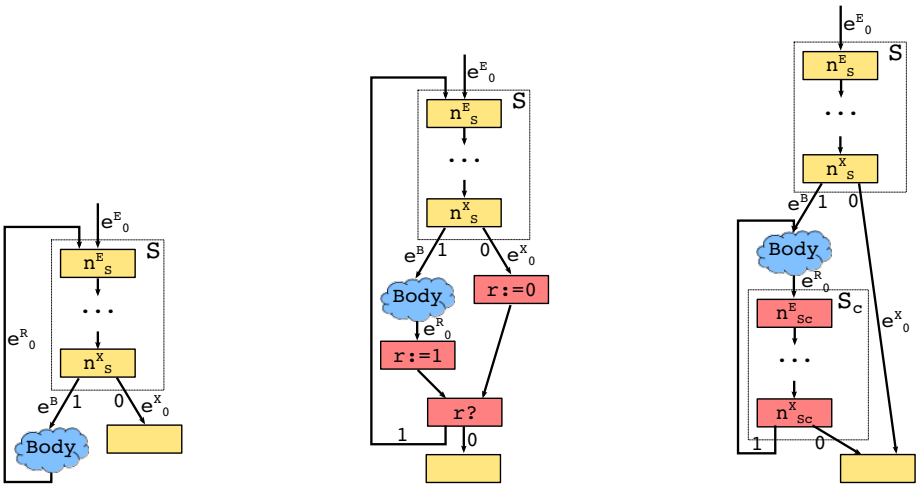


Figure C1.4.: Loop Restructuring with Copying. **Left**: A head-controlled loop. Nodes, edges, and subgraphs are labeled as identified by the algorithm. **Center**: The result of loop restructuring on a head-controlled loop. **Right**: The result of loop restructuring with copying on a head-controlled loop.

We restructure such a loop as follows:

- copy linear subgraph S. We further denote to this copy as S_c with its first node n^E_{Sc} and last node n^X_{Sc}
- divert edge e^R_0 to n^E_{Sc}
- insert a new repetition edge from n^X_{Sc} to the target of e^B
- insert a new exit edge from n^X_{Sc} to the target of e^X_0

Basically, the condition of the head-controlled loop is copied, and represents together with its body the new tail-controlled loop. The final result is shown in the right image of Figure C1.4. Note, even though we facilitate copying throughout this approach, it cannot lead to exponential code bloat [37].

C1.5. Experimental Evaluation

The transformation of unstructured control flow can eliminate redundant execution caused by branch divergence and therefore improve performance. This section describes the results of applying control flow restructuring to synthetic unstructured CFGs and kernels from the Rodinia benchmark suite [40]. The synthetic unstructured CFGs are used to demonstrate that our approach effectively eliminates redundant execution for unstructured graphs with branch divergence. We evaluate the dynamic overhead of branch restructuring and its potential impact on execution time. The benchmark suite consists mostly of SESE graphs, and we use it to evaluate the overhead of our transformations on these graphs in terms of execution time and representational overhead at compile-time.

C1.5.1. Compiler Implementation

We evaluated control flow restructuring by implementing it as an additional pass to NVIDIA's CUDA compiler. The pass takes PTX as input, restructures all CFGs, and produces PTX for further processing as output. We extracted the grammar for parsing PTX from the Ocelot compiler framework [57] and

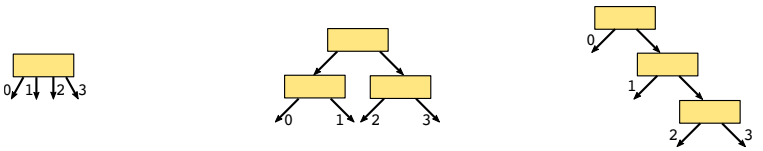


Figure C1.5.: **Left**: Basic block with 4 outgoing edges. **Center**: 4-way branch resolved in a breadth-first manner. **Right**: 4-way branch resolved in a depth-first manner.

create an AST with it. This AST is converted to a CFG, restructured with our algorithms from Section C1.4, and converted back to PTX.

A necessary constraint of control flow restructuring on a CFG is the support of n-way branches. These need to be resolved to cascades of 2-way branches with the help of additional auxiliary variables in order to make a conversion to PTX possible. Different cascades, such as breadth-first or depth-first as shown in Figure C1.5, or a mix of both, are possible. For our experiments, we resolve n-way branches with depth-first cascades of 2-way branches.

C1.5.2. Experimental Platform and Setup

The evaluation is performed on a system with an Intel Core i7-3770K CPU @ 3.5 GHz, an NVIDIA Tesla K20, and NVIDIA's driver version 346.46. We use the CUDA 7.0 toolkit, running on Ubuntu 12.04. We perform our experiments on a NVIDIA platform, since it allows us to experiment with structured and unstructured control flow. An AMD platform would have given us only the possibility to execute structured control flow, and would have made it impossible to quantify the difference between structured and unstructured control flow.

All programs were compiled with `-Xcicc=-O0` and `-Xptxas=-O0` to ensure no interference from other compilation stages. Ideally, control flow restructuring should be carried out as late in the compilation pipeline as possible in order to avoid side effects from other compilation stages.

Each benchmark in Section C1.5.4 is run 10 times, and we report the average kernel execution time of all runs. We measured execution times using

the CUDA profiler. In case benchmarks consists of multiple kernels, we add the execution time of all kernels in each run before computing the average. Benchmark results were verified to equal their results when restructuring is disabled.

C1.5.3. Synthetic Control Flow Graphs

This section demonstrates that our approach effectively eliminates redundant basic block executions for unstructured graphs with branch divergence. We evaluate the dynamic overhead of branch restructuring and its potential impact on execution time.

Experimental Setup

We evaluate the dynamic overhead of control flow restructuring by generating the incidence matrices for all acyclic CFGs with binary branches for a given dimension. We filter out all minimal unstructured CFGs and convert these matrices to CUDA code. All branches were made divergent to ensure redundant execution of basic blocks. The other basic blocks contained no computation, in order to ensure accurate dynamic overhead measurements.

We compile the CFGs with and without branch restructuring, and count the redundant executions of basic blocks for the unstructured case as well as the number of executed instructions for both cases using the CUDA profiler. We compute the *dynamic instruction overhead* for each graph by subtracting the number of executed instructions of the restructured CFG from the corresponding unstructured CFG.

Key Observations

We produce all synthetic CFGs up to 7 nodes, resulting in 1447 CFGs after filtering. We restrict our experiments to synthetic CFGs of this size, because it produces a sufficient number of unstructured graphs to demonstrate the effect of branch restructuring. Figure C1.6 shows the dynamic instruction overhead for these graphs. We group the CFGs by their number of redundantly

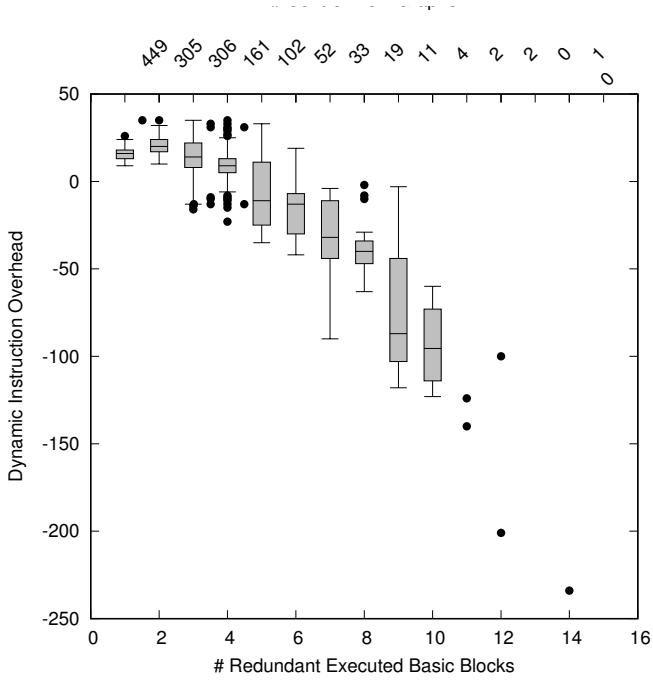


Figure C1.6.: Dynamic Instruction Overhead for all unstructured CFGs up to size 7.

executed basic blocks and count the number of CFGs for each group. For example, as shown in Figure C1.6, we count 449 CFGs which execute one basic block redundantly, and only 2 CFGs which execute 12 basic blocks redundantly. We plot a box and whisker plot for each group. The bottom and top of the boxes represent the first and third quartile, and the line inside the box the median. The ends of the whiskers indicate 1.5 times the interquartile range, and all points not within that range are outliers plotted as small dots.

Unstructured control flow in combination with branch divergence leads to redundant execution of basic blocks. Figure C1.6 shows that 73% of the synthetic CFGs have up to 3 redundant executions, and that the maximum number of redundantly executed basic blocks is 14. The maximum dynamic instruction overhead is 35, indicating that the added dynamic overhead of branch restructuring in the presence of branch divergence is small. Thus, in our experiments branch restructuring is desirable as long as the combined instruction count of the redundant executions exceeds 35. Moreover, Figure C1.6 clearly shows that the dynamic overhead of branch restructuring becomes smaller, the more redundant executions a graph contains. The dynamic instruction overhead is always negative for graphs with more than 7 redundant executions, indicating that fewer instructions are executed in the restructured than the corresponding unstructured graph. For these graphs, the number of redundantly executed instructions in the unstructured graphs *always* exceeds the overhead inserted by branch restructuring. Thus, branch restructuring is always desirable for these graphs even without any computation contained in the basic blocks.

C1.5.4. Benchmarks

This section describes the results of applying control flow restructuring to kernels from the Rodinia benchmark suite [40]. We evaluate the overhead of our transformations on these kernels in terms of execution time and representational overhead at compile-time.

	acyclic	cyclic
Linear	28	-
Tail-structured	56	3
SESE	3	139
Reducible	1	10
Irreducible	-	-
Total	240	

Table C1.1.: Program classification for the Rodinia benchmark suite. Classes are related as follows: Linear ⊂ Tail-structured ⊂ SESE ⊂ Reducible

Structural Analysis

In order to obtain an overview of the structural complexity of the benchmarks, we classified CFGs as *Linear* per Definition 21, *Tail-structured* per Definition 20, *single-entry/single-exit (SESE)* per Definition 19, *reducible* or *irreducible*. Tail-structured and SESE were identified by structural analysis [183], and irreducibility was determined by T1/T2 analysis [3]. A graph's cyclicity was identified by determining the presence of SCCs [206].

Table C1.1 shows the distribution of each class. The majority of the CFGs are single-entry/single-exit, and most acyclic SESE graphs are also tail-structured. Thus, the majority of programs in the Rodinia benchmark suite are expressed using simple if-then-else statements and head-controlled loops. Control flow restructuring introduces no overhead for the acyclic graphs, but transforms head-controlled loops to tail-controlled ones. We expect therefore an overhead associated with loop restructuring. Only a minority of the CFGs are in the reducible class. We inspected the source code for these graphs and found that the acyclic one is due to a switch statement with return statements in its cases. It is part of mummergpu. The cyclic graphs are due to loops with multiple exits and are part of hotspot, hybridsort, mummergpu, myocyte, particle-filter, and pathfinder.

