

**MELHORANDO O ALCANCE DE OTIMIZAÇÕES
DE CÓDIGO ATRAVÉS DE ANÁLISE HÍBRIDA
DE INTERVALOS DE ACESSO À MEMÓRIA**

PÉRICLES RAFAEL OLIVEIRA ALVES

**MELHORANDO O ALCANCE DE OTIMIZAÇÕES
DE CÓDIGO ATRAVÉS DE ANÁLISE HÍBRIDA
DE INTERVALOS DE ACESSO À MEMÓRIA**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte
Dezembro de 2016

PÉRICLES RAFAEL OLIVEIRA ALVES

**ENABLING CODE OPTIMIZATIONS THROUGH
HYBRID ANALYSIS OF MEMORY ACCESS
RANGES**

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

December 2016

Ficha catalográfica elaborada pela Biblioteca do IEx - UFMG

Alves, Péricles Rafael Oliveira.

A474e Enabling code optimizations through hybrid analysis of
memory access ranges. / Péricles Rafael Oliveira Alves. –
Belo Horizonte, 2017.
xxiv, 74 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de
Minas Gerais – Departamento de Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira.

1. Computação - Teses. 2. Compiladores (programação
de computador). 3. Linguagem de programação
(Computadores). I. Orientador. II. Título.

CDU 519.6*32 (043)




UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


FOLHA DE APROVAÇÃO

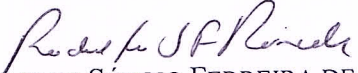
Enabling code optimizations through hybrid analysis of memory access ranges

PERICLES RAFAEL OLIVEIRA ALVES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG


PROF. ROBERTO IERUSALIMSCHY
Departamento de Informática - PUC-Rio


PROF. RODOLFO SÉRGIO FERREIRA DE RESENDE
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 03 de fevereiro de 2017.

To my parents

Acknowledgments

First and foremost I would like to thank Eduardo Figueiredo and Fernando Pereira. Eduardo accepted me in his group with open arms and taught me the very basics of scientific investigation and writing. Fernando took me in six years ago as his apprentice and by means of great advising, motivation, and friendship made a (*soon-to-be*) master of science out of me. I would also like to acknowledge the work of those who made this dissertation possible from a technical point of view: Gleison and Breno, who worked so diligently in the design and implementation of DawnCC. Tobias Grosser, who provided the necessary support for us to use Polly and is one of the authors of the polyhedral memory disambiguation method presented in Chapter 3. Fabian Gruber, who proposed and implemented the dynamic disambiguation method also described in Chapter 3. Michael Frank and Fabricio Ferracioli from LGE, who provided many useful insights about our work. I would like to thank all the friends I made in the Compilers Laboratory over these last years, who made the lab a fun and cool environment to work on and from whom I learned so much. I also thank CAPES, FAPEMIG, CNPq, and LGE Brazil for funding my scientific endeavors.

From a personal side, I thank my parents and my family for supporting all my decisions, even when they did not understand why I spent days in my room staring at those weird-looking lines of code in a computer screen. Finally, I would like to thank the friends I made in the Computer Science department throughout the years, who proved more than once to be the greatest allies one can ask for in life, especially Rubens, Douglas, Henrique, Demontiê, Victor, Luis, Thiago, Larissa, Mivian, Marcos, and Jonathan.

Resumo

Compiladores estáticos atuais implementam uma gama de otimizações de código que utilizam informações sobre memória para gerar código eficiente. A eficácia destas transformações, entretanto, é limitada pela imprecisão inerente de métodos estáticos projetados para extrair *insights* úteis relacionados ao uso da memória, como análise de *aliases* e inferência de tamanho de arranjos. Para tratar este problema, apresentamos uma análise simbólica que combina informações estáticas e dinâmicas para inferir limites para operações de acesso à memória. Nós mostramos que nosso método é preciso, sendo capaz de derivar limites simbólicos para 98% dos acessos à memória em PolyBench, impondo um *overhead* insignificante em tempo de execução. Para mostrar que nossa análise pode ser utilizada tanto para aumentar o alcance de transformações de código existentes quanto possibilitar a implementação de novas otimizações mais agressivas, nós apresentamos dois clientes distintos. O primeiro, uma técnica híbrida para desambiguação de apontadores, utiliza nossa análise de intervalos de acesso para prover ao compilador informações mais precisas sobre *aliasing*. Esta melhora na precisão nos permite gerar binários que são 10% mais rápidos quando comparados ao maior nível de otimização disponível no LLVM, um compilador de amplo uso na indústria. O segundo é um arcabouço para anotação automática de código para execução na GPU, que utiliza nossa inferência de intervalos de memória para gerar diretivas capazes de transferir dados para o dispositivo externo. As anotações geradas por este arcabouço, que chamamos de DawnCC, permitem *speedups* de mais de 100x em uma arquitetura Nvidia e mais de 50x em uma arquitetura ARM.

Abstract

Current static compilers implement a number of code optimizations that rely on precise memory-related information to generate efficient code. The effectiveness of such transformations, however, is bound by the inherent imprecision of static methods designed to extract useful memory insights, such as alias analyses and array size inference. To address this problem, we present a symbolic analysis that combines static and dynamic information to infer access bounds for memory operations. We show that our method is precise, being able to derive symbolic bounds for 98% of the memory accesses in PolyBench, while imposing negligible runtime overhead. To show that our analysis can be used to both improve the reach of current code transformations and enable new, more aggressive optimizations, we present two distinct clients. The first, a hybrid pointer disambiguation technique, uses our access range analysis to provide the compiler with more precise alias information. This improvement in precision allows us to generate binaries that are 10% faster when compared to the highest optimization level available in LLVM, a industrial-strength compiler. The second is a framework that automatically annotates code for GPU execution, using our memory range inference to generate directives capable of transferring data to the external device. The annotations generated by this framework, which we call DawnCC, lead to speedups of over 100x in an Nvidia architecture, and over 50x in an ARM architecture.

List of Figures

2.1	The syntax of our core language. We let $n \in \mathbb{N}$	8
2.2	(a) example program adapted from Chabbi and Mellor-Crummey [2012] and (b) a version of it re-written in our core language. The symbol \bullet denotes an unknown input. We have eliminated the control flow in Figure (b), as it has no influence in our analysis.	9
2.3	Abstract interpretation of arithmetic expressions in our core language. We let $\text{MaxTp}_s(v)$, $\text{MinTp}_s(v)$ be the maximum and minimum values of variable v 's type.	10
3.1	An example (<code>array_sum_1</code>) in which potential aliasing hinders compiler optimizations such as loop invariant code motion as done in <code>array_sum_2</code>	14
3.2	Modified version of <code>array_sum_1</code> in which the compiler generates checks to disambiguate pointers.	15
3.3	Modified version of <code>array_sum_1</code> in which the program can inspect the size of the region referenced by a pointer. Here <code>sizeof(int)</code> is considered to be 4.	16
3.4	Example adapted from Chabbi and Mellor-Crummey [2012]. This program contains code that is difficult to optimize due to aliasing.	17
3.5	Program of Figure 3.4 with Single-Entry, Single-Exit regions highlighted.	18
3.6	Program after instrumentation.	19
3.7	Program that illustrates over-approximation of our purely dynamic approach.	19
3.8	(a) Structure of the red-black nodes that we use to track memory size. (b) Dynamic view of the red-black tree when the program flow reaches line 5 of Figure 3.6.	20
3.9	Overview of the code generation methodology used in Section 3.3.	22
3.10	Generation of dynamic pointer checks.	25

3.11	Relative increase in the number of loops within Static Control Parts when using our disambiguation approaches. The base values are the number of loops optimized when using only LLVM’s static alias analyses. The three approaches provide the same increase in almost all benchmarks.	29
3.12	Execution time overhead of our hybrid approaches when compared to the use of “restrict” in Polly. Our biggest overhead was in <code>gramschmidt</code> , which executes in 5.19 seconds in its “restrict” version. When running on Polly augmented with our polyhedral approach, this number grows to 7.86 seconds, being 1.5x slower, as the chart shows.	30
3.13	Execution time improvement of our hybrid techniques when applied to LLVM-O3. Our biggest speedup was in <code>bicg</code> , which is executed in 150 milliseconds by LLVM-O3. Both of our hybrid approaches reduce this number to 56 milliseconds, making it 2.7x faster, as the Figure shows.	30
3.14	Runtime improvement of our symbolic approach (Chapter 2) over LLVM-O3, when followed by a round of loop invariant code motion. As in Figure 3.13, our biggest speedup was in <code>bicg</code> , being 2.7x faster.	32
3.15	Execution time improvement of our hybrid methods over Polly, running on LLVM-O3. Our best result can be observed in <code>covariance</code> , which is executed in 4.29 seconds by Polly. Our polyhedral approach lowers this number to 1.29 seconds, being 3.3x faster.	33
3.16	Code size expansion due to our optimization.	34
3.17	Execution time of our purely dynamic (Section 3.2) and combined (dynamic + hybrid) approaches when compared to our hybrid technique (Chapter 2). <code>ludcmp</code> executes in 1.27 seconds when using our hybrid method. The purely dynamic approach reduces this number to 1.00 second, being 1.2x faster, as the Figure shows.	34
3.18	Examples of programs that illustrate advantages and disadvantages of our different runtime pointer disambiguation techniques.	36
4.1	(a) Standard C implementation of the Single Precision $AX + Y$ (SAXPY) kernel. (b) Same algorithm written in C for CUDA.	40
4.2	(a) SAXPY annotated with OpenACC pragmas. (b) SAXPY annotated with OpenMP pragmas. The gray area denotes code created automatically.	40
4.3	Overview of DawnCC.	42
4.4	(a) example adapted from our original SAXPY code (Figure 4.1 (a)) and (b) a version of it annotated with OpenMP 4.0 pragmas to run on a GPU.	43

4.5	(a) LLVM instructions generated for the body of the loop in line 3 of Figure 4.4 (a); (b) LLVM assembly representing the symbolic access bounds of arrays x and y	44
4.6	(a) C code generated for the symbolic bounds in Figure 4.5 (b); (b) steps used to improve the readability of expressions that describes upper bound of x	44
4.7	(a) example program adapted from the <i>correlation</i> benchmark, (b) code produced following a simple per-loop annotation approach, (c) code produced using the <i>data environment</i> feature of OpenMP 4.0, and (d) CFG for the program divided into SESE regions.	47
4.8	(a) Example program adapted from the <i>fdtd_2d</i> benchmark, (b) SESE regions in the program. (c) Unsafe annotations.	50
4.9	(a) Annotations produced by DawnCC for program in Figure 4.8 (a), (b) <i>scope tree</i>	51
4.10	Desktop (DawnCC+pgcc vs pgcc) Speedup due to the annotations inserted by DawnCC, compared to execution of sequential code on the Desktop setup. Both programs, original and annotated, have been compiled with pgcc. Y-axis show speedup, in number of times. The higher the bar, the better. Numbers represent absolute runtime, in seconds, of benchmarks with largest inputs, running on the CPU.	54
4.11	Phone (DawnCC+gpuclang vs gpuclang) Comparison between the code that DawnCC has annotated with OpenMP pragmas and compiled with gpuclang, and the benchmarks without the annotations. Y-axis show speedup, in number of times. The higher the bar, the better. Numbers represent absolute runtime, in seconds, of benchmarks with largest inputs, running on the CPU.	54
4.12	Results produced by the copy coalescing optimization of Section 4.3 in the Desktop setup. “Orig.” denotes the original program, and “Opt.” its optimized version. Runtime is the GPU’s, for the largest input size. The bullets in the last two columns indicate benchmarks where no coalescing could be performed.	55
4.13	(a) Performance improvement due to the copy coalescing optimization discussed in Section 4.3. Bars show percentage of speedup of optimized over non-optimized code. The higher the bar, the better. (b) Percentage of copy time saved. X% is the reduction in the time spent copying data. The higher the bar, the better.	56