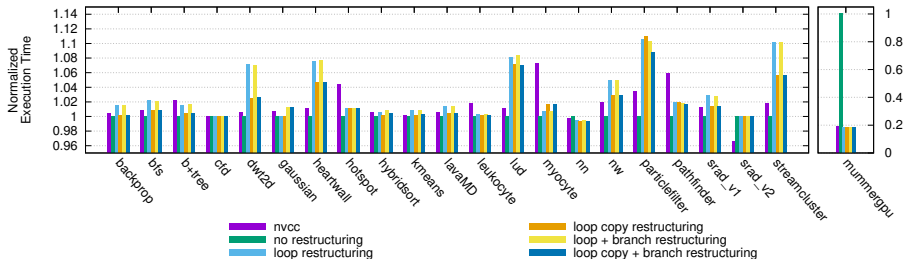


Figure C1.7.: Execution Times for different Control Flow Restructuring Configurations

Overall, the Rodinia benchmark suite consists mostly of SESE graphs, which can always be executed efficiently on GPUs. It offers little opportunity for improvements through control flow restructuring, considering that the presence of branch divergence is also required. This is rather unsurprising, since developers are aware of the potential disadvantages of unstructured control flow for GPUs and therefore try to avoid it. A compiler supporting control flow restructuring would be able to remove unstructured control flow altogether. This would allow programmers to delegate this task to the compiler and spend their time on tuning other aspects of a program.

Execution Times

Figure C1.7 shows the measured execution times for the Rodinia benchmark suite. We use six different restructurer configurations:

- *nvcc*: The benchmarks were compiled with the unmodified nvcc compilation pipeline.
- *no restructuring*: The PTX files are parsed, converted to CFGs, and immediately reconverted. No restructuring is performed.
- *loop restructuring*: Loop restructuring as described in Section C1.4.1.
- *loop copy restructuring*: Loop restructuring with copying as described in Section C1.4.3.

- *loop + branch restructuring*: Loop and branch restructuring as described in Section C1.4.2.
- *loop copy + branch restructuring*: Loop restructuring with copying and branch restructuring.

The *no restructuring* configuration serves as baseline, and all other configurations are normalized to it. The reason for using the *no restructuring* and not the *nvcc* configuration as baseline is due to the conversion passes. The CFG to AST conversion lays out basic blocks differently than they are in the input PTX file. This results in a different basic block order and therefore a different number of fall-through branches in the output PTX. The effect alters execution time by no more than 8%, except in the case of mummergpu, where we observe a 5 fold increase. We found that the difference is due to an additional basic block in the layout of nvcc. The basic block contains no instructions and has one incoming and outgoing edge and could therefore be safely removed without effecting the computation. However, ptxas produces pbk and brk instructions for the inner kernel loop when it is present. These instructions allow an early reconvergence of divergent threads in the loop, making it possible to avoid redundant executions for loops with multiple exits. Although these instructions allow to reduce execution time when divergence is present, nvcc seems not be able to reliably generate them.

Loop restructuring transforms all loops into tail-controlled loops by inserting additional branches and assignments. Only loops that are already tail-controlled are not altered. However, the majority of the loops in the Rodinia benchmark suite are head-controlled. These loops are converted to tail-controlled loops by converting one unconditional to a conditional branch, and inserting two assignments, with one of them being executed every loop iteration. This results in a noticeable execution time overhead for most benchmarks. The overhead is particularly pronounced with over 5% for dwt2d, heartwall, lud, nw, particlefilter, and streamcluster.

The benchmarks dwt2d, lud, nw, particlefilter, and streamcluster consist of very small kernels with an average execution time of less than 1.5ms per invocation. Loop restructuring adds additional instructions to the kernels of these benchmarks, and therefore creates an overhead that is a noticeable fraction of the execution time. For example, the average execution time of streamcluster's

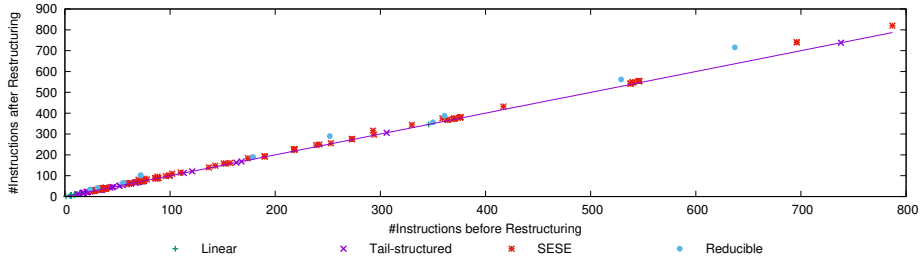


Figure C1.8.: Representational Overhead for Loop Copy + Branch Restructuring Configuration

kernel is only 750μs, but it is invoked 1611 times. In case of heartwall, the average kernel execution time is with 195ms significantly longer, but it consists of 48 head-controlled loops which are responsible for the overhead in execution time.

For most benchmarks, overhead is reduced using loop restructuring with copying. It transforms head-controlled to tail-controlled loops by employing loop inversion [130] instead of inserting assignments and branches. Thus, no additional assignment is executed every loop iteration. Another positive effect on execution time can be observed for mummergpu. Loop restructuring improves its performance 5 fold, rendering it equivalent to the code produced by nvcc. It allows divergent threads to reconvergence early and therefore reduces redundant execution as the pbk and brk instructions.

Branch restructuring is employed after all loops have been restructured and ensures proper nesting of splits and joins. Figure C1.7 shows that it has no significant effect on the execution times, and performs similarly to the corresponding loop restructuring. Most acyclic graphs are already tail-structured and we suspect a proper nesting of splits and joins in the cyclic ones as well. It is therefore no surprise that the execution times show no significant change compared to the corresponding loop restructuring configurations.

Overall, the experiments with the Rodinia benchmark suite indicate that control flow restructuring adds minor and varying overhead to the execution times of programs. It varies between not measurable and 12%, with an average of 2.1% among all benchmarks. The reason for this is that the Rodinia bench-

mark suite consists mainly of SESE graphs, and control flow restructuring only inflicts no overhead to the subset of tail-structured graphs. While this overhead is not insignificant, it is much lower than the 100 - 150% reported by Domínguez *et al.* [61, 60]. On the other hand, when unstructured control flow and branch divergence is present, control flow restructuring can help to reduce execution time significantly as demonstrated for mummergpu. This suggests that it should be applied more selectively, *e.g.* in combination with structural analyses [183] to discover unstructured subgraphs, and divergence analysis [47] for detecting divergent branches. In contrast to other restructuring methods [61, 60, 226, 227], it also does not lead to exponential code inflation [37].

Compile-Time Overhead

Control flow restructuring can add constructs to a CFG, causing representational overhead at compile-time. This is quantified in Figure C1.8, which relates the number of instructions before restructuring to the number of instructions after restructuring for the *loop copy + branch restructuring* configuration. The grey line marks the identity function, representing points with no overhead.

There is a clear linear relationship for all cases, suggesting that control flow restructuring is practically feasible in terms of space requirements. All linear and tail-structured graphs lie exactly on the line, confirming that no representational overhead is introduced. SESE and reducible graphs lie slightly above the line, indicating the insertion of additional instructions. The average representational overhead for these graphs in terms of instructions is 5.2%. Figure C1.8 is representative for all the other configurations, which exhibited similar behavior for their representational overhead.

C1.6. Related Work

Reducing the performance impact of thread divergence is a topic of extensive and ongoing research. Several works proposed changes to GPU hardware to ameliorate the problem. ElTantawy *et al.* [67] replaced the traditional stack

based thread reconvergence mechanism with a set of tables, potentially allowing warps to reconverge before the branch's IPDOM. They evaluated their approach on a set of benchmarks with unstructured control flow and achieved a harmonic mean speedup of 32% compared to traditional execution. Branch herding was proposed by Sartori *et al.* [179]. It forces all threads of a warp to take the path of the majority. This led to incorrect results, but was acceptable for error tolerant applications such as visual computing applications. Their hardware implementation improved performance for a set of benchmarks by 30% on average. Brunie *et al.* [31] proposed to add additional hardware to co-issue different instructions to disjoint sets of the same warp, or to a subset of a different warp. Diamos *et al.* [56] proposed thread frontiers, a combined hardware and software approach. In this approach, the compiler finds potential early reconvergence points, while additional hardware checks whether a warp can reconverge at these points.

Two software based approaches were proposed by Han *et al.* [77]. They reduce branch divergence through iteration delaying and branch distribution. Iteration delaying reorders loop iterations with branches so that branches taking the same direction are executed together. Branch distribution factors out similar code from branches. Both techniques require manual code rewriting. Zhang *et al.* [236] removed divergence through data reordering and job swapping, *i.e.* changing the mapping between threads, data, and work. This must be done asynchronously by the CPU at runtime, and therefore requires to launch a kernel multiple times in a loop. Lee *et al.* [113] proposed algorithms that remove all control flow by predicating and linearizing different execution paths. They implemented their algorithms in the CUDA LLVM compiler and showed that a predication-only architecture based on their algorithms is competitive in performance to one with hardware support for tracking divergence.

Finally, like our method, several approaches transform unstructured to structured control flow to reduce the impact of branch divergence. Anantpur *et al.* [9] proposed a technique for transforming unstructured to structured CFGs by linearizing them with the help of guard variables. They implemented it as PTX transformations and evaluated it on a set of benchmarks. It increased code size by up to 10% and execution time by up to 73%. Wu *et al.* [226, 227] use adaptions of the transformations of Zhang *et al.* [237]. They show that several Rodinia, Parboil, and Optix benchmarks, as well as CUDA SDK samples

contain unstructured control flow. Applying their transformations increased static instruction count, and decreased performance by up to 1% due to code expansion. Dominguez *et al.* [60, 61] developed a tool for translating PTX to AMD IL in order to understand the performance differences between structured and unstructured control flow on GPUs. They also used the transformations of Zhang *et al.* [237] to handle unstructured control flow. Their tool produced code that performed 2.1 times worse on average than a straightforward manual CUDA to OpenCL translation.

C1.7. Conclusion and Future Work

In this paper, we presented a transformation for converting unstructured to structured control flow. Our evaluation shows that our approach effectively eliminates redundant basic block execution and improves execution time for unstructured graphs with branch divergence. It adds a minor average overhead of 2.1% to execution time of already structured kernels. While this overhead is notable, it is significantly lower than the 100-150% reported by Domínguez *et al.* [61, 60]. This suggests that our transformations should be applied more selectively, *e.g.* in combination with structural analysis [183] to discover unstructured subgraphs, and divergence analysis [47] for detecting divergent branches. The representational overhead at compile-time is linear in terms of instructions. In contrast to other restructuring methods [61, 60, 226, 227], exponential code inflation is impossible [37].

We also showed that the main increase in execution time in structured kernels is due to restructuring of head-controlled loops. Our main direction for future work is therefore to extend our algorithm to SESE graphs in order to avoid the added overhead and therefore the need for structural analysis. Another direction for future work would be to combine loop restructuring with loop merging [78]. This optimization merges a divergent loop with one or more of its surrounding loops in order to overlap the iteration spaces of the inner loop for threads of different warps.

Part D.