List of Tables

Contents

Acknowledgments	xi
Resumo	xiii
Abstract	xv
List of Figures	xvii
List of Tables	xxi
1 Introduction	1
1.1 Client I - Runtime Pointer Disambiguation	2
1.2 Client II - Automatic Annotation for Data Parallelism and Offloading	4
2 Symbolic Inference of Memory Access Ranges	7
2.1 The Core Language	7
2.2 Computing Symbolic Variable Bounds	9
2.2.1 Bootstrapping the Code Generator	11
3 Runtime Pointer Disambiguation	13
3.1 Overview	13
3.2 Purely Dynamic Pointer Disambiguation	16
3.2.1 Implementation Details	20
3.3 Hybrid Pointer Disambiguation	21
3.3.1 Building Alias Checks From Symbolic Bounds	23
3.3.2 Polyhedral Access Range Analysis	23
3.3.3 Code Generation	24
3.4 Experiments	26
3.4.1 What can LLVM's Purely Static Approaches Do?	27
3.4.2 Hybrid Approaches	29

3.4.3	The purely dynamic approach	34
3.4.4	Discussion	35
4	DawnCC: Automatic Annotation for Data Parallelism and Offloading	39
4.1	Overview	39
4.2	Static Analyses	41
4.2.1	Memory Disambiguation	41
4.2.2	From Source to IR, and Back Again	42
4.3	Data Transfer Optimizations	46
4.3.1	Deciding which Loops can be Merged into Common Transfer Blocks	49
4.3.2	Carrying on with Coalescing into Source Code	49
4.4	Experiments	52
4.4.1	Runtime Results	53
4.4.2	The Impact of Copy Coalescing	54
4.4.3	DawnCC vs manual code annotation	56
5	Related Work	59
5.1	Literature on Memory Disambiguation	60
5.2	Literature on Automatic Program Annotation and Offloading	62
6	Conclusion	65
6.1	Contributions	65
6.2	Publications and Software	66
6.3	Future Work	67
	Bibliography	69

Chapter 1

Introduction

Code optimizations rely heavily on memory-related information to yield performance improvements. Knowledge of which pointers in a program can access the same regions in memory, for instance, is essential for most loop transformations, automatic parallelization, and any compiler pass that performs instruction reordering. Information about the size of memory regions, as another example, is needed for passes that deal with data placement, such as spatial locality optimizations and automatic generation of code for heterogeneous architectures. Despite its importance, having meta-information about memory available during compilation is the exception rather than the norm for static compilers.

While being the target of extensive research in the last several decades, developing efficient and precise analyses for static compilers capable of extracting useful memory information still represents a great challenge. Checking if two instructions can write to the same place in memory, for instance, is a hard task at compile time. Clang, one of the most used C and C++ compilers nowadays, is not able to detect the absence of dependence for 62% of the alias queries made during the compilation of the PolyBench suite. This number is obtained enabling the five static alias analyses available in the compiler [Alves et al., 2015]. Most of this lack of precision arise from the fact that the compiler must give a conservative answer whenever the same pair of memory pointers may alias or not depending on the execution context. Finding out the size of a memory region, in the other hand, is a difficult problem even at runtime, as languages such as C and C++ do not attach allocation size information to arrays and other structures, contrary to more dynamic languages, like Java, C# or Python. In this work, we show how an inference technique that combines static and dynamic information can be used to tackle these and other problems regarding the precision of memory-related analyses.

In this work, we present a lightweight compiler analysis for C and C++ programs

capable of inferring symbolic access bounds for memory regions. Given a pointer and a target region of code in the input program, our analysis derives a pair of symbolic expressions, denoting the lowest and highest addresses that can be accessed through that pointer within the region. The generated bound expressions are inserted at the entry of the target region. Its symbols, which represent variable names from the original program, are replaced at runtime by their actual values, thus yielding the final integer access bounds. As range expressions are generated at compile time and solved at runtime, we say that this analysis operates in a *hybrid* fashion, rather than being completely static or completely dynamic. We show experimentally that our analysis is able to infer symbolic bounds for 98% of the memory accesses found in the source code of the benchmarks that compose the PolyBench suite.

To demonstrate the usefulness of the symbolic range analysis presented here, we use it to both (i) improve current code optimizations available in industrial-strength static compilers and (ii) enable new, more aggressive optimizations. For the first task, we use our analysis to perform pointer disambiguation, extending the reach of transformations such as loop unrolling and automatic vectorization, implemented in the LLVM compiler infrastructure. We also use the result of the pointer disambiguation technique built around our range inference analysis to boost the optimizations performed by Polly, a polyhedral framework for LLVM. For the second, we describe the implementation of DawnCC, a framework that automatically annotates code to run on GPUs, in which our symbolic access range analysis is an essential module. In the following subsections we introduce and motivate these two clients of our analysis, which are investigated in detail in Chapters 3 and 4.

1.1 Client I - Runtime Pointer Disambiguation

A fundamental chain in the compilation pipeline is the resolution of dependencies between memory locations. Solving such dependencies enables better instruction scheduling as dependence information gives the compiler more freedom to reorder program statements. The ability to disambiguate memory locations is also essential to enable more specific optimizations, such as code vectorization [Karrenberg and Hack, 2011], automatic parallelization [Lee et al., 2009; Yang et al., 2010], scalar promotion [Suresh et al., 2014] and several loop transformations, such as fusion, fission, reversal, interchanging, skewing and tiling [Wolfe, 1996, Ch.9]. Nevertheless, as important as this problem is, the research community has not yet solved dependence analyses satisfactorily.

To solve memory dependencies effectively, compilers need an accurate alias analysis which tells, for each pair of pointers, if they must, may or may-not dereference overlapping areas. However, solving these queries precisely is undecidable in the presence of procedure calls [Landi and Ryder, 1991], as aliasing may or may not happen depending on the calling context. In the absence of this feature, the problem becomes NP-hard [Horwitz, 1997]. One of the goals of this work is to improve this scenario by equipping compiler writers with a pointer disambiguation technique that lets them address two challenges – the resolution of memory dependencies and the maintenance of this solution across program optimizations – in a precise and efficient way.

We present different ways to distinguish memory regions at runtime. Section 3.3 introduces a hybrid pointer disambiguation technique based on our symbolic access range analysis. Given a program region where memory is accessed, we generate – statically – the conditions that must be met so that memory locations do not overlap. We compare it to a second hybrid method to produce such checks, based on the polyhedral model [Bondhugula et al., 2008]. In Section 3.2 we discuss a totally dynamic pointer disambiguation technique, which augments `libc`'s memory allocator with machinery to associate size information with pointers. We can query this meta-data at runtime, to check if two pointers may dereference areas that overlap.

We clone the regions guarded by dynamic checks, letting the compiler optimize them with the extra knowledge that they are alias-free. At runtime we can query these guards, and the result tells us when it is safe to jump into the optimized version of the code.

The purely dynamic approach of Section 3.2, and the hybrid techniques of Section 3.3 can be used independently, or combined. The former can be used more extensively: it works for any pointer. The hybrid approaches have lower overhead, but they only work for pointers whose bounds can be determined statically.

The dynamic disambiguation of pointers has been used in specific contexts before, for instance, to enable automatic code parallelization [Rus et al., 2003]. Compilers for C/C++ and other languages also commonly use guards and code versioning to enable vectorization in the presence of pointer aliasing, but their use of versioning is commonly limited to what is needed to permit classical inner-loop vectorization. As we aim to enable the effective optimization of large and possibly complex loop nests, we propose new techniques that are powerful enough to handle larger program regions and possibly imperfectly nested, multi-level loop nests.

In Section 3.4 we validate these points. We have implemented our ideas in the LLVM [Lattner and Adve, 2004] compilation infrastructure, and have used this implementation to improve the effectiveness of Polly [Grosser et al., 2012], a loop-

optimizer implemented on top of LLVM. We have tested our implementation on PolyBench [Pouchet, 2014]. As we show in Section 3.4.1, the alias analyses currently available in LLVM, including a sub-cubic implementation of the Dyck-CFL-Reachability algorithm [Zhang et al., 2013], are unable to disambiguate most of the memory access in PolyBench. Our checks, on the contrary, can do this job. Consequently, we produce code that is faster than the code that Polly-LLVM would produce in many cases. The time that we take to generate the tests is negligible: we analyze the 30 benchmarks available in PolyBench in milliseconds.

1.2 Client II - Automatic Annotation for Data Parallelism and Offloading

Heterogeneous architectures formed by clusters of CPUs and GPUs give us, today, a *de facto* standard in terms of high-performing computing, due to their widespread adoption. To illustrate this statement, recently an implementation of conjugate gradients has been able to scale to 3.12 million heterogeneous cores on Tianhe-2, reaching 623 Tflop/s [Liu et al., 2016]. Currently, directive-based annotation systems stand out among the several different techniques used to program these machines, mostly because they require less platform-specific knowledge when compared to alternatives such as CUDA. Examples of such systems include OpenMP [Jaeger et al., 2015], OpenACC [Wienke et al., 2012] and OpenSs [Meenderinck and Juurlink, 2011]. The idea behind this programming model is simple, yet appealing: annotations work as a meta-language, which give developers the ability to grant parallel semantics to syntax originally written to run sequentially. Hence, developers can reap all the benefits from modern parallel hardware, without having to worry too much about minutiae of concurrent programming, such as race conditions and deadlocks – such inconveniences are left to the compiler. Success stories of such annotation systems abound, and, combined with modern accelerators, they have led to substantial performance gains [Bertolli et al., 2014; Reyes et al., 2012; Tabuchi et al., 2016; Wienke et al., 2012].

Nevertheless, annotating code to run in an accelerator device is still a difficult task, which often requires familiarity with the input program that must be transformed. Three challenges, in particular, are daunting: the discovery of parallel loops, the disambiguation of pointers, and the estimation of memory bounds. All these problems have been attacked, in a way or another, by the programming language community. Automatic parallelization has been the focus of much research, to the point that compilers are able, today, to detect parallel loops with high accuracy [Baskaran et al.,

2010]. Pointer disambiguation is another well studied problem [Hind, 2001], with fully static [Andersen, 1994], fully dynamic [Duck and Yap, 2016], and hybrid solutions, with the latter being one of the targets of this work. Finally, the problem of estimating with high accuracy the bounds of memory regions has been tackled before [Nazaré et al., 2014] and is extended here. And yet, in spite of all these advances, we still do not have the necessary equipment that enables developers to annotate code automatically with acceleration directives. In this work, we combine our symbolic memory range analysis, hybrid pointer disambiguation, and traditional compiler analyses to solve this problem. To this effect, we introduce a tool, DawnCC, which annotates code automatically.

The contributions of this new technology are the following: (i) we use our symbolic range analyses to derive access bounds to memory regions, which lets us annotate code with data-copy directives. These directives move data between the different processors that constitute a heterogeneous parallel system. (ii) We use our hybrid pointer disambiguation technique to enable the discovery of more parallel regions in the sequential source code, as it reduces dependencies between data. (iii) We resort to Ferrante *et al's* notion of Program Dependence Graph [Ferrante et al., 1987] to eliminate redundant data-transfer primitives. This optimization, which we call *copy coalescing*, lets us remove transfer operations between parallel regions marked as kernels. Furthermore, as a by-product of the analyses that we perform, it lets us avoid moving read-only data from accelerator back to host. Preserving the correct semantics of the C program, when used in tandem with OpenMP 4.0 or OpenACC annotations, was one of the core challenges that we had to surpass in this work. (iv) Finally, a key challenge that we had to solve was how to bridge the gap between source code and compiler's intermediate representation. All the analyses and optimizations described so far have been conceived to run on programs in the Static Single Assignment (SSA) representation. Yet, our annotations must be inserted onto source code. Far from being a “pretty-printing” problem, recovering high-level information from an optimized low-level language turned out to be a non-trivial problem.

The final product that stems out of these contributions is a tool, henceforth called DawnCC, that frees developers from the tedious and error prone task of inserting copy directives in programs. This tool inserts OpenACC or OpenMP 4.0 annotations into programs, with the goal of being more optimized and as readable as the annotations inserted by a person. Annotated loops run on an accelerator. This transformation does not require any intervention from users – it is completely automatic. In order to validate these ideas, we have used DawnCC to transform programs present in PolyBenchGPU¹.

¹<http://cavazos-lab.github.io/PolyBench-ACC/>

In this suite, we are able to annotate and parallelize up to 95% of all the loops. The net result is performance: by annotating loops automatically, we have been able to observe speedups of up to 105x on a CPU-GPU based architecture.

Chapter 2

Symbolic Inference of Memory Access Ranges

In this chapter we present an analysis capable of inferring symbolic integer bounds for memory access operations. This method, however, is not new; we build on top of the symbolic region analysis for arrays originally proposed by Rugina and Rinard [2000] and thoroughly extended by Nazaré et al. [2014]. Our variant of this technique operates on a simpler core language, with simpler and easier to compute range operations, while still being precise enough to handle the majority of constructs found in real-world programs. In the following sections we define the core representation upon which our analysis operates, show how to derive symbolic ranges out of such a representation and, finally, present a code generation strategy to convert the inferred symbolic expressions into actual computation.

2.1 The Core Language

The technique we present in this chapter requires us to compute symbolic bounds on expressions. Figure 2.1 defines the syntax of the programs that we handle: a small subset of the expressions typically found in programming languages, enough to describe most of the memory accesses and conditions that control the loops found in actual programs, as we show experimentally. This core language is, however, not a programming language. Its sole purpose is to identify which constructs are relevant to our analyses. We do not convert input programs to it, but rather filter out any code that does not belong to the subset that it represents. This language has only assignments, dereferences, sequences of commands, and loops, about which we assume a few facts that simplify the analyses that we will describe further.

$$\begin{array}{l}
C ::= \text{while } v < v : C \\
\quad | C; C \\
\quad | v = E \\
\quad | \text{deref}(v, v, n) \\
E ::= n \\
\quad | v \\
\quad | v + v \\
\quad | n \times v \\
\quad | v /_u n \\
\quad | \text{trunc}(n, v) \\
\quad | \text{sign_ext}(n, v) \\
\quad | \text{zero_ext}(n, v) \\
\quad | \phi(v, v)
\end{array}$$

Figure 2.1: The syntax of our core language. We let $n \in \mathbb{N}$.

Firstly, every loop in our language is controlled by a comparison such as $v_i < v_n$, in which v_n is a loop invariant value. The generalization of the techniques that we discuss in this Section to the other three relational disjunctions, e.g., \leq , $>$, \geq is trivial. Such loops are called *interval loops* by Alves et al. [2014] and although simple, they are very common. According to Alves *et al.*, this format is found in 67% of all loops in SPEC CPU 2006, a widely used benchmark suite with millions of lines of C and C++ code Henning [2006]. Secondly, we assume that v_i is a *canonical induction variable*. An induction variable is canonical if it is always incremented by one at each loop iteration. There are standard techniques to canonicalize *affine* induction variables used in interval loops [Appel and Palsberg, 2002]. An affine induction variable’s value is given by $b + i \times s$, where b is its initial value, s is its increment, and i counts the number of iterations of the loop. Therefore, even assuming canonical induction variables, we are still able to handle all the 67% of interval loops found in SPEC CPU 2006.

Example 1 *Figure 2.2 shows an example program adapted from Chabbi and Mellor-Crummey [2012] with constructs that do not belong to our core language filtered out. We have eliminated the control flow and the negation inside the loop, as these constructs play no role in our analyses.*

Because our core language is so uninvolved, we shall not provide a formal semantics to it; instead, we shall describe a few of its constructs informally. The command “ $\text{deref}(v_p, v_i, s_i)$ ” denotes an array dereference: we are accessing the memory location of size s_i at address $v_p + v_i$, where v_p is a base pointer, and v_i is an indexing expression, as defined in Figure 2.1. This access can denote either a read or a write operation. We distinguish sign and unsigned (*u*) operations. We are able to determine bounds

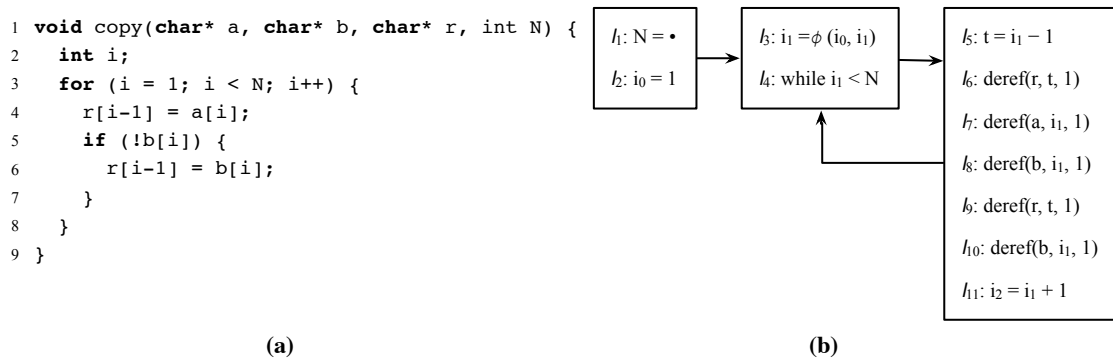


Figure 2.2: (a) example program adapted from Chabbi and Mellor-Crummey [2012] and (b) a version of it re-written in our core language. The symbol \bullet denotes an unknown input. We have eliminated the control flow in Figure (b), as it has no influence in our analysis.

of expressions involving signed and unsigned addition and multiplication, but we only deal with unsigned division. We shall use the subscript u next to the division bar, to indicate that it is unsigned. We only allow multiplications and divisions of expressions by a constant. The operation “`trunc(n, v)`” converts the value carried by v into an n -bits integer by removing v ’s most significant bits. Nothing happens if v already uses less than n bits. The operation “`sign_ext(n, v)`” does the opposite: it extends the binary representation of v ’s value to an n -bits integer. This extension preserves v ’s arithmetic sign. Similarly, “`zero_ext(n, v)`” does type extension, but fills the extra most-significant bits with zeros. For reasons that we clarify in Section 2.2, we work on programs in the Static Single Assignment (SSA) format [Cytron et al., 1991]; thus, our core language defines ϕ -functions. Induction variables are defined by ϕ -functions.

2.2 Computing Symbolic Variable Bounds

If the same base pointer v_p is dereferenced n times within a loop nest, then let $\text{deref}(v_p, v_i, s_i), 1 \leq i \leq n$ be each one of these accesses. We generate code to compute, for each v_i , its lower and upper limits. We store the lower limit into a fresh variable v_{il} and the upper limit in another fresh variable v_{iu} . We estimate the largest region M covered by the array via Equation 2.1.

$$\begin{aligned}
 M &= \max(v_{1u} + s_1, \dots, v_{nu} + s_n) \\
 &\quad - \min(v_{1l}, \dots, v_{nl}) + 1
 \end{aligned}
 \tag{2.1}$$

$$\begin{array}{c}
\frac{\mathbf{B}(v_1) \rightarrow (_, v_{l1}, v_{u1}) \quad \mathbf{B}(v_2) \rightarrow (_, v_{l2}, v_{u2})}{\mathbf{G}(v = v_1 + v_2) \triangleright \mathbf{B}(v) \leftarrow ("v_l = v_{l1} + v_{l2}; v_u = v_{u1} + v_{u2}", v_l, v_u)} \\
\\
\frac{\mathbf{B}(v_1) \rightarrow (_, v_{l1}, v_{u1})}{\mathbf{G}(v = n \times v_1) \triangleright \mathbf{B}(v) \leftarrow \begin{array}{l} \text{if } n < 0 \quad \text{then } ("v_l = n \times v_{l1}; v_u = n \times v_{u1}", v_l, v_u) \\ \text{else } ("v_l = n \times v_{l1}; v_u = n \times v_{u1}", v_u, v_l) \end{array}} \\
\\
\frac{\mathbf{B}(v_1) \rightarrow (_, v_{l1}, v_{u1})}{\mathbf{G}(v = v_1 \ /_u n) \triangleright \mathbf{B}(v) \leftarrow \begin{array}{l} \text{if } n \neq 0 \quad \text{then } ("v_l = v_{l1} \ /_u n; v_u = v_{u1} \ /_u n", v_l, v_u) \\ \text{else } ("v_l = 0; v_u = \text{MaxTp}_u(v)", v_l, v_u) \end{array}} \\
\\
\frac{\mathbf{B}(v_1) \rightarrow (_, v_{l1}, v_{u1})}{\mathbf{G}(v = \text{trunc}(n, v_1)) \triangleright \mathbf{B}(v) \leftarrow \begin{array}{l} \text{if } v_{l1} \geq -2^n \text{ and } v_{u1} \leq 2^n - 1 \\ \text{then } ("v_l = v_{l1}; v_u = v_{u1}", v_l, v_u) \\ \text{else } ("v_l = \text{MinTp}_s(v); v_u = \text{MaxTp}_s(v)", v_l, v_u) \end{array}} \\
\\
\frac{\mathbf{B}(v_1) \rightarrow (_, v_{l1}, v_{u1})}{\mathbf{G}(v = \text{sign_ext}(n, v_1)) \triangleright \mathbf{B}(v) \leftarrow \begin{array}{l} \text{if } v_{l1} \geq -2^n \text{ and } v_{u1} \leq 2^n - 1 \\ \text{then } ("v_l = -2^n; v_u = 2^n - 1", v_l, v_u) \quad \text{else } ("v_l = v_{l1}; v_u = v_{u1}", v_l, v_u) \end{array}} \\
\\
\frac{\mathbf{B}(v_1) \rightarrow (_, v_{l1}, v_{u1})}{\mathbf{G}(v = \text{zero_ext}(n, v_1)) \triangleright \mathbf{B}(v) \leftarrow \begin{array}{l} \text{if } v_{l1} \geq -2^n \text{ and } v_{u1} \leq 2^n - 1 \\ \text{then } ("v_l = 0; v_u = 2^n - 1", v_l, v_u) \quad \text{else } ("v_l = v_{l1}; v_u = v_{u1}", v_l, v_u) \end{array}}
\end{array}$$

Figure 2.3: Abstract interpretation of arithmetic expressions in our core language. We let $\text{MaxTp}_s(v)$, $\text{MinTp}_s(v)$ be the maximum and minimum values of variable v 's type.

The computation of M , in Equation 2.1, reads a few variables v_{ij} . We can estimate the bounds of these variables using expressions produced by the code generator in Figure 2.3. The rules in Figure 2.3 create a mapping \mathbf{B} . \mathbf{B} maps every variable v_i used as a dereference index into a tuple (I, v_{il}, v_{iu}) , such that I is a set of assembly instructions, v_{il} is the name of the variable that will hold v_i 's lower bound and v_{iu} is the name of the variable that holds v_i 's upper bound. The assembly instructions I will be used later to produce the runtime checks that we need to disambiguate pointers, as well as data transfer directives. Because our mapping \mathbf{B} is a function on variable names, we require every variable name to be unique in the program. The SSA representation [Cytron et al., 1991], so common in modern compilers, gives us this property.

2.2.1 Bootstrapping the Code Generator

When generating code, we must ensure that each instruction visited has all its operands already mapped by \mathbf{B} . To meet this requirement we must (i) define mappings for loop invariant variables plus canonical induction variables, and (ii) visit instructions within the loop in a pre-order of the program’s dominance tree. A loop invariant variable is used within the loop, but defined outside it. To ensure (i), we “bootstrap” the code generator with range information. We assume the existence of an environment \mathbf{R} which maps variables to their symbolic range intervals. Constructing \mathbf{R} for programs in our core language is a standard procedure in the compiler literature. A more formal treatment on this subject is given by Nazaré et al. [2014]. If $\mathbf{R}(v_i) = [l, u]$, then before starting the traversal of the loop instructions, we perform the binding below for each variable v_i that we need to bootstrap:

$$\mathbf{B}(v_i) \leftarrow (“v_{il} = l; v_{iu} = u”, v_{il}, v_{iu})$$

Example 2 *The loop in Figure 2.2 (b) contains one invariant variable: N , and one canonical induction variable: i_1 . A symbolic range analysis determines that $\mathbf{R}(N) = [N, N]$, and that $\mathbf{B}(i_1) = [0, N]$. Thus, we start the mapping \mathbf{B} with the following bindings: $\mathbf{B}(N) = (“v_{Nl} = N; v_{Nu} = N” , v_{Nl}, v_{Nu})$ and $\mathbf{B}(i_1) = (“v_{i_1l} = 0; v_{i_1u} = N” , v_{i_1l}, v_{i_1u})$.*

As mentioned before, we visit the instructions in the loop body in a pre-order traversal of that body’s dominance tree. During this traversal, \mathbf{G} updates the mapping \mathbf{B} . We let $\mathbf{B}(v) \leftarrow t$ denote a new function \mathbf{B}' , such that $\mathbf{B}' = \lambda x. \text{if } x = v \text{ then } t \text{ else } \mathbf{B}(x)$.