Grain Graphs

D1. Diagnosing Highly-Parallel OpenMP Programs With Aggregated Grain Graphs

Nico Reissmann and Ananya Muddukrishna

Published in
Proceedings of the 2018 European Conference on Parallel and Distributed Computing (Euro-Par)

Abstract. Grain graphs simplify OpenMP performance analysis by visualizing performance problems from a fork-join perspective that is familiar to programmers. However, when programmers decide to expose a high amount of parallelism by creating thousands of task and parallel for-loop chunk instances, the resulting grain graph becomes large and tedious to understand. We present an aggregation method that hierarchically groups related nodes together to reduce grain graphs of any size to one single node. This aggregated graph is then navigated by progressively uncovering groups and following visual clues that guide programmers towards problems while hiding non-problematic regions. Our approach enhances productivity by enabling programmers to understand problems in highly-parallel OpenMP programs with less effort than before.

D1.1. Introduction

The *grain graph* [135] is a recent visualization method that simplifies OpenMP performance analysis by highlighting problems from a fork-join perspective. Task and parallel for-loop chunk instances are collectively termed *grains* in

the grain graph method. Grains that suffer performance problems such as work inflation, inadequate parallelism, and low parallelization benefit are pinpointed on the grain graph along with precise links to the problematic source code. This enables programmers to perform optimizations productively without relying on experts or trial-and-error tuning.

Programmers optimize OpenMP programs for large machines with hundreds of cores by exposing a high amount of parallelism during execution. This is achieved by adjusting special program inputs called *cutoffs* and *chunk sizes* such that a large number of fine-grained tasks and for-loop chunks are created. Scalability problems invariably occur when the runtime system is unable to efficiently handle the parallelism exposed [143, 231, 133]. These problems are pinpointed on the grain graph using metrics that isolate low parallelization benefit, work inflation, and poor memory hierarchy utilization to specific grains.

However, the large grain graphs resulting from highly-parallel OpenMP execution make problem diagnosis tedious (Fig. D1.1). Programmers have to zoom and pan to different sections while remembering characteristics of visited sections. Problems that are spread out become difficult to locate. Non-problematic grains that are shown dimmed to increase focus on problems combine at lower zoom levels and become pronounced. Programmers can perceive the dimming effect and spot problematic grains only when zoomed into higher levels. A powerful workstation with a large screen and copious amount of memory is required to render large grain graphs responsively. In light of these demands, programmers prefer to pore over text summaries and tabular formats of large graphs and reserve the visual approach only for small graphs.

This paper contributes with a new aggregation method that makes visual analysis of large grain graphs practical. The aggregation method (Section D1.3) groups related nodes by matching recurrent patterns in the grain graph, ultimately resulting in an aggregated graph with a single group node. Programmers navigate the aggregated graph by progressively opening and closing groups. Groups with problems are highlighted and non-problematic sections are removed from sight for distraction-free diagnosis. Navigation is further sped up through new group-based metrics that enable programmers to traverse

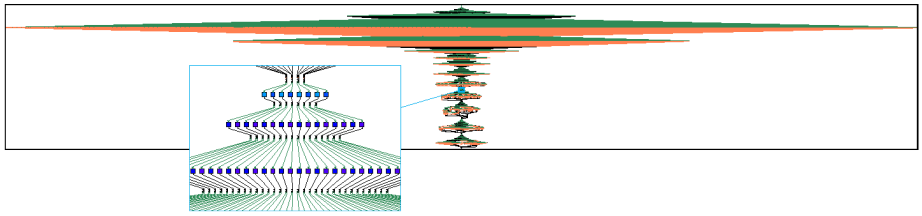
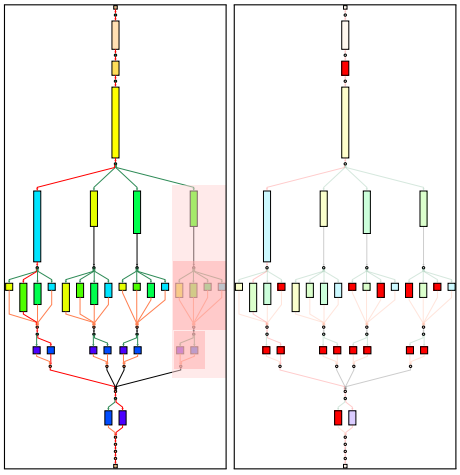


Figure D1.1.: The grain graph of the task-recursive Sort program from the Barcelona OpenMP Task Suite (BOTS) for a high-parallelism input (`n=20971520, cutoffs={65536,8192,128}`) is dense with 11059 grains. Inset (blue box) zooms into a section at magnification 40X.

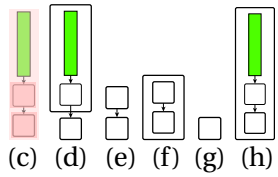
the critical path and compare groups for structural similarity. Using highly-parallel executions of standard OpenMP programs, we demonstrate (Sections D1.3 and D1.4) that aggregated grain graphs enhance the the state-of-the-art in OpenMP problem diagnosis.

D1.2. Background on Grain Graphs

The grain graph [135] is a visualization for OpenMP that connects performance problems to the fork-join program structure at the resolution of *grains* – task and parallel for-loop chunk instances created during execution. This simplifies problem diagnosis as programmers can readily identify with the fork-join program structure. In contrast, existing visualizations based on timeliness and call graphs complicate diagnosis by connecting performance problems to scheduling events that are unfamiliar and unpredictable to programmers [135, 93]. Experts who understand scheduling internals nevertheless find it tiring to follow timelines and call graphs that depict recursive task-based execution – a popular style of using OpenMP.



(a) (b)



(c) (d) (e) (f) (g) (h)

```
1  bool is_graingroup(Node n) {
2      return is_grain(n) || is_forkjoin(n) || is_linear(n)
3  }
4
5  void reduce(Node n) {
6      if (is_graingroup(n) && is_graingroup(succ(n))) {
7          n' ← reduce_linear(n)
8          reduce(n')
9      } else if (is_graingroup(n) && is_fork(succ(n))) {
10         reduce(succ(n))
11         n' ← reduce_linear(n)
12         reduce(n')
13     } else if (is_fork(n)) {
14         forall s in succ(n)
15             reduce(s)
16         n' ← reduce_forkjoin(n)
17         reduce(n')
18     }
19 }
```

(i)

Figure D1.2.: Grain graph of the task-based Sort program from BOTS for small input (`n=512, cutoffs={256,64,16}`). (a) Structural view (b) Problem view highlighting low parallel benefit in red (c) After two fork-join pattern reductions of the highlighted subgraph (d-g) Linear pattern reductions leading to a single group node (h) After normalization (i) Reduction pseudocode

D1.2.1. Structure

The grain graph is a directed acyclic graph whose nodes denote grains and runtime system operations, and edges denote control-flow. Parent and child grains are shown in close proximity on the graph using *logical-time* placement [51, 93] to maintain familiarity with the fork-join perspective (Fig. D1.2a[1]). The grain graph is laid out using the *Sugiyama* layout [201, 66]. This layout places nodes in layers, removes cycles, and prevents edge crossings. These features are essential to depict fork-join progression in an uncluttered manner.

[1] Readers should print in color as they are crucial to appreciate grains graphs.

D1.2.2. Diagnosing problems

Grains are annotated with unique schedule-independent identifiers, links to source code locations, as well as performance metrics measured during profiling and derived post profiling. Profiled metrics include execution time, cache miss ratio, memory latency, and timestamps of control-flow events such as grain creation and synchronization. These metrics are used to compute derived metrics such as critical path, work deviation, instantaneous parallelism, memory hierarchy utilization, scatter, load balance, and parallel benefit.

Parallel benefit is a custom metric used in several discussions of this paper. It is computed by dividing a grain's execution time by its parallelization cost including creation time. This metric aids inlining and cutoff decisions as grains with low parallel benefit should be executed sequentially to reduce overhead.

Commonly sought out metrics are encoded visually for quick identification on the graph (Fig. D1.2a). The length of a grain is set proportional to its execution time. Grain colors denote source code locations by default. Edges are colored by type and highlighted red if they are on the critical path.

Grains with metric values that cross programmer-defined thresholds are inferred as problematic. The thresholds have sensible values by default. Problematic grains are highlighted with a color that encodes problem severity in a separate view while non-problematic grains are dimmed (Fig. D1.2b). Additionally, problems are summarized in a separate text file and highlighted in a tabular form of the grain graph shown on a separate visualization widget.

Grain graphs have multiple conceptual views with colors encoding a single problem or property per view. Programmers shift between these views to understand properties or tackle problems. Problematic grains are highlighted and non-problematic grains are dimmed, and clicking on a grain opens a separate window that shows the grain's properties and performance metrics. Fig. D1.2b-a show the programmer cycling between the low parallel benefit problem view and the structural view where no problems are highlighted.

D1.3. Grain Graph Aggregation Method

Our aggregation method for grain graphs conceptually consists of four phases:

1. **Reduction** matches and replaces subgraph patterns with group nodes to construct an *aggregation tree*. This tree captures the graph structure and serves as a basis for further processing. After aggregation is complete, the tree is converted back to an aggregated grain graph with problematic grains exposed and non-problematic grains hidden.

2. **Normalization** transforms the aggregation tree into a canonical form, simplifying further processing.

3. **Propagation** propagates grain metrics at the leaves of the tree to upper levels in a sensible manner.

4. **Separation** transforms the aggregation tree to separate problematic nodes. This enables grouping and hiding of non-problematic grains in the resulting aggregated graph.

The algorithmic complexity of all four phases is linear in the number of graph nodes plus edges. The rest of this section explains the phases in detail and discusses the navigation of the resulting aggregated graph at the end.

D1.3.1. Reduction

The reduction phase matches a *fork-join* and *linear* pattern, and replaces them with group nodes to construct an *aggregation tree*. The fork-join pattern consists of a single fork node connected to child grains or groups, which in turn are connected to a join node (Fig. D1.2c). The linear pattern has two nodes, either a grain or a group node, that are connected to each other (Fig. D1.2d). Both patterns are repeatedly matched, and replaced by a single group node until the entire grain graph is reduced to a single node (Fig. D1.2d-g).

The pseudocode of the reduction algorithm is shown in (Fig. D1.2i). The key steps in the psuedocode are explained next:

- Line 6 matches the linear pattern (Fig. D1.2d-g). It uses the helper function $is\_graingroup$ to detect whether a node and its successor is a grain or a group, and reduces the pattern to a linear group node. Reduction continues with the newly-created group node.

- Line 9 matches a grain or group node with a fork node as successor. The matched fork node is recursively aggregated to a fork-join group node (Fig. D1.2c). The resulting linear pattern is then reduced to a linear group node. Reduction continues with the linear group node.

- Line 13 matches a fork node (Fig. D1.2a) and recursively aggregates all successors of the fork node. The resulting fork-join pattern is then reduced to a fork-join group node. Reduction continues with the fork-join group node.

The grain graph is reduced greedily by the reduction algorithm. It always continues with the newly-created group node after a pattern match and never traverses past a join node. This ensures that the innermost fork-join in a nesting is reduced first.

The aggregation tree consisting of group and grain nodes explicitly captures the grain graph's nesting and fork-join structure. The leaves of the tree are grains and its intermediate nodes are the newly-created group nodes. Linear group nodes have the two matched nodes from the pattern as children, whereas fork-join group nodes have the children of the matched fork node as children.