Example 3 *When analyzing the body of the loop seen in Figure 2.2 (b), we visit instructions in the order $\ell_5, \ell_6, \ell_7, \ell_8, \ell_9, \ell_{10}, \ell_{11}$. Only variables i_1 and t are used to index arrays. $\mathbf{B}(i_1)$ has been initialized in the bootstrapping phase. Upon visiting $\ell_5 : t = i_1 - 1$, we find that $\mathbf{B}(t) \rightarrow (“v_{tl} = v_{i_1l} - 1; v_{tu} = v_{i_1u} - 1” , v_{tl}, v_{tu})$.*

Chapter 3

Runtime Pointer Disambiguation

In this chapter, we show how our symbolic access range analysis can be used to build a lightweight hybrid memory disambiguation pass. We compare our solution to two other disambiguation methods: one hybrid as our own and another one fully dynamic, implemented by two distinct research groups.

3.1 Overview

We shall use the functions in Figure 3.1 to motivate the ideas that we discuss in this chapter¹. Function `array_sum_1` is a simple routine that sums up all the elements in an array of integers. Figure 3.1 contains another version of this routine: `array_sum_2`. This second implementation corresponds to the code that would be produced by a traditional compiler technique: *loop invariant code motion*, which consists in moving invariant computations outside loops. In our example, this computation is the load of `*s`, which happens in line 3 of `array_sum_1`, and the store into `acc`, in line 4.

At first glance, these optimizations may seem easy venture for mainstream compilers. However, that is not the case. We have fed `array_sum_1` to three different compilers: Clang 3.4, gcc 4.8.2 and icc 15.0, and none of them, at their highest optimization level, were able to produce `array_sum_2`. The culprit for this failure is aliasing. Not knowing if `acc` and `s` alias or not, the compiler must assume such a possibility. In face of aliasing, the store at line 4 of `array_sum_1` could change the result of the load of `s` at line 3. The compiler cannot hoist the store into `acc` outside the loop either. If `acc` overlaps any address within array `src`, then functions `array_sum_1` and `array_sum_2` are not semantically equivalent. As a testimony of these limitations, it

¹We use source code to illustrate our techniques, but all our optimizations happen at the compiler's intermediate representation level.

```

1 void array_sum_1(int *src, int *s, int *acc) {
2     int i;
3     for (i = 0; i < *s; i++) {
4         *acc += src[i];
5     }
6 }

1 void array_sum_2(int *src, int *s, int* acc) {
2     int i;
3     int N0 = *s;
4     int N1 = *acc;
5     for (i = 0; i < N0; i++) {
6         N1 += src[i];
7     }
8     *acc = N1;
9 }

```

Figure 3.1: An example (`array_sum_1`) in which potential aliasing hinders compiler optimizations such as loop invariant code motion as done in `array_sum_2`.

suffices to add the `restrict` keyword to the declaration of the arguments of function `array_sum_1`, and all compilers will be able to vectorize that code.

In Figure 3.1, the possibility of aliasing is unfortunate, for it hinders the compiler from applying very effective optimizations: function `array_sum_2` is substantially faster than its original version. On an Intel Core i5 at 2.9GHz, using Clang 3.4 -O3, `array_sum_2` can be over $2.5\times$ faster than `array_sum_1`. Such speedup is possible due to the extensive vectorization support that exists in the Intel hardware. Static pointer analyses [Andersen, 1994; Pereira and Berlin, 2009; Zhang et al., 2013] cannot enable this kind of optimization, because pointer overlapping may indeed happen in Figure 3.1. In other words, nothing prevents function `array_sum_1` from receiving the same pointer as its actual parameters.

A purely dynamic approach to pointer disambiguation. The disambiguation of pointers such as those passed to `array_sum_1` as parameters requires runtime knowledge. A possible way to make such knowledge available consists in modifying the memory allocation system used by C. In this manner, we can tag different memory regions with different identifiers to disambiguate pointers that refer to distinct allocations. Binding meta-information to pointers is not a new technique. It has been used to secure programs written in C against out-of-bounds memory errors [Akritidis et al., 2009; Nagarakatte et al., 2009; Necula et al., 2002; Serebryany et al., 2012]. In this work, we use these techniques to disambiguate pointers in order to enable optimizations. Figure 3.2 illustrates such a use. In this example, we assume that $T(p)$ returns

```

1 void array_sum_3(int* src, int* s, int* acc) {
2   void *heapId_src = T(src);
3   void *heapId_s   = T(s);
4   void *heapId_acc = T(acc);
5
6   if ((heapId_src != heapId_s)
7   && (heapId_src != heapId_acc)
8   && (heapId_s != heapId_acc)) {
9     // Code region where aliasing will never happen
10    int i;
11    int N0 = *s;
12    int N1 = *acc;
13    for (i = 0; i < N0; i++) { N1 += src[i]; }
14    *acc = N1;
15  } else {
16    // Code region where aliasing may happen
17    int i;
18    for (i = 0; i < *s; i++) {*acc += src[i];}
19  }
20 }

```

Figure 3.2: Modified version of `array_sum_1` in which the compiler generates checks to disambiguate pointers.

a unique identifier of the allocated memory block pointed by `p`. By combining these runtime checks with code versioning, we can certify to the compiler that some program regions are alias-free. Some compilers provide support for this type of interaction. The LLVM intermediate representation, for instance, contains a `noalias` type modifier and corresponding metadata to model the absence of possible pointer aliasing.

The technique that we have discussed in Figure 3.2 is totally dynamic: it relies on a modified memory allocator to disambiguate pointers at runtime. Its main advantage is applicability: it gives us the opportunity to distinguish any pair of pointers that refer to different allocations. On the other hand, it may impose a runtime overhead on the program: metadata must be kept for each pointer, and depending on the implementation, $T(p)$ may not be $O(1)$. This technique, for which we provide more details in Section 3.2, was mainly designed and implemented by Fabian Gruber, from INRIA-France [Alves et al., 2015]. For now, we present an alternative to this approach, which combines checks produced statically with runtime knowledge.

A hybrid approach to pointer disambiguation. Figure 3.3 shows a different way to carry out pointer disambiguation. In this example, we try to infer - statically - conservative bounds on the memory referenced by the pointers that exist within a

```

1 void array_sum_4(int *src, int *s, int *acc) {
2   int len_src = (*s) * 4;
3   if ((src + len_src <= s || src >= s + 4)
4       && (src + len_src <= acc || src >= acc + 4)
5       && (s + 4 <= acc || s >= acc + 4)) {
6     // Code region where aliasing will never happen
7     ...
8   } else {
9     // Code region where aliasing may happen
10    ...
11  }
12 }

```

Figure 3.3: Modified version of `array_sum_1` in which the program can inspect the size of the region referenced by a pointer. Here `sizeof(int)` is considered to be 4.

given code region. These bounds are written as functions of variables used in that area. The value of these variables may not be known statically. However, during the execution of the program, once its flow reaches the entry of the region analyzed, we can inspect these values, and use them to obtain a concrete estimate of the size of allocated memory. When analyzing function `array_sum_1` in Figure 3.1 we know that pointers `acc` and `s` are constants – even though their contents may not be, due to aliasing. Moreover, we know that the induction variable `i`, used to index `src`, ranges from 0 to `*s-1`. In the absence of aliasing, this upper limit, e.g., `*s`, is invariant. These observations give us the subsidies to generate the checks ² seen in lines 2-5 of Figure 3.3. In Section 3.3 we explain in deeper detail two strategies to produce these checks without any intervention from the user. The first one, in Section 3.3.1, builds on top of the Symbolic Range Analysis that we presented, both implemented by us on top of LLVM. The second one, based on the Polyhedral Model, was proposed and implemented by Johannes Doerfert and Tobias Grosser [Alves et al., 2015], the main maintainers of Polly.

3.2 Purely Dynamic Pointer Disambiguation

We have implemented a memory allocator that lets us disambiguate pointers at runtime [Alves et al., 2015]. To achieve this end, our allocator provides an efficient data structure for querying the starting address of the heap allocation for arbitrary pointers. Disambiguation is enabled by the fact that this starting address, the base pointer,

²The checks are presented in C code for illustrative purposes. The actual expressions are generated in LLVM’s Intermediate Representation and inserted in the final assembly


```

1 void copy(char* a, char* b, char* r, int N) {
2     int i;
3     for (i = 1; i < N; i++) {
4         r[i-1] = a[i];
5         if (!b[i]) {
6             r[i-1] = b[i];
7         }
8     }
9 }

```

Figure 3.4: Example adapted from Chabbi and Mellor-Crummey [2012]. This program contains code that is difficult to optimize due to aliasing.

uniquely identifies each heap allocation. The identifier for pointers that do not point to a valid heap allocation is the null pointer, since that is never a valid heap address. Our memory allocator works for programs written in C/C++; nevertheless, similar techniques can be used in other programming languages that support pointer arithmetic, including assembly languages. We have replaced libc’s `malloc` routine with an implementation of our own. Our surrogate overwrites the implementations of `malloc`, `calloc`, `realloc`, `aligned_alloc`, `posix_memalign` and `free` with versions that update the data structure used to query the heap allocation for a given pointer. This is implemented as a *red-black tree* [Bayer, 1972] T that will provide us with logarithmic-time access to this meta information. The additional data required for the tree is embedded in a header directly before the user visible allocation. A tree lookup gives us the address of the header and thus the unique identifier associated with the block of memory that p can dereference. The actual memory allocations are performed via the wrapped allocator. The sizes of all allocations are adjusted so they can fit the required header for the search tree.

Whenever memory is allocated through libc’s `malloc`, we add an entry to T . An assignment such as $p = \text{malloc}(N)$ will assign p to the range $[p, p + N[$ in our red-black tree. Notice that these intervals are bounded by arbitrary integers determined at runtime, not symbols. That is possible because the values of p and N are known at runtime. Now, assuming the program is correct, a pointer p_1 derived from another pointer p_0 (e.g. $p_0 = p_1 + c$, $c \in \mathbb{N}$) will cause $T(p_0) = T(p_1)$. Contrary to previous work on program correctness [Akritidis et al., 2009; Nagarakatte et al., 2009; Serebryany et al., 2012], our goal is to enable optimizations, not to find bugs in programs.

In this section, and also in Section 3.3, we shall be using the program in Figure 3.4 to illustrate our ideas. The program in the figure has been adapted from an example used by Chabbi and Crummey [Chabbi and Mellor-Crummey, 2012] to illustrate per-

```

void copy(char* a, char* b, char* r, int N) {
    int i;
    // Outermost SESE region
    for (i = 1; i < N; i++) {
        r[i-1] = a[i];
        // Innermost SESE region
        if (!b[i]) {
            r[i-1] = b[i];
        }
    }
}

```

Figure 3.5: Program of Figure 3.4 with Single-Entry, Single-Exit regions highlighted.

formance bugs. Function `copy` moves the contents of array `a` or `b` to buffer `r`. The program contains two statements that store into `r`: the first at line 4, the second at line 6. If every cell of array `b` is zero, then both store instructions will be always performed during the program’s execution. The double occurrence of the store instruction cannot be optimized away, because if arrays `b` and `r` overlap, then the assignment at line 4 may change the outcome of the test that happens in line 5. In other words, due to aliasing, we cannot convert that code into `tmp = b[i]; r[i-1] = tmp != 0 ? tmp : a[i]`. Instead, we need to store `r[i-1]` before loading `b[i]`, as the value of `b[i]` may be changed by this store.

We insert runtime checks at the beginning of the subparts of a program called *Single-Entry-Single-Exit* (SESE) regions [Ferrante et al., 1987]. SESE regions are sets of basic blocks with a single incoming edge and a single outgoing edge to the rest of the program’s control flow graph (CFG). We only insert tests at the beginning of SESE areas because, in this way, we are certain that any program flow entering that region will execute our tests. Figure 3.5 highlights two of the regions that exist in our original example. Whenever we have a nested set of SESE blocks, we try to insert checks at the outermost sector in which all the pointers to be tested are *alive*. We say that a variable `v` is alive at a program point `p` if there exists a path from `p` to a use of `v` that does not go across any redefinition of `v`. In Figure 3.4 we have three pointers to disambiguate. All these pointers are alive at the entry of the outermost SESE sector within the function. Our checks will be inserted at that point. Our approach is actually not limited to SESE regions but can also support regions with multiple entries and exits, though this complicates code versioning. The limitation to SESE regions in our implementation stems from the fact that optimizations performed by Polly, the optimizer used in our experiments, are restricted to such regions.

```

1 void copy(char* a, char* b, char* r, int N) {
2   int i;
3   int heapId_a = T(a);
4   int heapId_b = T(b);
5   int heapId_c = T(c);
6   if ((heapId_a != heapId_b)
7     && (heapId_a != heapId_c)
8     && (heapId_b != heapId_c)) {
9     // Code region where aliasing will never happen
10    ...
11  } else {
12    // Code region where aliasing may happen
13    ...
14  }
15 }

```

Figure 3.6: Program after instrumentation.

```

1 char* copy_message(char* src, char* dst) {
2   for (int i = 0; i < 4; i++) {
3     dst[i] = src[i];
4   }
5 }
6
7 struct message *msg =
8     malloc(sizeof(struct message));
9 copy_message(&msg.s, &msg.d);

```

Figure 3.7: Program that illustrates over-approximation of our purely dynamic approach.

Figure 3.6 shows our example program, after it has been instrumented with runtime checks. The original program contains uses for three different pointers at the SESE region that we are instrumenting. Thus, we had to insert checks to disambiguate three pairs of memory regions. For the sake of simplicity, we insert $O(N^2)$ checks at the beginning of a region that contains N pointers. We avoid inserting some checks whenever static pointer analysis is able to disambiguate pairs of pointers at compilation time. As we show in Section 3.4, in practice our technique creates a small number of guards per region.

At runtime we can query T for the unique identifier associated with p . The cost of such a query is logarithmic in the number of allocations that are live in the program at the time of the query. We know that two pointers, p_0 and p_1 *cannot* dereference overlapping memory regions if $T(p_0) \neq T(p_1)$. Notice that in the opposite case, e.g., $T(p_0) = T(p_1)$, it is still possible that p_0 and p_1 do not overlap, as the

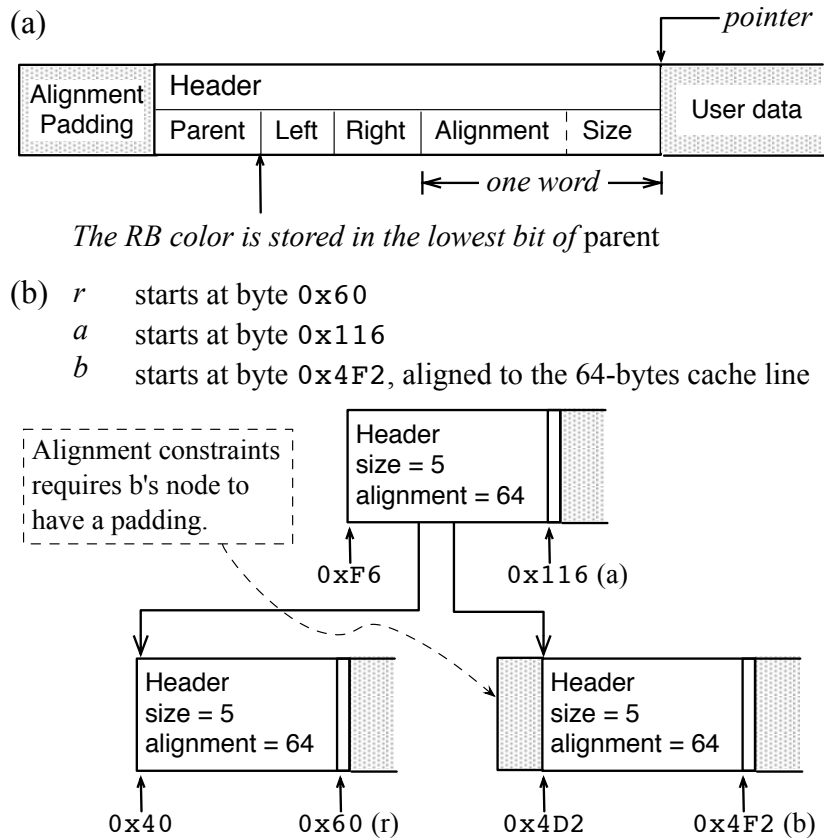


Figure 3.8: (a) Structure of the red-black nodes that we use to track memory size. (b) Dynamic view of the red-black tree when the program flow reaches line 5 of Figure 3.6.

function `copy_message`, in Figure 3.7, illustrates. In this figure, the store and load at line 3 cannot dereference the same memory address. Nevertheless, we have that the base pointers of these stores will be associated with the same block of memory, which is passed to `copy_message` in line 9. We adopt this conservative approach for two reasons. First, it makes our implementation simpler. Second, when situations such as that seen in Figure 3.7 happen, it may still be possible that we use static approaches to avoid inserting dynamic guards. As an example, LLVM’s basic alias analysis is able to disambiguate the two pointers in line 9 of Figure 3.7.

3.2.1 Implementation Details

Figure 3.8 shows a few implementation details of our memory allocator. The structure of each node can be seen in Figure 3.8 (a). The part (b) of the figure shows a snapshot of the tree. Each call to memory allocation routines such as `malloc`, `calloc`, `realloc` and `aligned_alloc` is intercepted and the requested size is augmented to make room

for meta-data. The extra memory region where we store this meta information shall be called the *node's header*. The actual pointer returned to the user references the next address after this header. Besides the pointers used by the tree itself, e.g., left and right, a node's header also stores the size of the allocation and its alignment, as we show in Figure 3.8 (a). Hence, on a 64 bit system, by default, our implementation uses 32 bytes of data per allocation. Storing alignment information is required for supporting `aligned_alloc` and related libc APIs. If the user requests memory with a large alignment factor, then we also have to add some padding before the header, to ensure that this header does not break the requested alignment. Large alignment constants are usually the size of the cache line, or of the virtual memory page.

Our implementation gives us room for optimizations. Firstly, since the alignment is always a power of two, we save space by only storing its binary logarithm, which can be efficiently computed in different architectures, such as in x86 using the count-leading-zeros (`clz`) instruction. Moreover, we have the guarantee that the header is always aligned to 8 bytes; hence, we can safely encode the color of the red-black node in the least significant bit of the parent pointer. The keys in the red-black tree are simply the starting addresses of the nodes themselves. Therefore, they are directly available while traversing the tree and do not need to be stored separately. A red-black tree is an ordered search tree; thus, we can efficiently search for the allocation related to a pointer p by finding the node with the largest address smaller than p .

In general we assume that the source program is correct and does not use invalid pointers, so for a pointer that points past the end of an allocation we still return the base pointer of the closest allocation. Nevertheless, for debug purposes, we still store the size of each allocation in the nodes header, which allows us to detect such erroneous pointers. For non-heap allocated memory regions, such as stack allocations, our query function will not return an identifier and the disambiguation check will not pass. Note that there are more specialized data structures for interval queries, such as segment trees [Bentley and Friedman, 1979]. Segment trees support efficient queries even with overlapping intervals, but since the intervals defined by the beginning and end of allocations never overlap, we chose to use a simplified interval tree implemented on top of a red-black tree [Cormen, 2009].

3.3 Hybrid Pointer Disambiguation

The purely dynamic approach that we have discussed in Section 3.2 is very precise, being able to disambiguate pointers in many different scenarios, as we show experi-

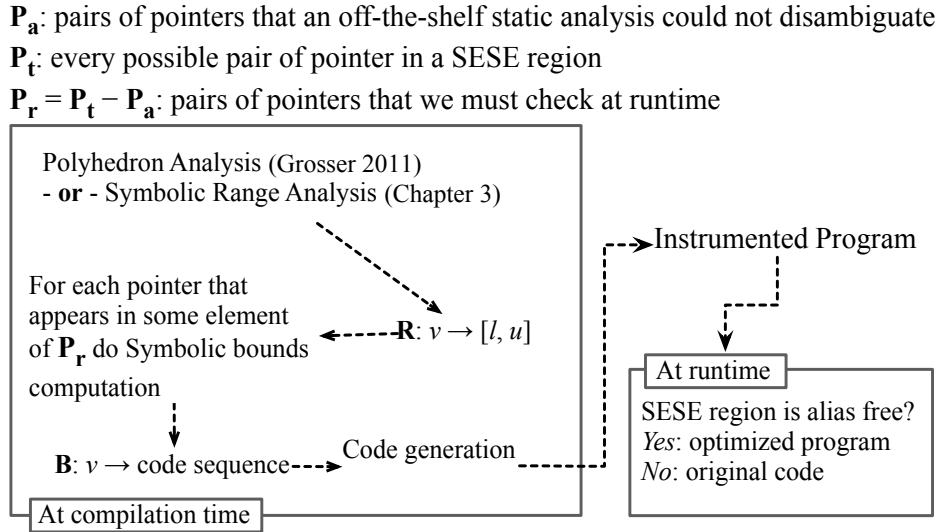


Figure 3.9: Overview of the code generation methodology used in Section 3.3.

mentally; however, it has one shortcoming: queries such as T(a), T(b) and T(c) in lines 3-5 of Figure 3.6 may be expensive. In our implementation, each such query is logarithmic on the number of currently alive memory allocations. To avoid this cost, we have designed a hybrid pointer disambiguation technique, which does not require us to change the memory allocation library, and whose queries are $O(1)$. In this section we describe such approach.

Figure 3.9 presents a general overview of the approach that we advocate here. For each SESE region in a program we run the steps seen in Figure 3.9. We combine code versioning with runtime checks to give the compiler the opportunity to optimize certain program regions assuming the absence of aliasing. To reduce the overhead of the execution of the runtime checks, we may *optionally* prune some of these tests away with a profiler. In other words, if a pair of pointers is found to alias with some probability, we do not try to disambiguate those pointers dynamically.

In this section we explain how we generate the checks that we use to disambiguate pointers at runtime. To this end, we explain the actual computation of the disambiguation checks in Section 3.3.1. We answer more practical questions, such as where to insert the checks, in Section 3.3.3. We use the example first seen in Figure 3.4 to illustrate the notions introduced along these developments.

3.3.1 Building Alias Checks From Symbolic Bounds

If M_1 and M_2 are estimations for the regions dereferenced by two different pointers p_1 and p_2 , as defined in Section 2.2, then we say that they will not overlap if:

$$p_1 + M_1 \leq p_2 \text{ or } p_2 + M_2 \leq p_1 \quad (3.1)$$

To compute the region size M_p , we implemented and evaluated two approaches: (1) bound computation based on the polyhedral model, and (2) bound computation bootstrapped by symbolic range analysis presented in Chapter 2. The latter can be used to disambiguate pointers for arbitrary regions without the need of a polyhedral analysis while the former requires the polyhedral description of the SESE region. One advantage of (2) is the fact that it can handle non-affine patterns of access while (1) cannot; one advantage of (1) (when relations are affine) is the extra precision: it relies on a relational analysis; hence, it is able to derive relations between variables, even if these variables are not linked syntactically in the program text. The program below illustrates this point:

```
void foo(int *u, int *v, int N, int S) {
    int i;
    int j = S;
    for (i = 0; i < N; i++) { u[j] = v[i]; j++; }
}
```

The polyhedral-based approach (Section 3.3.2) relies on the relations between integer points bounded by linear constraints. It can infer that index j is such that $S \leq j < S + N$. On the other hand, the method based on range analysis tries to infer symbolic bounds to variables. This method assigns $[0, N - 1]$ to i , and $[0, +\infty]$ to j , because j is not bounded by the loop. Therefore, whereas the polyhedron-based approach of Section 3.3.2 can deal with the program above, the technique based on symbolic range analysis would not disambiguate pointers in this case ³.