The reduction algorithm is applicable to grain graphs where parents synchronize with all their children before completion. This essential property ensures that fork-join patterns are properly nested, permitting their reduction in a hierarchy of group nodes. While this property holds for well-behaved OpenMP 3.X programs, the *taskgroup* construct in OpenMP 4.0 violates this property. The construct permits parents to synchronize with their children and descendants in one step. This impedes reduction unless the grain graph is restructured so that all descendants are placed as immediate children of the root parent.

D1.3.2. Normalization

Normalization transforms the aggregation tree into a canonical form by flattening nested linear group nodes. In the reduction phase, linear group nodes are always created for a pair of grain or group nodes, even if more nodes are chained together. This constructs nested linear subtrees where linear group nodes are the children of other linear group nodes as exemplified in Fig. D1.2d-g. Normalization flattens these subtrees to a single linear group node with all non-linear group nodes from the subtree as its children (Fig. D1.2h). In practice, this phase can be incorporated into the previous phase to speedup aggregation.

D1.3.3. Propagation

This phase propagates leaf node metrics to the enclosing groups all the way up to the root node. It traverses the aggregation tree in post-order and attributes group nodes with metrics sensibly-derived from their children. For example, the *work* metric of a group node is the sum of the execution times of its children, while the schedule-independent identifiers of children are concatenated with the group node's depth to derive a schedule-independent identifier.

Metrics are attributed such that problems propagate to the root group. If a child is problematic, then its parent is marked as problematic as well. The minimum of the memory hierarchy utilization, parallel benefit, and instantaneous parallelism as well as the maximum of the load balance, work deviation, and scatter metrics of children are attributed to the parent group. Programmers can refine existing propagation metrics and define new ones. Given this ability, the range of values and other summary statistics of a group can be easily captured (for example, as string attributes). One useful custom metric that programmers could define is the percentage of time spent by a group on the critical path.

D1.3.4. Separation

The separation phase groups non-problematic nodes to separate them from problematic nodes. This enables programmers to focus on problems and re-

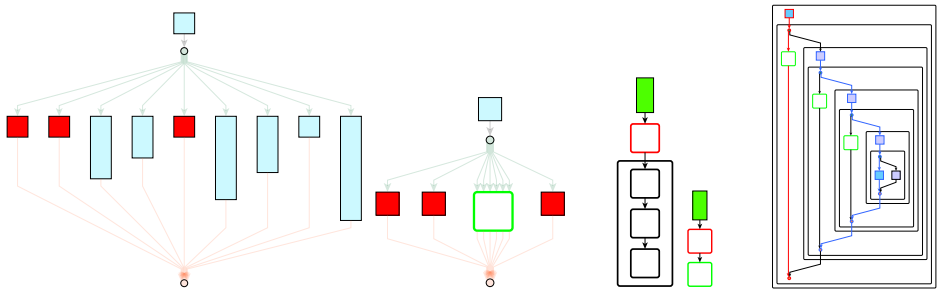


Figure D1.3.: Separation of problematic from non-problematic nodes. (a-b) Fork-join node separation. (c-d) Linear node separation. (e) Local (blue) and global (red) critical paths

duces graph viewer load. For example, consider a fork-join group that encloses a thousand grains among which only a single grain is problematic. An unseparated graph would require all grains to be rendered, while a separated graph requires only the rendering of one problematic grain and a non-problematic group node.

Separation traverses the aggregation tree in post-order and separates subtrees rooted at fork-join and linear nodes. In a fork-join separation, all non-problematic children of a fork-join node are grouped under a newly-created group node (Fig. D1.3a-b), while in a linear node separation, all consecutive non-problematic children of a linear group node are grouped under a new linear group node (Fig. D1.3c-d). After the separation phase, the aggregation tree is converted back to a grain graph where non-problematic subgraphs are hidden.

D1.3.5. Navigation

The navigation of an aggregated graph starts at the root and continues by progressively opening/closing group nodes to understand graph structure and problems (Fig. D1.4). In contrast to the navigation in unaggregated graphs, the cognitive load on programmers and the graph viewer's resources are reduced as only a subset of the grains are laid out. Navigation is sped up using several optimizations:

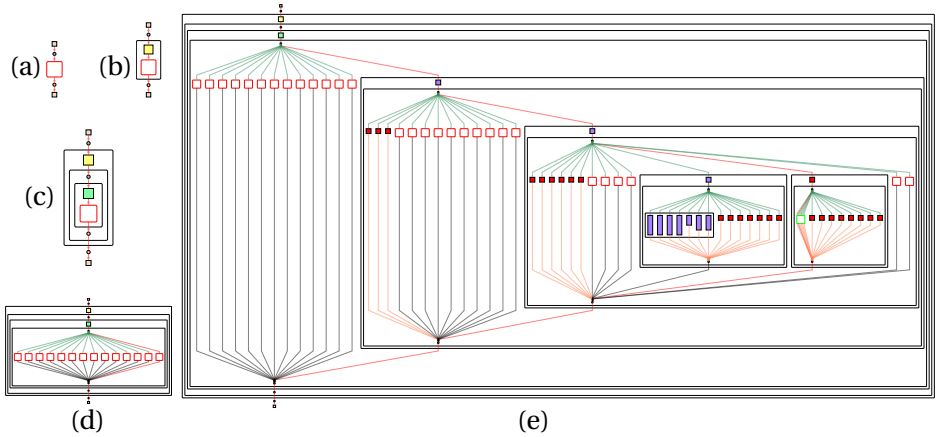


Figure D1.4.: Navigating the aggregated grain graph of NQueens program from BOTS for high-parallelism input (`n=14, cutoff=4`). The graph has 21492 grains and 3073 group nodes. Grains with low parallel benefit are highlighted as problems. (a-d) Drilling down to sibling groups at a depth of 3 from the root group. (a) Root group. (b) At depth 1. (c) At depth 2. (d) At depth 3. (e) Drilling down along the critical path to sibling groups at the lowest depth.

1. Groups can be opened to show all grains including those inside subgroups (full collapse), or drilled down to a specific group or depth level (Fig. D1.4).

2. Group nodes are drawn as rounded rectangles with no filling to differentiate them from grains. Group metrics are shown in a separate property window, similar to grains. Opened groups grow as large as required to envelop members whereas closed group nodes have a constant size. The borders of problematic closed groups are colored red to draw programmer attention, while the borders of non-problematic groups are colored green for quick identification. Our choices of group colors and sizes allow programmers already familiar with grain graphs to smoothly transit to the aggregation feature.

3. Once a group's structure is known, other similarly structured groups can be navigated confidently or skipped if problem-free. For example, twelve groups in Fig. D1.4d have the same structure. Group similarity is computed on-demand using a Weisfeiler-Lehman graph kernel [186].

4. Groups on the *global critical path* (gcb) are inspected first since they are good optimization candidates (Fig. D1.4e). The local critical path of groups not on the gcb can be computed on-demand and used for prioritized inspection (Fig. D1.3e). If off-gcb grains are optimized to reduce the total amount of work, the resulting slack can be used to execute grains on the gcb.

D1.4. Prototype Implementation

The grain graph visualization is implemented in a prototype [136] that produces grain graphs in GRAPHML by processing profiling data from OMPT extensions [107] or the MIR runtime system [134, 133, 132]. We extended the prototype to produce aggregated graphs upon programmer request [166]. The aggregation method was implemented in C++, leveraging support for nested groups [28] in GRAPHML and using the igraph [49] library for basic graph processing.

We used the graph viewer yEd [232] to visualize aggregated grain graphs since it has sufficiently mature support for GRAPHML files with nested aggregations. For example, it has features to interactively open and close groups, and jump to groups at any hierarchy level. Its property editor dialog shows the annotations of group nodes. Switching between problem views was achieved by cycling through tabs that highlighted different problems.

External programs parameterized by group identifiers were used to compute local critical path and similarity. These programs do not update the visualization and programmers are required to manually load their output into yEd. Similarity was computed using a third-party implementation [202] of the Weisfeiler-Lehman graph kernel.

We recognize that interactions with aggregated graphs in yED have quite some room for improvement. Our plan is to incorporate improvements in a dedicated grain graph viewer as yEd is closed-source. The dedicated viewer will also enable programmers to define custom metrics derived from basic grain and group metrics in a GUI. This improves over the prototype where programmers customize metrics by editing source-code in convenient locations.

D1.5. Evaluation

We tested our prototype on C/C++ benchmarks from SPEC OMP 2012 (SPEC-OMP12), Barcelona OpenMP Task Suite v2.1.2 (BOTS) and Parsec v3.0 (Parsec). The benchmarks were compiled with MIR-linked GCC v4.4.7 and profiled on a 48-core machine with 64GB memory and four AMD Opteron 6172 processors running at 2.1GHz with frequency scaling disabled. We provided input values that exposed abundant, fine-grained parallelism to standard OpenMP programs to obtain large grain graphs (Table D1.1).

D1.5.1. Visible node count

We use the metric *visible node count* (θ) to judge the ability of our aggregation method to reduce programmer effort in navigating and diagnosing problems. θ is defined as the minimum number of visible nodes in a grain graph while

Table D1.1.: Benefit of aggregation for standard OpenMP benchmarks.

Benchmark	**Input**	**#Nodes**	**#Grains**	θ_c^{max}	**Savings (%)**	**Low Parallel Benefit**		
						#Prbl. Grains	θ_{pb}^{max}	**Savings (%)**
Strassen[1]	8192, 128, 2000	176480	137258	60	99.97	157	49	99.97
Bodytrack[2]	B261, 4, 261, 4000, 5, 3, 48, 0	126615	69061	5767	95.45	24627	5757	95.45
Floorplan[1]	15, 7	117960	82490	149	99.87	31125	148	99.87
376.kdtree[3]	200000, 10, 2	32808	16400	58	99.82	2055	57	99.83
NQueens[1]	14, 4	24565	21492	70	99.71	10540	66	99.73
359.botsspar[3]	64, 64	24161	23905	1154	95.22	2	9	99.96
358.botsalgn[3]	prot.200.aa	20505	20101	406	98.02	7	17	99.92
Sort[1]	20971520, 65536, 8192, 128	20293	11509	55	99.73	288	51	99.75
FFT[1]	16777216, 8192, 2	9240	4592	53	99.43	414	49	99.47
367.imagick[3]	See caption of Fig. D1.5	3935	3801	405	89.71	649	182	95.37
Blackscholes[2]	4M	2205	1201	112	94.92	400	112	94.92
Freqmine[2]	kosarak_990k.dat, 790	2111	2017	389	81.57	66	30	98.58

[1] BOTS [2] Parsec [3] SPEC-OMP12

diagnosing a problematic grain. If it is small, the cognitive load on programmers and the resource requirements of viewers are reduced.

The visible node count for a problematic grain in an aggregated graph is the number of nodes exposed by opening groups in the path leading to the grain. In contrast, the visible node count in an unaggregated graph is equal to the number of nodes in the entire graph irrespective of the position of the problematic grain, assuming programmers do not pan and zoom to the vicinity of the problematic grain manually.

Table D1.1 shows the maximum θ for two cases. The first is a conservative case (θ_c^{max}) that assumes all grains in the graph are problematic, while the second (θ_{pb}^{max}) considers graphs with low parallel benefit. For both cases, the reduction in maximum θ compared to the total size of the graph, *i.e.*, the maximum θ for the unaggregated graph, is reported as *Savings*.