3.3.2 Polyhedral Access Range Analysis

For the polyhedral access range analysis, in the context of the core language presented in Section 2.1, we require all v_i to be (piecewise) multidimensional affine functions f_{v_i} in

³Our implementation of the symbolic range analysis presented in Chapter 2 still works precisely for this example, because it will use LLVM’s scalar-evolution analysis to infer bounds to variable j . Scalar evolution is described in [Grosser et al., 2012, p.18].

values invariant in the region (parameters) and induction variables of loops surrounding $\text{deref}(v_p, v_i, s_i)$. This excludes non-affine accesses, however the computation of access ranges is a compositional task and a general range analysis can be applied to non-affine expressions.

To get a symbolic range for all dynamic values of the access index v_i we first apply the access function f_{v_i} to the iteration domain of the access. Hence, we symbolically compute the access index for each dynamic instance of $\text{deref}(v_p, v_i, s_i)$ in the multidimensional access space. The linearized minimal and maximal access indices of any access to a base pointer v_p are the bounds of the symbolic access range M_p . To represent the access functions and the iteration domain we utilize the integer set library `isl` [Verdoolaege, 2010]. Using `isl` we also compute the lexicographical minimum and maximum of the multidimensional access space, hence the respective minimal and maximal access, to each base pointer. Note that for non-perfect loops, these bounds can be produced around regions where the accesses are defined by affine functions. The bounds we obtain are again modeled as piecewise affine expressions. To translate them into actual program code, we pass these bounds to a polyhedral AST generator [Grosser et al., 2015] which derives program code that can compute the array bounds at runtime. During program execution, the parameter values in the bounds are known and constant for one execution of the region. As a result, we obtain for each execution of a region precise minimal and maximal access bounds. The parametric bounds we obtain also let us identify accesses that can never be executed under the same parameter evaluation. Hence, if only one of two possible aliasing base pointers will be accessed for fixed but unknown parameter values, then there is no need to generate runtime alias checks.

3.3.3 Code Generation

Algorithm 3.10 inserts runtime checks in a program to disambiguate pointers. The “goto” in line 5 executes whenever we can prove that two pointers, e.g., p_1 and p_2 do not overlap. If we cannot offer such a proof, then we use the jump in line 6. These checks are created for every pair of pointers used within a SESE region. Notice that they are generated statically, but their execution happens dynamically. Thus, by filling up the values in the tests with runtime values we can solve very complex “less-than” checks, even those involving max and min expressions.

Finding a place to insert checks. The loops in any program written in our core language (Figure 2.1) have the SESE property; this implies that these loops have a

Algorithm 3.10: Let p_1 and p_2 be two pointers dereferenced within the same Single-Entry-Single-Exit region, such that:

- p_1 is dereferenced by a set of n instructions $\text{deref}(p_1, v_1^1, s_1^1), \dots, \text{deref}(p_1, v_n^1, s_n^1)$
- p_2 is dereferenced by a set of m instructions $\text{deref}(p_2, v_1^2, s_1^2), \dots, \text{deref}(p_2, v_m^2, s_m^2)$
- $\mathbf{B}(v_j^1) = (I_{1j}, v_{jl}^1, v_{ju}^1), 1 \leq j \leq n$
- $\mathbf{B}(v_j^2) = (I_{2j}, v_{jl}^2, v_{ju}^2), 1 \leq j \leq m$

The following code sequence disambiguates p_1 and p_2 . In this code, variables v_{M1} and v_{M2} have fresh names:

1. $I_{11}; \dots; I_{1n}; I_{21}; \dots; I_{2m};$
2. $v_{M1} = \max(v_{1u}^1 + s_1^1, \dots, v_{nu}^1 + s_n^1) - \min(v_{il}^1, \dots, v_{nl}^1) - 1;$
3. $v_{M2} = \max(v_{1u}^2 + s_1^2, \dots, v_{mu}^2 + s_m^2) - \min(v_{il}^2, \dots, v_{ml}^2) - 1;$
4. if $p_1 + v_{M1} \leq p_2$ or $p_2 + v_{M2} \leq p_1$
5. then goto “ p_1 and p_2 do not overlap”
6. else goto “ p_1 and p_2 overlap”

Figure 3.10: Generation of dynamic pointer checks.

header block H , and an exit block E , such that any node outside the loop can reach any node inside only through H . Similarly, nodes within the loop can reach nodes outside it only through E . H dominates and E post-dominates all the nodes in the loop. In practice, not every loop in an actual assembly program has the SESE property, yet the proportion is high: we have found that all but one of all the loops in PolyBench are SESE. Thus, for practical reasons, we restrict our transformations to SESE loops.

We insert pointer checks in the *pre-header* of the outermost loop in a nest of loops. The pre-header of a loop is the single predecessor of its header. If a pre-header does not exist, i.e., the header has multiple predecessors, then we create one. We only create a check to disambiguate a pair of pointers p_1 and p_2 if all the symbolic expressions that the algorithm in Figure 3.10 needs for this check are available at the loop header. *Available expressions*, a classic compiler analysis [Aho et al., 2006, Ch.9], gives us this information.

Example 4 *Continuing with our example, the disambiguation checks that we want to insert in Figure 2.2 (b)’s program require only the value of variable N . The pre-header of the loop that we analyze contains labels ℓ_1 and ℓ_2 . We insert the check after ℓ_2 . The code sequence that constitutes our check is given by:*

$v_{N1} = N; v_{Nu} = N; v_{il} = 1; v_{1u} = N; v_{il} = v_{il} - 1; v_{tu} = v_{iu} - 1; v_{M1} = v_{tu} - v_{il}; v_{M2} = v_{tu} - v_{il};$ if $r + v_{M1} > b$ or $b + v_{M2} > r$ goto p_1, p_2 do not overlap else goto p_1, p_2 overlap;

The expressions and checks that we produce here are not C code, but rather sets of instructions in a compiler’s intermediate representation that will produce the final values when executed. Our implementation, specifically, outputs expressions in LLVM

Intermediate Representation. Later, in Section 4.2.2, we show how these generated instructions can be converted back to valid C code. The computations generated by our analyses also need to be guarded against type overflow to ensure complete correctness. Many compilers, Clang being one of them, embed overflow checking infrastructure in their intermediate representation. For the polyhedral-based disambiguation technique, shown in the last section and implemented in Polly, overflow checks were added to the produced range checks, without increasing their time complexity. The checks that we insert usually contain sets of instructions that can be simplified. For instance, the sequence in Example 4 computes $v_{tu} - v_{tl}$ twice. One of these operations will be removed by Kennedy’s *redundancy elimination* [Kennedy et al., 1999], a technology readily available in modern compilers. In Section 4.2.2 we provide more details on how the expressions that we produce can be simplified.

3.4 Experiments

To validate the techniques that we discuss in this chapter, we have implemented them on top of the LLVM [Lattner and Adve, 2004] compilation infrastructure and have used it together with Polly⁴. Polly [Grosser et al., 2012] is a loop optimizer built on top of LLVM. It implements typical transformations, such as tiling and loop fusion to improve the target program’s data locality. Several of these transformations are hindered by the lack of aliasing information. The techniques that we introduce in this chapter provide such information.

The polyhedral-based approach discussed in Section 3.3.2 was implemented in Polly revision r236395, from May 3rd, 2015. The other two approaches, purely dynamic and the hybrid version based on our symbolic range analysis, were implemented in Polly revision r216844, from August 31st, 2015. The reason for these two versions is pragmatic: the purely dynamic technique and the hybrid disambiguation based on our range analysis are still research artifacts, so their implementation was always carried in a fixed version of LLVM. The polyhedral-based disambiguation, in the other hand, is currently available in LLVM’s official repositories as production-ready code, thus it must be continuously updated to keep consistency with any changes made to the source of LLVM. Nevertheless, these versions of Polly apply different compiler optimizations on the code that they produce. Thus, our experiments let us know the speedup enabled by a pointer disambiguation technique on top of the original compiler, and the capacity of each technique to identify more optimizable regions. However, they should not be

⁴The versions of LLVM and Polly used in our experiments were obtained from LLVM’s main repository (<http://llvm.org/svn/llvm-project>) at revisions 217065 and 216844, respectively.

used to compare the runtime of the code produced by the three pointer disambiguation approaches, given that the set of optimizations applied on these programs differs.

We have chosen to test our approach on **PolyBench 4.0** [Pouchet, 2014]. This suite is widely adopted by the polyhedral community and used in many works to evaluate the impact memory-related analyses and optimizations. **PolyBench** is composed of benchmarks from different areas, such as data mining, linear algebra, and stencil algorithms. **PolyBench** tests consist of two main parts: (i) the initialization of arrays that will serve as input and output buffers, allocated either in the heap or the stack, and (ii) a kernel that performs the main computation, usually composed of no more than 100 lines of code. **PolyBench** also provides a number of options that can be enabled or disabled at the tester’s will, such as: different dataset sizes, automatic insertion of the `restrict` keyword in pointer parameters, use of scalar values in loop bounds, and a number of cache-related options. These options give us a more controlled environment in which to test our runtime pointer disambiguation strategies. All the numbers that we show have been obtained on an second generation (Sandy Bridge) Intel dual-core i5 with a clock of 1.7GHz. The runtime numbers that we report are obtained on top of LLVM-O3, and are the average of six executions of each benchmark.

Our goal, in this section, is to show that (i) our technique is effective, i.e., it delivers runtime improvement on top of LLVM + Polly-O3; (ii) the amount of code replication that we cause is affordable, given the runtime gains that we bring; and (iii) purely static techniques cannot do better than we do. We address this third point in Section 3.4.1, and leave the other two for Sections 3.4.2 and 3.4.3. A brief discussion comparing our three different pointer disambiguation techniques is presented in Section 3.4.4.

3.4.1 What can LLVM’s Purely Static Approaches Do?

Traditionally, compilers use static alias analyses to disambiguate pointers. It is well-known in the literature that scalable implementations of such analyses are imprecise [Mock et al., 2001]. In this section we support this knowledge with data of our own. We have performed an alias query for every pair of pointers in our benchmarks, using the LLVM implementations of points-to analysis. LLVM uses five different techniques to disambiguate pointers statically:

- *type-based*: C and C++ forbid aliasing between pointers of different types since C89/C++98. Thus, this analysis flags pointers of different types as no-alias;
- *global-refs-based*: relies on the fact that globals that do not have their address taken cannot alias anything;

- *basic*: uses a suite of simple rules, i.e. the stack does not alias the heap or globals, for instance;
- *scalar-evolution-based*: tries to place bounds on arrays, and based on these bounds determines if they may overlap each other or not.
- *Dyck-CFL-based*: implements a context-free language (CFL) based context-insensitive alias analysis. This algorithm is implemented after Zheng and Rugina [2008] and Zhang Zhang et al. [2013].

The precision of queries is cumulative: if any of these five implementations is able to disambiguate two pointers, than they are marked as no aliases. In our experiments, we use all of these analysis in combination.

After applying these static analyses in PolyBench, we found out that each kind of aliasing appears in the following percentages for pairs of pointers: may alias = 62.78%, must alias = 9.92%, no alias = 8.73% and partial alias = 18.57%. For three benchmarks: `lu`, `floyd-warshall` and `seidel-2d`, the may-alias rate is lower than 20%. For six others, it is above 70% and for the rest it is above 50%. The percentages for no-alias and must-alias is remarkably low, below 18% across all the benchmarks. Only 8.7% of queries are no-alias. Hence, if we were to use a purely static approach to disambiguate pointers, then there would be still over 62% of pairs whose relation we would not know.

Notice that the PolyBench kernels do not contain overlapping arrays. Yet, as discussed in the previous paragraphs, typical implementations of static analyses – even in an industrial compiler – are not able to prove this fact. For PolyBench, fully inlining all function calls would be sufficient to give the missing context, but as PolyBench uses an allocation wrapper that is defined in a different translation unit, full inlining is only possible when running link-time optimizations. For libraries, even optimizing at library link-time is insufficient, as library functions can be called from outside the library with parameters that may indeed alias. Hence, optimizing under the assumption of possible aliasing is often necessary to maintain correctness. Inlining function calls or cloning and specializing functions to handle these kind of problems has already been proposed by Metzger and Stroud [Metzger and Stroud, 1993] or by Hall [Hall, 1991, Cp.5]. For cases with insufficient static information (or limited analysis power) mainstream compilers such as LLVM, gcc and icc use code versioning with run-time alias checks for specific use cases, e.g. to prove correctness of inner-loop vectorization, but they commonly do not use function specialization.

Figure 3.11 shows the increase in the number of loops within Static Control Parts (SCoPs) that our three different pointer disambiguation approaches provide. A SCoP is defined in Polly as a Single-Entry-Single-Exit region containing structured control flow

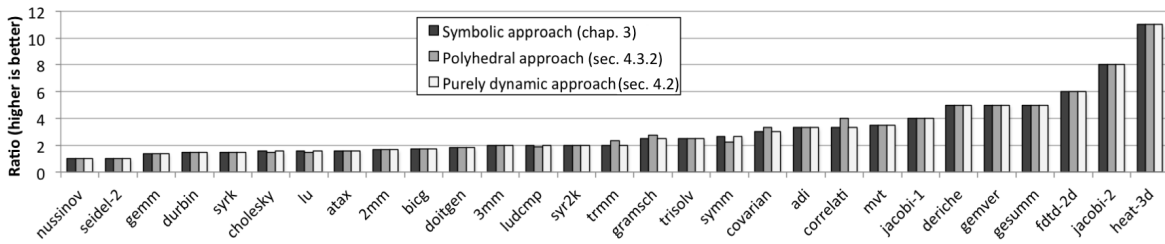


Figure 3.11: Relative increase in the number of loops within Static Control Parts when using our disambiguation approaches. The base values are the number of loops optimized when using only LLVM’s static alias analyses. The three approaches provide the same increase in almost all benchmarks.

(a combination of for-loops and if-then-else blocks) where control flow conditions, loop bounds and memory offsets can be statically modelled as piecewise-affine expressions in loop induction variables and SCoP-invariant variables (parameters). In other words, a SCoP is a region of code that the compiler can safely analyse and optimize. The purely dynamic approach of Section 3.2, and the hybrid approach of Chapter 2, which is based on symbolic range analysis, identify 2.21x more loops within SCoPs than the baseline compiler. Again, the baseline used in this experiment is Polly plus the five static analyses available in LLVM. The polyhedron-based approach of Section 3.3.2 recognized 2.19x more loops within SCoPs than the baseline approach.

This increase is even more substantial when considering only *kernel* functions. A kernel, in the PolyBench jargon, is the function that contains the bulk of the computation that will be performed by a benchmark, thus being our main optimization target. The numbers that we report in Figure 3.11 include loops that are not inside a kernel, e.g., code in charge of initializing arrays or checking results. If we consider only kernels, both the symbolic and purely dynamic approaches increase the number of loops being optimized by 5.88x. The polyhedral-based technique, by its turn, recognizes 5.56x more loops. This small difference between approaches is due to the use of different compiler versions, as explained in the beginning of this section. We note that all three approaches were able to correctly recognize all analyzable loops in the kernels of the PolyBench suite. By “analyzable” we mean the loops that Polly could handle if it had access to perfect aliasing information.

3.4.2 Hybrid Approaches

In this section, we present data that shows that the two hybrid approaches of Section 3.3 are effective and useful. Our goal is to demonstrate that these techniques can bring

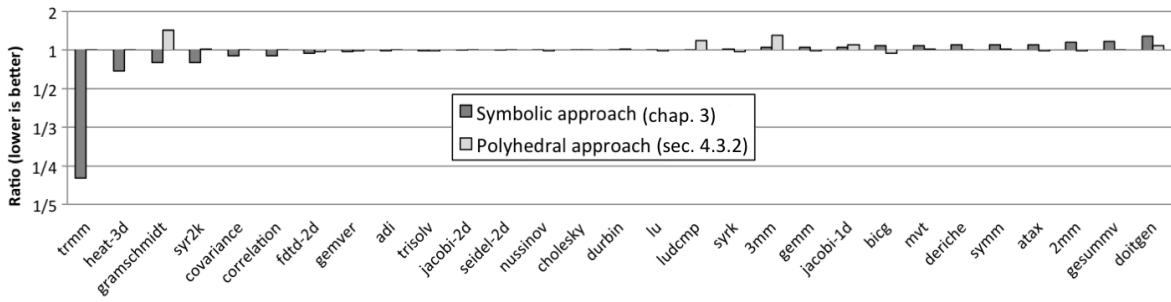


Figure 3.12: Execution time overhead of our hybrid approaches when compared to the use of “restrict” in Polly. Our biggest overhead was in `gramschmidt`, which executes in 5.19 seconds in its “restrict” version. When running on Polly augmented with our polyhedral approach, this number grows to 7.86 seconds, being 1.5x slower, as the chart shows.

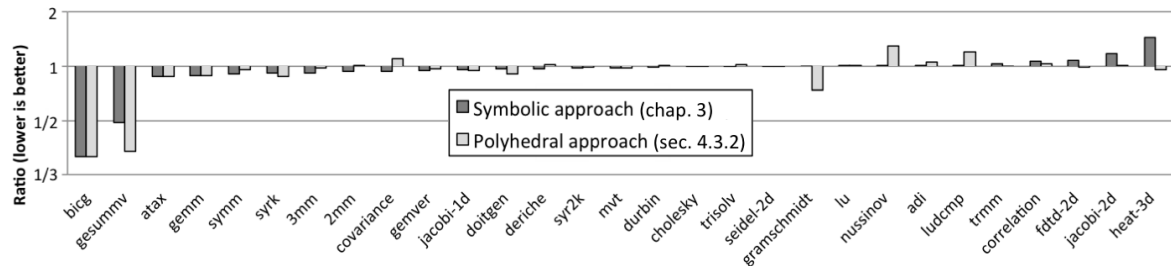


Figure 3.13: Execution time improvement of our hybrid techniques when applied to LLVM-O3. Our biggest speedup was in `bigc`, which is executed in 150 milliseconds by LLVM-O3. Both of our hybrid approaches reduce this number to 56 milliseconds, making it 2.7x faster, as the Figure shows.

speedup on top of highly optimized code, at the expense of an increase in code size.

3.4.2.1 Overhead of dynamic checks

Our dynamic checks incur negligible overhead on the PolyBench programs, as Figure 3.12 reports. To measure this overhead, we have modified Polly to assume absence of aliasing in the source programs. We do it by adding the “restrict” keyword to the arguments of the functions. This keyword, available since C99, tells the compiler that a pointer does not share memory accesses with aliases in the scope of the function in which that pointer is declared. If a function argument p is marked as “restrict”, it is still possible to derive new pointers out of it, such as $p_0 = p + 1$. Thus, by using this keyword, the programmer signs a contract with the compiler, specifying that the memory pointed by p can only be accessed through it, or via one of its derived pointers.

PolyBench comes with an option to enable the “restrict” modifier for parameters

of pointer type. In this case, Polly + LLVM-O3 optimize the same regions that we do, but without having to resort to dynamic bound checks and code duplication. Hence, this experiment lets us check the overhead of our dynamic checks. For a number of benchmarks in Figure 3.12, our hybrid techniques are not as efficient as the “restrict” keyword. In cases like `gesummv` and `ludcmp`, we need to disambiguate a large number of pairs of pointers in the entry of regions that the compiler is not able to aggressively optimize, even though our dynamic checks succeed at runtime. For instance, the polyhedron-based approach of Section 3.3.2 inserts for the kernel function of `ludcmp` a runtime check to disambiguate four base pointers (two read-only, two read-write). This check performs for each of the twelve pointer pairs with at least one read-write base pointer two comparisons as well as a boolean `or` and also requires eleven boolean `and` operations to combine the conditions between pointer-pairs. For `gramschmidt` and `3mm`, the symbolic approach can disambiguate more pointers than its polyhedral-based counterpart. In `doitgen`, the opposite happens, i.e., the polyhedral version can analyze more pointer pairs. This observation justifies the runtime difference observed in these benchmarks. On average we see that the runtime of the polyhedral-based approach (Section 3.3.2) is 2.8% slower and the runtime of the symbolic range analysis based approach (Chapter 2) is 1.6% slower than using “restrict”.

3.4.2.2 Speedup

Our ultimate goal is to speed up programs. We achieve this goal by giving the compiler the opportunity to run more aggressive optimizations on said programs. There are many different ways to test the benefits of our analysis. Compilers commonly provide many options to select the transformations that are run on a given program and LLVM is no exception. We have performed an extensive search in this space, and report findings in this section. First, Figure 3.13 shows how each of our hybrid approaches improves the runtime of LLVM-O3. On average, the polyhedral approach (Section 3.3.2) improves LLVM-O3 by 8.7%. The range-based approach (Chapter 2) gives us a speedup of 6.5%.

It is well known that the order in which optimizations are applied on a program can influence the final runtime of its binary code [Kulkarni et al., 2006]. The numbers reported in Figure 3.13 use the default optimization order of LLVM-O3. Thus, we can go beyond the speedups seen in that figure by changing the order in which some of the LLVM optimizations run. For instance, if we follow the insertion of our checks with a round of loop invariant code motion, then we can hoist more load and store operations outside loops. This strategy lets us convert `array_sum_1` into `array_sum_3`

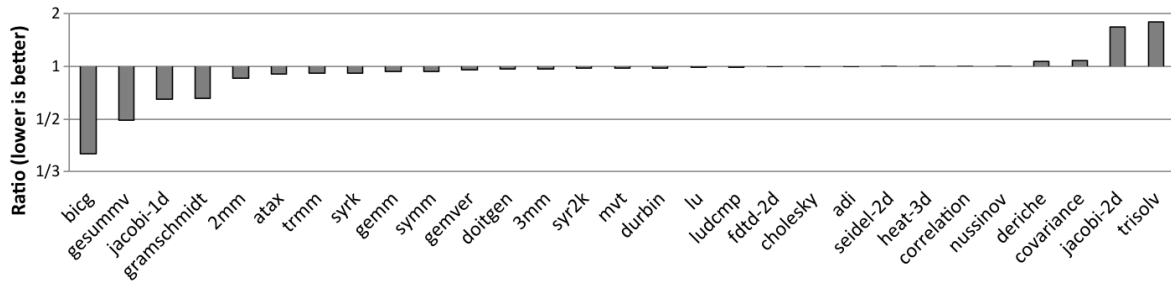


Figure 3.14: Runtime improvement of our symbolic approach (Chapter 2) over LLVM-O3, when followed by a round of loop invariant code motion. As in Figure 3.13, our biggest speedup was in `bicg`, being 2.7x faster.

in Figure 3.1, for instance. Figure 3.14 shows the runtime numbers that we obtain in this way, again, comparing against LLVM-O3. The only change that we performed in this case was to run loop invariant code motion right after inserting the disambiguation checks. This order is the same for all the benchmarks.

We have also evaluated the runtime benefit of our approaches when applied on top of the combined optimization sequence of LLVM-O3 and Polly [Grosser et al., 2012]. The optimizations obtained by Polly’s scheduling optimizer and possibly enabled by our run-time checks can include combinations of classical loop transformations, such as strip mining, tiling, fission, fusion and interchange, but also transformations that are difficult to express as a set of classical loop transformations. Figure 3.15 shows these results. The speedups we report are for the default compile-time options of PolyBench. PolyBench uses by default parametric loop bounds and fixed-sizes arrays. This mismatch requires Polly to respect data-dependences that are only relevant when the parametric loop bounds have a value that is larger than the corresponding fixed size array dimensions. In practice, the loop bounds and the array dimensions always have identical (or clearly related) sizes. When compiling PolyBench with scalar loop bounds (`-DPOLYBENCH_USE_SCALAR_LB`) no spurious data-dependences hinder Polly optimizations and we see for the polyhedral-based approach a speedup of 18.5% comparing LLVM-O3 and Polly with run-time alias checks against LLVM-O3 and Polly without run-time alias checks.⁵

As Figure 3.15 shows, we have observed slowdowns in a few benchmarks: `trisolv`, `gemm`, `atax`, etc. Several of these slowdowns are caused by Polly “optimizing” additional or larger code regions that only become amenable for optimizations when using our new alias-checking techniques. The transformations Polly applies are obtained through

⁵Since commit r222754 (25. November 2014), Polly can take optimistic assumptions to effectively optimize PolyBench in its default configuration.