For the conservative case, we see a large reduction in θ. The biggest saving is 99.97% for the Strassen benchmark and the smallest saving is 81.57% for Freqmine, with an average saving of 95.98%. This shows that aggregation can significantly reduce θ for any problematic grain in our evaluation setup.

For the second case, we see a further reduction in θ since non-problematic grains are grouped during the separation phase (Section 3). Benchmarks Freqmine, 367.imagick, 358.botsalgn, 359.botsspar, show large savings from ag-

gregation since they contain a small number of problematic grains. On the other hand, Bodytrack and Floorplan show barely any improvement over the conservative case due to a higher concentration of problematic grains that are clustered as siblings. Problematic siblings are ignored during separation by design.

D1.5.2. Reducing distractions

We further illustrate the benefit of aggregation using the 367.imagick benchmark from SPEC-OMP12 for an input that SPEC programmers noticed as poorly scaling. The unaggregated grain graph shows a chain of nine dense for-loops (Fig. D1.5a). The sixth loop contains several chunks that suffer from low parallel benefit since several instances of the parallelization-throttling macro *omp_throttle* are missing in the source. Diagnosing these problematic chunks requires programmers to sweep attentively across the graph ignoring the abundance of non-problematic grains and the frequent non-responsive rendering of the graph. The aggregated graph enables programmers to diagnose problematic chunks group by group (Fig. D1.5b), keeping only those groups with problematic chunks open, while uninteresting loops and non-problematic chunks are hidden from sight. This results in a more responsive graph viewer since fewer nodes need to be rendered.

D1.5.3. Similarity across runs

Grain graphs produced from two independent executions of a given program can be different in shape due to unpredictable inlining decisions taken by the runtime system or if the program adapts its behavior sensitive to available execution resources. Understanding such changes can provide vital clues for problem diagnosis. However, detecting the dissimilar sections by manually inspecting a pair of large grain graphs is extremely tiring and akin to finding matches between fingerprints using a magnifying lens.

Similarity is a powerful metric that not just helps to skip over structurally similar groups within the same graph (as demonstrated in Section D1.3.5), but can also compare groups across runs to detect structural differences. Programmers can gradually open two graphs side-by-side and compute the similarity

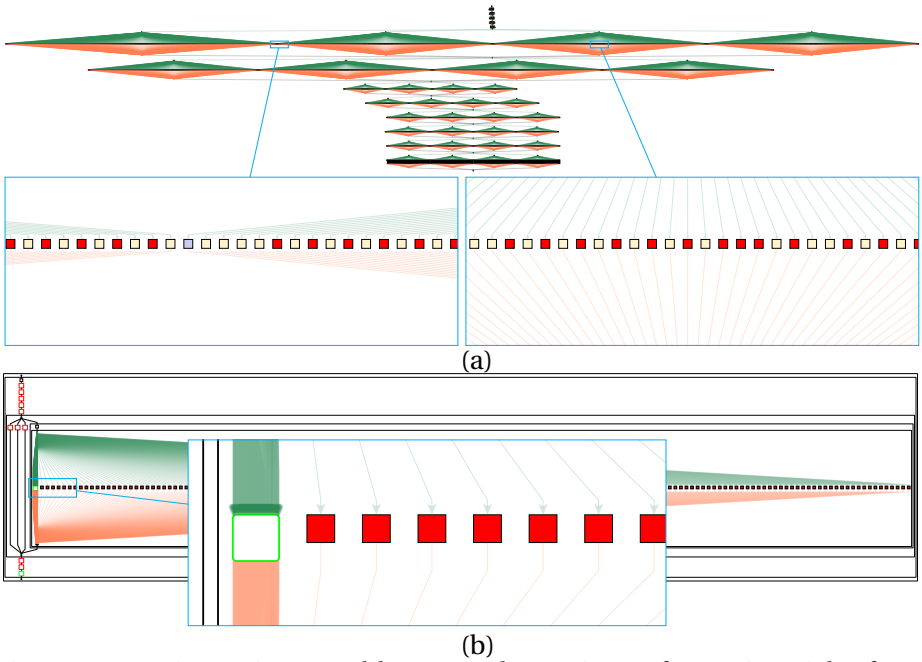


Figure D1.5.: Diagnosing problems with grains of 367.imagick from SPEC-OMP12 for input `-shear 31 -resize 1280x960 -negate -edge 14 -implode 1.2 -flop -convolve 1,2,1,4,3,4,1,2,1 -edge 100 ref/input/input1.tga`. (a) Sweeping across the entire unaggregated graph with 3801 grains to spot problems. (b) Aggregated grain graph enables programmers to diagnose problematic grains group-wise. Non-problematic grains are separated to promote focus (inset).

metric for visible groups using their schedule-independent identifiers. Those groups that have the same identifier but different similarity metrics are the sections that have changed between the graphs. We demonstrate this for the Floorplan program from BOTS in Figure D1.6. Floorplan is a search-based program whose pruning behavior changes non-deterministically when more cores are allotted for execution.

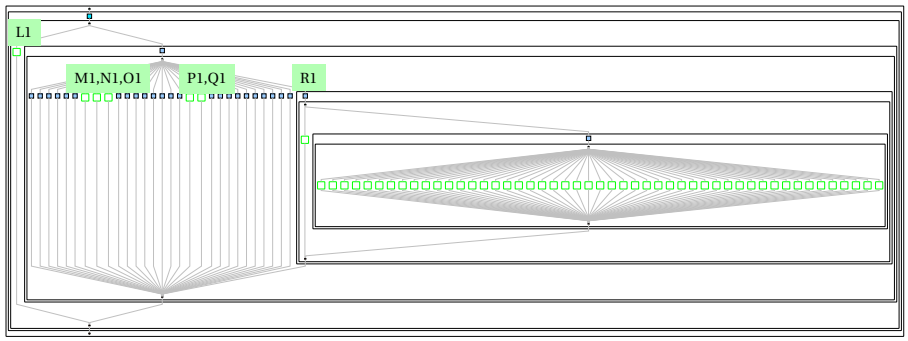
D1.6. Related Work

Aggregation is a standard approach to scale visualizations with increasing data [94, 220]. Sensible dimensions for aggregation include the program structure (*e.g.* tasks), middleware stack (worker threads), physical processing components (processors), and the visualization (node-links). However, aggregation can remove vital diagnosis data when applied aggressively across several dimensions. Isaacs *et al.* [94] recognize the balance between aggregation aggressiveness and information preservation as an important challenge. Our method strives to maintain this balance by reducing the size of the rendered graph and focusing it on problematic sections, while keeping the expected fork-join perspective.

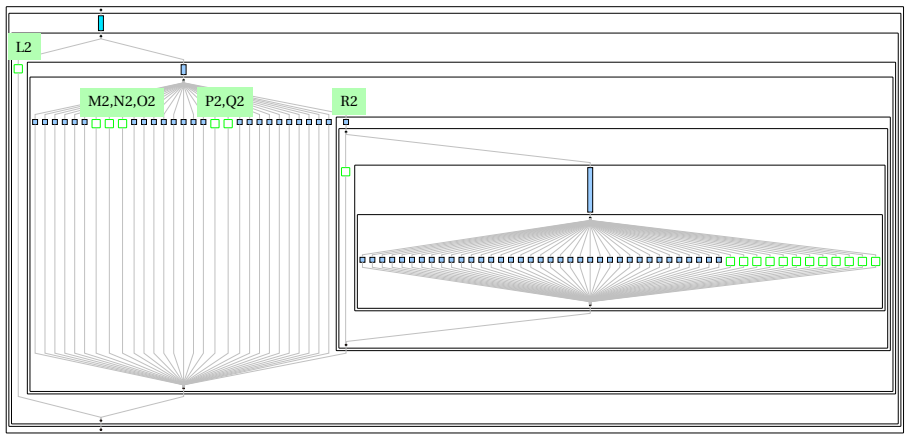
For space reasons, we restrict the discussion to abstraction-centric, logical-time aggregated visualizations similar to grain graphs, and refer readers for other visualizations to recent surveys [94, 220] and a visualization explorer [100].

The dominant aggregation scheme in visualizations is statistical rather than visual, *i.e.*, metrics of selected elements in the main visualization are aggregated statistically and reported separately, typically as a property table [30, 14, 200, 23, 79, 62]. The cognitive load of the main visualization is only reduced by zooming out to focus on large elements, while support for visual aggregation at the same zoom level is absent. Consequently, such visualizations suffer similar navigation and diagnosis difficulties as large unaggregated grain graphs.

The aggregation method for task graphs in *DAGViz* [90] resembles our work. It presents programmers with a single aggregated node that can be interactively opened to reveal subgraphs as well as a dedicated viewer. However, our approach is tailored to grain graphs and is unique in tracing the critical path and



(a)



(b)

Figure D1.6.: Finding dissimilar sections in grain graphs from two independent executions of the non-deterministic Floorplan program from BOTS for input `cell-file=input.5, cutoff=5`. (a) Graph produced from execution on 4 cores has 7974 grains. (b) Graph produced from execution on 48 cores has 3190 grains. The similarity metric allows programmers to understand without inspection that groups L1-2, M1-2, N1-2, and O1-2 have the same structure but P1-2, Q1-2, and R1-2 do not. Groups R1-2 are opened to show the dissimilarity. R2 encloses fewer subgroups than R1.

identifying the similarity of subgraphs. Unaggregated grain graphs are more effective in pinpointing problems than unaggregated DAGViz graphs due to more derived metrics. The expansion of DAGViz graphs results also in the rendering of more nodes as they show a fork-node per grain. Grain graphs avoid this thanks to fork-node reductions that produce a fork-node per set of siblings. DAGViz combats the scaling problem by using an elegant aggregation method that reduces subgraphs that executed wholly on a single worker-thread into a single, non-collapsible node.

ThreadScope [225] visualizes the logical-time structure of task-parallel programs. Its memory operations nodes can be grouped to improve clarity, but it is unclear whether programmers can interact with groups to uncover members.

The *causality graph* [235] visualization permits programmers to manually select and repeatedly aggregate nodes into *supernodes*, while special care must be taken to avoid graph cycles on their creation. Supernode metrics include the local critical path and metrics computed using user-defined combinators. The causality graph presents an unaggregated graph by default, while we present a fully aggregated graph and use sensible aggregation metrics to guide programmers.

D1.7. Conclusion

This paper contributes an aggregation method for grain graphs that enables programmers to easily understand problems in highly-parallel OpenMP programs. Our method groups nodes arranged in recurring patterns to produce an aggregated graph that programmers can navigate by progressively opening and closing groups. Problematic groups are highlighted and non-problematic sections are cleared from sight, enabling focus without compromising the fork-join perspective expected by programmers. Using standard OpenMP programs as examples, we demonstrate a significant reduction of visible nodes throughout problem diagnosis. For future work, we plan to implement a dedicated grain graph viewer that smoothly and precisely guides programmers towards OpenMP problems and hints at solutions.

Acknowledgment

The paper was funded by the TULIPP project (grant number 688403) and the READEX project (grant number 671657) from the EU Horizon 2020 Research and Innovation programme. The authors thank NTNU colleagues Peder Voldnes Langdal, Magnus Själander, Jan Christian Meyer, and Magnus Jahre for constructive comments and KTH Royal Institute of Technology for providing test machinery.

Bibliography

[1] Advanced Micro Devices. ATI Intermediate Language (IL) Specification v2.4, 2011.

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 248–259. ACM, 2000.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[4] Frances E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19. ACM, 1970.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–11. ACM, 1988.

[6] Saman Amarasinghe. The Looming Software Crisis due to the Multicore Menace. `http://groups.csail.mit.edu/commit/papers/06/MulticoreMenace.pdf`, 2006.

[7] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS)*, pages 483–485. ACM, 1967.

[8] Zahira Ammarguellat. A Control-Flow Normalization Algorithm and Its Complexity. *IEEE Transactions on Software Engineering*, 18:237–251, 1992.

[9] Jayvant Anantpur and Govindarajan R. Taming Control Divergence in GPUs through Control Flow Linearization. In *Proceedings of the International Conference on Compiler Construction*, pages 133–153. Springer, 2014.

[10] A. A. Aqrawi and A. C. Elster. Bandwidth Reduction through Multithreaded Compression of Seismic Images. In *Proceedings of International Symposium on Parallel and Distributed Processing Workshops (IPDPSW) and Phd Forum*, pages 1730–1739. IEEE, 2011.

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, Technical Report, UC Berkeley, 2006.

[12] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno. CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search. In *Proceedings of International Parallel Distributed Processing Symposium Workshops (IPDPSW)*, pages 228–235, 2014.

[13] Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer. Perfect Reconstructability of Control Flow from Demand Dependence Graphs. *ACM Transactions on Architecture and Code Optimization*, 11(4):66:1–66:25, 2015.

[14] Barcelona Supercomputing Center. OmpSs task dependency graph, 2013. http://pm.bsc.es/ompss-docs/user-guide/run-programs-plugin-instrument-tdg.html. Accessed 10 April 2015.

[15] Luiz André Barroso. The Price of Performance. *Queue*, 3(7):48–53, 2005.

[16] Oliver Bastert and Christian Matuszewski. *Drawing Graphs: Methods and Models*. Springer, 2001.

[17] W. Baxter and H. R. Bauer, III. The Program Dependence Graph and Vectorization. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–11. ACM, 1989.

[18] Cyrus Bazeghi, Francisco J. Mesa-Martinez, and Jose Renau. uComplexity: Estimating Processor Design Effort. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 209–218. IEEE, 2005.

[19] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk. A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 8–15, 2012.

[20] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: A Verifier for GPU Kernels. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Language and Applications*, pages 113–132. ACM, 2012.

[21] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic Intra-Register Vectorization for the Intel Architecture. *International Journal of Parallel Programming*, 30(2), 2002.

[22] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of Thread-level Parallelism in Desktop Applications. In *Proceedings of the International Symposium on Computer Architecture*, pages 302–313. ACM, 2010.

[23] Wolfgang Blochinger, Michael Kaufmann, and Martin Siebenhaller. Visualizing Structural Properties of Irregular Parallel Computations. In *Proceedings of the ACM Symposium on Software Visualization*, pages 125–134. ACM, 2005.

[24] D. Blythe. Rise of the Graphics Processor. *Proceedings of the IEEE*, 96(5):761–778, 2008.

[25] Corrado Böhm and Giuseppe Jacopini. Flow Diagrams, Turing Machines And Languages With Only Two Formation Rules. *Communications of the ACM*, 9(5):366–371, 1966.

[26] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoît Dupont de Dinechin, and Christophe Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 114–125. IEEE, 2009.

[27] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[28] Ulrik Brandes, Markus Eiglsperger, and Jürgen Lerner. GRAPHML primer, 2017. `http://graphml.graphdrawing.org/primer/graphml-primer.html`. Accessed 27 July 2017.

[29] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software - Practice and Experience*, 28(8):859–881, 1998.

[30] Steffen Brinkmann, José Gracia, and Christoph Niethammer. Task Debugging with TEMANEJO. In *Tools for High Performance Computing 2012*, pages 13–21. Springer, 2013.

[31] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. *Proceedings of the International Symposium on Computer Architecture*, 40(3):49–60, 2012.

[32] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[33] J. Adam Butts and Gurindar S. Sohi. A Static Power Model for Architects. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 191–201. ACM, 2000.

[34] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs. In *Proceedings of the International Symposium on Computer Architecture*, pages 217–228. IEEE, 2014.

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

[36] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 33–36. ACM, 2011.

[37] Larry Carter, Jeanne Ferrante, and Clark D. Thomborson. Folklore confirmed: Reducible flow graphs are exponentially larger. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 106–114. ACM, 2003.

[38] Jeronimo Castrillon, Lothar Thiele, Lars Schorr, Weihua Sheng, Ben Juurlink, Mauricio Alvarez-Mesa, Angela Pohl, Ralph Jessenberger, Victor Reyes, and Rainer Leupers. Multi/many-core programming: Where are we standing? In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1708–1717, 2015.

[39] M. Ceccato, P. Tonella, and C. Matteotti. Goto Elimination Strategies in the Migration of Legacy Code to Java. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 53–62, 2008.

[40] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Workshop on Workload Characterization*, pages 44–54. IEEE, 2009.

[41] Kuan-Hsu Chen, Shen Bor-Yeh, and Yang Wuu. An Automatic Superword Vectorization in LLVM, 2009.

[42] J. Choi, S. Brown, and J. Anderson. From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 270–277, 2013.

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

[44] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 225–236. IEEE, 2010.

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

[46] Cliff Click and Michael Paleczny. A Simple Graph-based Intermediate Representation. In *Proceedings of the Workshop on Intermediate Representations*, pages 35–49. ACM, 1995.

[47] B. Coutinho, D. Sampaio, F.M.Q. Pereira, and W. Meira. Divergence Analysis and Optimizations. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*, pages 320–329, 2011.

[48] Mache Creeger. Multicore CPUs for the Masses. *Queue*, 3(7):64–ff, 2005.

[49] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006.

[50] CUDA C Programming Guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide`. Accessed: 2015-11-02.

[51] Janice E. Cuny, Alfred A. Hough, and Joydip Kundu. Logical Time in Visualizations Produced by Parallel Programs. In *Proceedings of the IEEE Conference on Visualization*, pages 186–193, 1992.

[52] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 25–35. ACM, 1989.

[53] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computational Science and Engineering*, 5(1):46–55, 1998.

[54] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[55] J. B. Dennis. Data Flow Supercomputers. *Computer*, 13(11):48–56, 1980.

[56] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD Re-Convergence at Thread Frontiers. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 477–488. ACM, 2011.

[57] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*, pages 353–364. ACM, 2010.

[58] J. Diaz, C. Muñoz-Caro, and A. Niño. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[59] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Proceedings of the Workshop on General Purpose Processing on Graphs Processing Units*, volume 28, 2007.

[60] R. Dominguez and D.R. Kaeli. Unstructured Control Flow in GPGPU. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshops*, pages 1194–1202, 2013.

[61] Rodrigo Domínguez, Dana Schaa, and David Kaeli. Caracal: Dynamic Translation of Runtime Environments for GPUs. In *Proceedings of the Workshop on General Purpose Processing on Graphs Processing Units*, pages 5:1–5:7. ACM, 2011.

[62] Andi Drebes, Jean-Baptiste Bréjon, Antoniu Pop, Karine Heydemann, and Albert Cohen. Language-centric performance analysis of openmp programs with aftermath. In *OpenMP: Memory, Devices, and Tasks"*, pages 237–250. Springer, 2016.

[63] M. Duranton, D. Black-Schaffer, K. De Boschere, and J. Maebe. The HiPEAC Vision for Advanced Computing in Horizon 2020, 2013.

[64] Marc Duranton, Koen De Boschere, Albert Cohen, Jonas Maebe, and Harm Munk. HiPEAC Vision 2015, 2015.

[65] Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis. In *Proceedings of the International Workshop on OpenMP (IWOMP)*, pages 171–185. Springer, 2013.

[66] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing. In *International Symposium on Graph Drawing*, pages 155–166. Springer, 2004.

[67] A. ElTantawy, J.W. Ma, M. O'Connor, and T.M. Aamodt. A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow. In *Proceedings of the International Symposium High-Performance Computer Architecture*, pages 248–259, Feb 2014.

[68] Ana Erosa and Laurie J. Hendren. Taming Control Flow: A Structured Approach to Eliminating Goto Statements. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 229–240. IEEE, 1994.

[69] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the International Symposium on Computer Architecture*, pages 365–376. ACM, 2011.

[70] Karl-Filip Faxén. Wool-A Work Stealing Library. *SIGARCH Computer Architecture News*, 36(5):93–100, 2009.

[71] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[72] Nicholas J. Fraser, Yaman Umuroglu, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Scaling Binarized Neural Networks on Reconfigurable Logic. In *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM)*, pages 25–30. ACM, 2017.

[73] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 407–420. IEEE, 2007.

[74] Anwar M. Ghuloum and Allan L. Fisher. Flattening and Parallelizing Irregular, Recurrent Loop Nests. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 58–67. ACM, 1995.

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

[76] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM (CACM)*, 31(5):532–533, 1988.

[77] Tianyi David Han and Tarek S. Abdelrahman. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Workshop on General Purpose Processing on Graphs Processing Units*, pages 3:1–3:8. ACM, 2011.

[78] Tianyi David Han and Tarek S. Abdelrahman. Reducing Divergence in GPGPU Programs with Loop Merging. In *Proceedings of the Workshop on General Purpose Processing on Graphs Processing Units*, pages 12–23. ACM, 2013.

[79] Blake Haugen, Stephen Richmond, Jakub Kurzak, Chad A. Steed, and Jack Dongarra. Visualizing Execution Traces with Task Dependencies. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*, pages 2:1–2:8. ACM, 2015.

[80] Paul Havlak. Construction of Thinned Gated Single-Assignment Form. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 477–499. Springer, 1993.

[81] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6th edition, 2017.

[82] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[83] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.

[84] R. Ho, K. W. Mai, and M. A. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[85] Anup Holey, Vineeth Mekkat, and Antonia Zhai. HAccRG: Hardware-Accelerated Data Race Detection in GPUs. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 60–69. IEEE Computer Society, 2013.

[86] Shin Hong and Moonzoo Kim. A Survey of Race Bug Detection Techniques for Multithreaded Programmes. *Software Testing, Verification, and Reliability*, 25(3):191–217, 2015.

[87] S. Horwitz, J. Prins, and T. Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 146–157. ACM, 1988.

[88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46. ACM, 1988.

[89] Ali R. Hurson, Joford T. Lim, Krishna M. Kavi, and Ben Lee. Parallelization of DOALL and DOACROSS Loops - A Survey. *Advances in Computers*, 45:53–103, 1997.

[90] An Huynh, Douglas Thain, Miquel Pericàs, and Kenjiro Taura. DAGViz: A DAG Visualization Tool for Analyzing Task-parallel Program Traces. In *Proceedings of the 2Nd Workshop on Visual Performance Analysis*, pages 3:1–3:8. ACM, 2015.

[91] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly Parallel Programming Models for Thousand-core Microprocessors. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 754–759. ACM, 2007.

[92] Intel Processors and FPGAs - Better Together. `https://itpeernetwork.intel.com/intel-processors-fpga-better-together/`. Accessed: 2018-07-23.

[93] Katherine E. Isaacs, Peer-Timo Bremer, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, and Bernd Hamann. Combing the Communication Hairball: Visualizing Large-Scale Parallel Execution Traces using Logical Time. *IEEE Transactions on Visualization and Computer Graphics*, 20(12), 2014.

[94] Katherine E. Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. State of the Art of Performance Visualization. In *EuroVis - STARs*. The Eurographics Association, 2014.

[95] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Transactions on Programming Languages and Systems*, 19(6):1031–1052, 1997.

[96] Nicklas Bo Jensen and Sven Karlsson. Improving Loop Dependence Analysis. *ACM Transactions on Architecture and Code Optimization*, 14(3):22:1–22:24, 2017.

[97] Neil Johnson and Alan Mycroft. Combined Code Motion and Register Allocation Using the Value State Dependence Graph. In *Proceedings of the International Conference on Compiler Construction*, pages 1–16. Springer, 2003.

[98] Neil E. Johnson. Code size optimization for embedded processors. Technical report, University of Cambridge, 2004.

[99] Richard Johnson, David Pearson, and Keshav Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–185. ACM, 1994.

[100] Katherine Isaacs. Performance Visualization: Living digital library of State of the Art of Performance Visualization, 2017. `http://cgi.cs.arizona.edu/~kisaacs/STAR/`. Accessed 31 July 2017.

[101] W. Kim and M. Voss. Multicore Desktop Programming with Intel Threading Building Blocks. *IEEE Software*, 28(1):23–31, 2011.

[102] Huawei announces the Kirin 970 - new flagship SoC with AI capabilities. `https://www.androidauthority.com/huawei-announces-kirin-970-797788/`. Accessed: 2018-07-23.

[103] Peter M. W. Knijnenburg. Flattening VLIW code generation for imperfectly nested loops, 1998.

[104] Øystein E. Krog and Anne C. Elster. Fast GPU-Based Fluid Simulations Using SPH. In *Proceedings of the International Workshop on Applied Parallel Computing (PARA)*, pages 98–109. Springer, 2012.

[105] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the International Symposium on Computer Architecture*, pages 46–57. ACM, 1992.

[106] Peder Voldnes Langdal. Generating Grain Graphs Using the OpenMP Tools API. Technical report, Norwegian University of Science and Technology, 2017.

[107] Peder Voldnes Langdal, Magnus Jahre, and Ananya Muddukrishna. Extending OMPT to Support Grain Graphs. In *IWOMP 2017: Scaling OpenMP for Exascale Performance and Portability*, pages 141–155. Springer, 2017.

[108] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–156. ACM, 2000.

[109] Samuel Larsen, Emmett Witchel, and Saman P. Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*, pages 18–29. IEEE, 2002.

[110] Ahmad Lashgar and Amirali Baniasadi. Performance in GPU Architectures: Potentials and Distances. In *Proceedings of the Workshop on Duplicating, Deconstructing and Debunking*, 2011.

[111] Alan C. Lawrence. Optimizing compilation with the Value State Dependence Graph. Technical report, University of Cambridge, 2007.

[112] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[113] Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Stephen W. Keckler, and Krste Asanović. Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 101–113. IEEE, 2014.

[114] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying GPU Kernels by Test Amplification. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 383–394. ACM, 2012.

[115] J. Liu, J. Wickerson, and G. A. Constantinides. Loop Splitting for Efficient Pipelining in High-Level Synthesis. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 72–79, 2016.

[116] LLVM. `https://bugs.llvm.org/show_bug.cgi?id=31851`, 2018. Accesssed: 2018-05-07.

[117] LLVM. `https://bugs.llvm.org/show_bug.cgi?id=37202`, 2018. Accesssed: 2018-05-07.

[118] LLVM. `https://bugs.llvm.org/show_bug.cgi?id=31183`, 2018. Accesssed: 2018-05-07.

[119] Holger Ludvigsen and Anne Cathrine Elster. Real-Time Ray Tracing Using Nvidia OptiX. In *Eurographics 2010 - Short Papers*. The Eurographics Association, 2010.

[120] Souley Madougou, Ana Varbanescu, Cees de Laat, and Rob van Nieuwpoort. The Landscape of GPGPU Performance Modeling Tools. *Parallel Computing*, 56(C):18–33, 2016.

[121] Souley Madougou, Ana Lucia Varbanescu, Cees de Laat, and Rob van Nieuwpoort. An Empirical Evaluation of GPGPU Performance Models. In *Euro-Par 2014: Parallel Processing Workshops*, pages 165–176. Springer, 2014.

[122] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. . W. Hwu. Characterizing the Impact of Predicated Execution on Branch Prediction. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 217–227, 1994.

[123] Jonathan Mak and Alan Mycroft. Limits of Parallelism Using Dynamic Dependency Graphs. In *Proceedings of the Seventh International Workshop on Dynamic Analysis (WODA)*, pages 42–48. ACM, 2009.

[124] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An Evaluation of Vectorizing Compilers. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*, pages 372–382. IEEE, 2011.

[125] Naraig Manjikian and Tarek S Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, 1997.

[126] Sally A. McKee. Reflections on the Memory Wall. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 162–. ACM, 2004.

[127] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* 2010.

[128] Intel Math Kernel Library. `https://software.intel.com/en-us/mkl`. Accessed: 2018-07-21.

[129] Moto X to feature standalone language processing and context awareness chips, and more rumors. `https://www.androidauthority.com/moto-x-language-processing-chip-238522/`. Accessed: 2018-07-23.

[130] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[131] Ananya Muddukrishna. *Improving OpenMP Productivity with Data Locality Optimizations and High-resolution Performance Analysis*. PhD thesis, KTH Royal Institute of Technology, 2016.

[132] Ananya Muddukrishna, Peter A. Jonsson, and Mats Brorsson. Characterizing Task-Based OpenMP Programs. *PLOS ONE*, 10(4):1–29, 2015.

[133] Ananya Muddukrishna, Peter A. Jonsson, and Mats Brorsson. Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors. *Scientific Programming*, 2015:5:5–5:5, 2016.

[134] Ananya Muddukrishna, Peter A. Jonsson, and Peder Langdal. anamud/mir-dev: MIR v1.0.0, March 2017.

[135] Ananya Muddukrishna, Peter A. Jonsson, Artur Podobas, and Mats Brorsson. Grain Graphs: OpenMP Performance Analysis Made Easy. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 28:1–28:13. ACM, 2016.

[136] Ananya Muddukrishna and Peder Langdal. anamud/grain-graphs: Grain Graphs v1.0.0, March 2017.

[137] Dheya Mustafa and Rudolf Eigenmann. PETRA: Performance Evaluation Tool for Modern Parallelizing Compilers. *International Journal of Parallel Programming*, 43(4):549–571, 2015.

[138] Dorit Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004.

[139] R. Namballa, N. Ranganathan, and A. Ejnioui. Control and Data Flow Graph Extraction for High-Level Synthesis. In *IEEE Computer Society Annual Symposium on VLSI*, pages 187–192, 2004.

[140] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[141] Dorit Nuzman and Richard Henderson. Multi-platform Auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 281–294. IEEE, 2006.

[142] NVIDIA. NVIDIA Tesla V100 GPU Architecture - The world's most advanced data center GPU. `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`. Accessed: 2018-08-23.

[143] Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. Characterizing and Mitigating Work Time Inflation in Task Parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 65:1–65:12. IEEE, 2012.

[144] Kunle Olukotun and Lance Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, 2005.

[145] OpenACC. `www.openacc.org`. Accessed: 2018-08-14.

[146] OpenCL - The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl`. Accessed: 2015-11-02.

[147] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 257–271. ACM, 1990.

[148] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*, pages 105–118. IEEE, 2005.

[149] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the International Symposium on Computer Architecture*, pages 206–218. ACM, 1997.

[150] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpotTM Server Compiler. In *Proceedings of the JavaTM Virtual Machine Research and Technology Symposium*, pages 1–1. USENIX Association, 2001.

[151] S.J. Pennycook, J.D. Sewall, and V.W. Lee. Implications of a metric for performance portability. *Future Generation Computer Systems*, 2017.

[152] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Proceedings of the*

Conference on Field Programmable Logic and Applications, pages 1–4. IEEE, 2013.

[153] Angela Pohl, Biagio Cosenza, and Ben Juurlink. Correlating Cost with Performance in LLVM. In *Proceedings of the 13th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, 2017.

[154] Angela Pohl, Biagio Cosenza, and Ben Juurlink. Control Flow Vectorization for ARM NEON. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 66–75. ACM, 2018.

[155] Angela Pohl, Biagio Cosenza, and Bin Juurlink. Cost Modelling for Vectorization on ARM. In *Proceedings of International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018.

[156] Vasileios Porpodas and Timothy M. Jones. Throttling Automatic Vectorization: When Less is More. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*, pages 432–444. IEEE, 2015.

[157] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. PSLP: Padded SLP Automatic Vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 190–201. IEEE, 2015.

[158] Louis-Noël Pouchet. Polybench/C 4.2. `http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`, 2017. Accessed: 2017-12-13.

[159] Parallel Thread Execution ISA Version 4.3. `http://docs.nvidia.com/cuda/parallel-thread-execution`. Accessed: 2015-11-02.

[160] Pixel Visual Core: A closer look at the Pixel 2's hidden chip. `https://www.androidauthority.com/pixel-visual-core-808182/`. Accessed: 2018-07-23.

[161] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage Decoupled Software Pipelining. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 114–123. ACM, 2008.

[162] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. Performance Scalability of Decoupled Software Pipelining. *ACM Transactions on Architecture and Code Optimization*, 5(2):8:1–8:25, 2008.

[163] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the International Symposium High-Performance Computer Architecture*, pages 13–24. IEEE, 2007.

[164] Nico Reissmann. Utilizing the Value State Dependence Graph for Haskell. Technical report, University of Gothenburg, 2012.

[165] Nico Reissmann. jlm. `https://github.com/phate/jlm`, 2017. Accessed: 2017-12-13.

[166] Nico Reissmann. phate/ggraph: Vpa17, July 2017.

[167] Nico Reissmann, Thomas L. Falch, Benjamin A. Bjornseth, Helge Bahmann, Jan Christian Meyer, and Magnus Jahre. Efficient Control Flow Restructuring for GPUs. In *International Conference on High Performance Computing & Simulation (HPCS)*, pages 48–57, 2016.

[168] Nico Reissmann, Magnus Jahre, and Ananya Muddukrishma. Aggregating Large Grain Graphs For Improved OpenMP Productivity, 2017. Fourth International Workshop on Visual Performance Analysis (VPA).

[169] Nico Reissmann and Ananya Muddukrishna. Diagnosing Highly-Parallel OpenMP Programs with Aggregated Grain Graphs. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par)*, pages 106–119. Springer, 2018.

[170] A. D. Robison. Composable Parallel Patterns with Intel Cilk Plus. *Computing in Science and Engineering*, 15(2):66–71, 2013.

[171] R. Ronen, A. Mendelson, K. Lai, Shih-Lien Lu, F. Pollack, and J. P. Shen. Coming Challenges in Microarchitecture and Architecture. *Proceedings of the IEEE*, 89(3):325–340, 2001.

[172] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 12–27. ACM, 1988.

[173] Ira Rosen, Dorit Nuzman, and Ayal Zaks. Loop-Aware SLP in GCC. In *Proceedings of the GCC Developers Summit*, pages 131–142, 2007.

[174] Radu Rugina and Martin C. Rinard. Recursion Unrolling for Divide and Conquer Programs. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 34–48. Springer, 2001.

[175] K. Rupp. 42 years of microprocessor trend data. `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`. Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten, New plot and data collected for 2010-2017 by K. Rupp, Accessed: 2018-07-18.

[176] Ingar Saltvik, Anne C. Elster, and Henrik R. Nagel. Parallel Methods for Real-Time Visualization of Snow. In *Proceedings of the International Workshop on Applied Parallel Computing (PARA)*, pages 218–227. Springer, 2007.

[177] Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. Divergence Analysis. *ACM Transactions on Programming Languages and Systems*, 35(4):13:1–13:36, 2014.

[178] V. Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Journal of Research and Development*, 35(5-6):779–804, 1991.

[179] J. Sartori and R. Kumar. Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications. *IEEE Transactions on Multimedia*, 15(2):279–290, 2013.

[180] Robert R Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[181] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–265. ACM, 2017.

[182] Dirk Schmidl, Christian Terboven, Dieter an Mey, and Matthias S. Müller. Suitability of Performance Tools for OpenMP Task-Parallel Programs. In *Tools for High Performance Computing 2013*, pages 25–37. Springer, 2014.

[183] M. Sharir. Structural Analysis: A new Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, 5(3-4):141–153, 1980.

[184] David E. Shaw, Martin M. Deneroff, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, Kevin J. Bowers, Jack C. Chao, Michael P. Eastwood, Joseph Gagliardo, J. P. Grossman, C. Richard Ho, Douglas J. Ierardi, István Kolossváry, John L. Klepeis, Timothy Layman, Christine McLeavey, Mark A. Moraes, Rolf Mueller, Edward C. Priest, Yibing Shan, Jochen Spengler, Michael Theobald, Brian Towles, and Stanley C. Wang. Anton, a Special-Purpose Machine for Molecular Dynamics Simulation. In *Proceedings of the International Symposium on Computer Architecture*, pages 1–12. ACM, 2007.

[185] David E. Shaw, J. P. Grossman, Joseph A. Bank, Brannon Batson, J. Adam Butts, Jack C. Chao, Martin M. Deneroff, Ron O. Dror, Amos Even, Christopher H. Fenton, Anthony Forte, Joseph Gagliardo, Gennette Gill, Brian Greskamp, C. Richard Ho, Douglas J. Ierardi, Lev Iserovich, Jeffrey S. Kuskin, Richard H. Larson, Timothy Layman, Li-Siang Lee, Adam K. Lerer, Chester Li, Daniel Killebrew, Kenneth M. Mackenzie, Shark Yeuk-Hai Mok, Mark A. Moraes, Rolf Mueller, Lawrence J. Nociolo, Jon L. Peticolas, Terry Quan, Daniel Ramot, John K. Salmon, Daniele P. Scarpazza, U. Ben Schafer, Naseer Siddique, Christopher W. Snyder, Jochen Spengler, Ping Tak Peter Tang, Michael Theobald, Horia Toma, Brian Towles, Benjamin Vitale, Stanley C. Wang, and Cliff Young. Anton 2: Raising the Bar for Performance and Programmability in a Special-purpose Molecular Dynamics Supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 41–53. IEEE, 2014.

[186] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

[187] Jaewook Shin. Introducing Control Flow into Vectorized Code. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*, pages 280–291. IEEE, 2007.

[188] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*, pages 45–55. IEEE, 2002.

[189] Jaewook Shin, Mary Hall, and Jacqueline Chame. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 165–175. IEEE, 2005.

[190] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. *Microprocessors and Microsystems*, 33(4):235 – 243, 2009.

[191] Gaurav Singh, Sumit Gupta, Sandeep Shukla, Rajesh Gupta, and San Deigo. High-Level Synthesis: A Code Transformational Approach to High-Level Synthesis, 2006.

[192] Erik Smistad, Mohammadmehdi Bozorgi, and Frank Lindseth. Fast: framework for heterogeneous medical image computing and visualization. *International Journal of Computer Assisted Radiology and Surgery*, 10(11):1811–1822, 2015.

[193] Erik Smistad, Anne C. Elster, and Frank Lindseth. Gpu accelerated segmentation and centerline extraction of tubular structures from medical images. *International Journal of Computer Assisted Radiology and Surgery*, 9(4):561–575, 2014.

[194] Erik Smistad, Thomas L. Falch, Mohammadmehdi Bozorgi, Anne C. Elster, and Frank Lindseth. Medical image segmentation on GPUs - A comprehensive review. *Medical Image Analysis*, 20(1):1–18, 2015.

[195] Daniele G. Spampinato and Anne C. Elster. Linear optimization on modern GPUs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2009.

[196] James Stanier. *Removing and Restoring Control Flow with the Value State Dependence Graph*. PhD thesis, University of Sussex, 2012.

[197] James Stanier and Alan Lawrence. The Value State Dependence Graph Revisited. In *Proceedings of the Workshop on Intermediate Representations*, pages 53–60, 2011.

[198] James Stanier and Des Watson. A study of irreducibility in C programs. *Software: Practice and Expererience*, 2011.

[199] James Stanier and Des Watson. Intermediate Representations in Imperative Compilers: A Survey. *ACM Computing Surveys (CSUR)*, 45(3):26:1–26:27, 2013.

[200] Vladimir Subotic, Steffen Brinkmann, Vladimir Marjanovic, Rosa M. Badia, Jose Gracia, Christoph Niethammer, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Programmability and portability for exascale: Top down programming methodology and tools with StarSs. *Journal of Computational Science*, 4(6):450 – 456, 2013.

[201] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[202] Mahito Sugiyama, M. Elisabetta Ghisu, Felipe Llinares-López, and Karsten Borgwardt. graphkernels: R and python packages for graph comparison. *Bioinformatics*, 34(3):530–532, 2018.

[203] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[204] Herb Sutter and James Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, 2005.

[205] SystemVerilog. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[206] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[207] Scott Thompson, Paul Packan, and Mark Bohr. MOS Scaling: Transistor Challenges for the 21st Century. *Intel Technology Journal*, 1998.

[208] TOP500 June 2018 Supercomputer List. `https://www.top500.org/lists/2018/06/`. Accessed: 2018-07-19.

[209] Arm's new chips will bring on-device AI to millions of smartphones. `https://www.androidauthority.com/arm-unveils-new-npu-837015/`. Accessed: 2018-07-23.

[210] Jessica H. Tseng and Krste Asanović. Banked Multiported Register Files for High-frequency Superscalar Microprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 62–71. ACM, 2003.

[211] Peng Tu and David Padua. Efficient Building and Placing of Gating Functions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–55. ACM, 1995.

[212] Gary Tyson and Matthew Farrens. Evaluating the Effects of Predicated Execution on Branch Prediction. *International Journal of Parallel Programming*, 24(2):159–186, 1996.

[213] Y. Umuroglu and M. Jahre. An Energy Efficient Column-Major Backend for FPGA SpMV Accelerators. In *Proceedings of the IEEE International Conference Computer Design*, pages 432–439, 2014.

[214] Y. Umuroglu, D. Morrison, and M. Jahre. Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform. In *Proceedings of the Conference on Field Programmable Logic and Applications*, pages 1–8, 2015.

[215] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 65–74. ACM, 2017.

[216] Yaman Umuroglu and Magnus Jahre. A Vector Caching Scheme for Streaming FPGA SpMV Accelerators. In *Proceedings of Applied Reconfigurable Computing (ARC)*, pages 15–26. Springer, 2015.

[217] Sebastian Unger and Frank Mueller. Handling Irreducible Loops: Optimized Node Splitting Versus DJ-graphs. *ACM Transactions on Programming Languages and Systems*, 24(4):299–333, 2002.

[218] VHDL. IEC/IEEE International Standard - Behavioural languages - Part 1-1: VHDL Language Reference Manual. *IEC 61691-1-1:2011(E) IEEE Std 1076-2008*, pages 1–648, 2011.

[219] NVIDIA Volta Unveiled: GV100 GPU and Tesla V100 Accelerator Announced. `https://www.anandtech.com/show/11367/nvidia-volta-unveiled-gv100-gpu-and-tesla-v100-accelerator-announced`. Accessed: 2018-08-23.

[220] Tatiana Von Landesberger, Arjan Kuijper, Tobias Schreck, Jörn Kohlhammer, Jarke J van Wijk, J-D Fekete, and Dieter W Fellner. Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges. In *Computer Graphics Forum*, pages 1719–1749. John Wiley & Sons, 2011.

[221] David W. Wall. Limits of Instruction-level Parallelism. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, pages 176–188. ACM, 1991.

[222] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F. P. O'Boyle. Integrating Profile-Driven Parallelism Detection and Machine-Learning-Based Mapping. *ACM Transactions on Architecture and Code Optimization*, 11(1):2:1–2:26, 2014.

[223] Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[224] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value Dependence Graphs: Representation Without Taxation. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 297–310. ACM, 1994.

[225] Kyle B Wheeler and Douglas Thain. Visualizing Massively Multithreaded Applications with ThreadScope. *Concurrency and Computation: Practice and Experience*, 22(1):45–67, 2010.

[226] Haicheng Wu, Gregory Diamos, Si Li, and Sudhakar Yalamanchili. Characterization and Transformation of Unstructured Control Flow in GPU Applications. In *1st International Workshop on Characterizing Applications for Heterogeneous Exascale Systems*, 2011.

[227] Haicheng Wu, Gregory Diamos, Jin Wang, Si Li, and Sudhakar Yalamanchili. Characterization and transformation of unstructured control flow in bulk synchronous GPU applications. *The International Journal of High Performance Computing Applications*, 26(2):170–185, 2012.

[228] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[229] Rengan Xu, Sunita Chandrasekaran, Barbara Chapman, and Christoph F Eick. Directive-based Programming Models for Scientific Applications - A Comparison. *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, pages 1–9, 2012.

[230] B. Ylvisaker, C. Ebeling, and S. Hauck. Enhanced Loop Flattening for Software Pipelining of Arbitrary Loop Nests, 2010.

[231] Richard M. Yoo, Christopher J. Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. Locality-aware Task Management for Unstructured Parallelism: A Quantitative Limit Study. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 315–325. ACM, 2013.

[232] yWorks GmBh. yEd graph editor, 2015. `http://www.yworks.com/en/products_yed_about.html`. Accessed 10 April 2015.

[233] Ali Mustafa Zaidi. Accelerating control-flow intensive code in spatial hardware. Technical report, University of Cambridge, 2015.

[234] Ali Mustafa Zaidi and David Greaves. Value State Flow Graph: A Dataflow Compiler IR for Accelerating Control-Intensive Code in Spatial Hardware. *ACM Transactions on Reconfigurable Technology and Systems*, 9(2):14:1–14:22, 2015.

[235] Dror Zernik, Marc Snir, and Dalia Malki. Using Visualization Tools to Understand Concurrency. *IEEE Software*, 9(3):87–92, 1992.

[236] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, pages 369–380. ACM, 2011.

[237] F. Zhang and E.H. D'Hollander. Using Hammock Graphs to Structure Programs. *IEEE Transactions on Software Engineering*, 30(4):231–245, 2004.

[238] Fubo Zhang and E.H. D'Hollander. Extracting the Parallelism in Program with Unstructured Control Statements. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 264–270, 1994.

[239] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):104–115, 2014.

[240] V. Zyuban and P. Kogge. The Energy Complexity of Register Files. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 305–310. ACM, 1998.