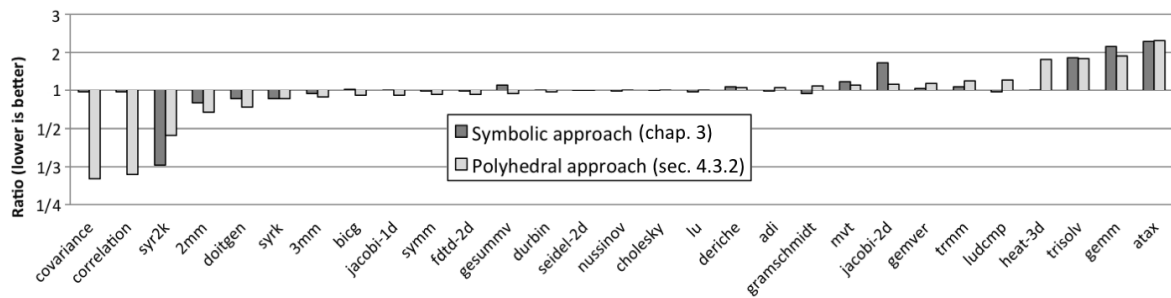


Figure 3.15: Execution time improvement of our hybrid methods over Polly, running on LLVM-O3. Our best result can be observed in `covariance`, which is executed in 4.29 seconds by Polly. Our polyhedral approach lowers this number to 1.29 seconds, being 3.3x faster.

(a slightly modified) reimplementation of the Pluto scheduling optimizer [Bondhugula et al., 2008]. Pluto optimizes the schedule of a SCoP by constructing an ILP problem that minimizes data-dependences while maximizing tilability and parallelism. In this process Pluto *only* considers data-dependences, but does not consider spatial locality. As a result, the Pluto scheduling optimizer may choose schedules that are better according to the criteria Pluto optimizes for, but which reduce spatial locality and consequently performance. The original implementation of Pluto addresses this problem by applying a set of post-scheduling optimizations that focus on spatial locality. The (current) lack of these optimizations in Polly seems to be the main reason for non-optimal code transformation choices and the performance regressions they imply. In this work, the implementation and tuning of program optimizations is not our focus. Instead, our goal is to disambiguate pointers with the lowest possible overhead aiming to increase the applicability of program transformations. The choice of optimal transformations is left to the compiler. Even if we only use LLVM optimizations, but exclude Polly’s, it is still possible to observe slowdowns in some benchmarks. For instance, Figure 3.14 shows slowdowns in two benchmarks - `jacobi-2d` and `trisolv`. We account this behavior to the choice and ordering of optimizations. Finding an optimal spot in this space is still an open problem [Kulkarni et al., 2006].

3.4.2.3 Code Size Expansion

Figure 3.16 shows how much our technique increases the size of binaries. This expansion is due to (i) the duplication of code, and (ii) the checks that we create to disambiguate pointers at runtime. We have not created tests for `nussinov`, and `seidel-2d`, because these benchmarks manipulate only one vector. Our checks are necessary only if the

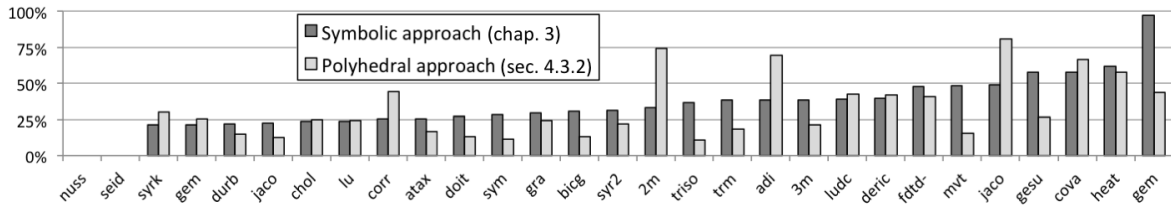


Figure 3.16: Code size expansion due to our optimization.

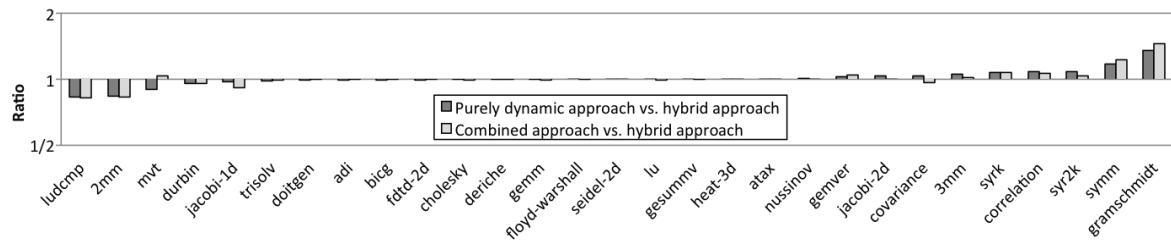


Figure 3.17: Execution time of our purely dynamic (Section 3.2) and combined (dynamic + hybrid) approaches when compared to our hybrid technique (Chapter 2). `ludcmp` executes in 1.27 seconds when using our hybrid method. The purely dynamic approach reduces this number to 1.00 second, being 1.2x faster, as the Figure shows.

optimizable region contains two or more different array accesses. Thus, for these two benchmarks there was no code-size expansion.

3.4.2.4 On the precision of our memory access estimates

The disambiguation checks of our symbolic hybrid approach (Chapter 2) are not always precise, but in some cases over-estimate the size of the accessed memory. We are interested in knowing how precise this over-estimation is. To this end, we have compared our estimates of the lower and upper bounds of the range of accessed array addresses against the results that we obtain using the dynamic approach of Section 3.2. We use our memory allocator in the instrumented programs to check size estimates determined statically against actual memory size. In PolyBench the estimation is perfectly accurate as the accessed regions can be described lossless by both our static analyses.

3.4.3 The purely dynamic approach

Figure 3.17 shows a runtime comparison between the purely dynamic approach, discussed in Section 3.2, and the hybrid approach with the symbolic range tests seen in Chapter 2. We show results for PolyBench. Overall, the runtime of the benchmarks was very similar. We have observed that the hybrid approach is 6% faster. This number

is the geometric mean of the execution times. The hybrid approach is faster because, in general, its dynamic checks have a faster runtime, and it does not impose memory allocation and deallocation overheads. Whereas the purely dynamic checks are $O(\ln n)$, n being the number of memory allocations currently live in the program, the checks inserted by the hybrid approach execute in $O(1)$. Nevertheless, this constant can be high, because these guards may be formed by the combination of several complex arithmetic expressions. This observation explains why the purely dynamic methodology yields faster runtimes in some benchmarks. Additionally, the hybrid approach has been used to disambiguate more pointer pairs. This happened whenever we could not infer statically symbolic bounds for some pointers.

We have tried also to combine the purely dynamic approach and the hybrid approach of Chapter 2. In this case, whenever the hybrid approach could not generate a test to disambiguate a pointer pair, we would resort to the purely dynamic disambiguation. Figure 3.17 also shows these results. This combination has produced slight improvement on the runtime of the purely dynamic approach. We believe that we could not observe a larger speedup because a substantial part of the cost of this technique is due to the memory allocation overhead. For instance, we tried the purely dynamic approach in SPEC CPU 2006's 401.bzip2, a memory intensive benchmark Henning [2006]. In this allocation-heavy benchmark, the purely dynamic approach slowed the program down by 29%. Notice, however, that the purely dynamic technique does not depend on a particular implementation of a memory allocator. Thus, we believe that an interesting line of future research is to check if, and by how much, other implementations, such as SoftBounds [Nagarakatte et al., 2009], for instance, could reduce this overhead.

3.4.4 Discussion

In this chapter, we have discussed very different techniques to disambiguate pointers at runtime: the purely dynamic approach of Section 3.2, and the hybrid approach of Section 3.3. Additionally, we have used two different methods to generate the checks used in the hybrid approach, the polyhedrons of Section 3.3.2, and the symbolic range analysis of Chapter 2. All these techniques have advantages and shortcomings, and, to illustrate them, we shall rely on the programs seen in Figure 3.18.

The purely dynamic approach is more applicable, handling sparse data structures such as linked-lists, for instance. Furthermore, it does not depend on the ability of a static analyzer to infer bounds for arrays. For example, only the purely dynamic approach lets us produce tests for function `f0` in Figure 3.18. The hybrid approaches

<pre> 1 void f3(int *a, int *b, int N) { 2 int i, j; 3 for (i = 0; i < N; i++) { 4 for (i = 0; i < N; i++) { 5 a[i*j] += b[i]; 6 } 7 } 8 } </pre>	<pre> ✓ ✗ ✓ </pre>
<pre> 1 void f2(int *u, int *v, int N) { 2 int i; 3 for (i = 0; i < N; i++) { 4 if (N < 255) 5 u[i] = i; 6 else 7 v[i] = i; 8 } 9 } </pre>	<pre> ✓ ✓✓ ✓ </pre>
<pre> 1 void f1(char *u, char* v, int N) { 2 int i, j; 3 for (i = 0; i < N; i++) { 4 u[i] = 0; 5 for (j = 0; j < N; j++) { 6 u[i] += v[j]; 7 } 8 } 9 } 10 11 int main() { 12 const int N = 100; 13 char *v = malloc(2*N*sizeof(int)); 14 f1(v, (v+N), N); 15 return 0; 16 } </pre>	<pre> ✗ ✓ ✓ </pre>
<pre> 1 void f0(int* src, int *acc) { 2 int i; 3 *acc = 0; 4 for (i = 0; src[i]; i++) { 5 *acc += src[i]; 6 } 7 } </pre>	<pre> ✓ ✗ ✗ </pre> <p style="text-align: center;"> purely dynamic symbolic polyhedral-based </p>

Figure 3.18: Examples of programs that illustrate advantages and disadvantages of our different runtime pointer disambiguation techniques.

would fail in this case because the limits of the array `src` are not explicit in the loop. The purely dynamic technique can also handle loops containing function calls and non-affine array accesses, which are not addressed by the other strategies.

The main drawback of the purely dynamic method is the query time. Our implementation of it uses a balanced tree to store meta-information associated with each chunk of memory allocated during the execution of a program. In principle, we could use other data structures to retrieve the meta-data associated with the blocks of al-

located memory. We chose to implement our memory allocator with a red-black tree because it is relatively easy to implement and it offers us reliable access times. Furthermore, this data structure has been already used for similar purposes in the literature [Margiolas and O’Boyle, 2014]. Nevertheless, for memory intensive programs, the red-black tree’s search time might bring in a non-negligible overhead. In its current form the purely dynamic approach is completely ignorant of the underlying memory it wraps, as long as it provides a `libc` like interface. While this makes our implementation trivially portable between different platforms and allocators, it also means that we have to duplicate some metadata that most modern allocators already keep. In the future we plan to modify a high performance allocator to remove the overhead of tree queries for a large part of all allocations and significantly reduce the space overhead.

The hybrid approaches of Section 3.3 have lower overhead and work independently of the program’s execution environment. Our experiments indicate that inferring bounds for pointers statically should be preferred whenever possible. There are also programs that the hybrid approaches can disambiguate, but the dynamic technique cannot. For instance, the latter method will not be able to disambiguate the accesses to `u[i]` and `v[j]` in function `f1` in Figure 3.18 (line 6), because these pointers dereference addresses within the same allocated region, even though they operate over disjoint address ranges.

In our experiments, we have not found programs that could be analyzed differently by one of the two hybrid approaches of Section 3.3, but not for the other. Nevertheless, we can create such examples by hand. For instance, both methods are able to analyze function `f2` in Figure 3.18; however, the polyhedron-based one, from Section 3.3.2, will be able to infer – statically – that the accesses `u[i]` and `v[i]` are independent, because for a given set of parameters either line 5 or line 7, but never both, can be run during the execution of the kernel. On the other hand, this technique is not able to produce tests for the program in function `f3` in Figure 3.18. The culprit, in this case, is the fact that the expression `i * j` is non-affine. The approach based on symbolic range analysis can handle this example, although this feature is not yet available in our implementation.

Chapter 4

DawnCC: Automatic Annotation for Data Parallelism and Offloading

In this chapter, we present a second, more sophisticated client of our analysis: DawnCC; a framework that automatically annotates C and C++ code to run on a GPU. This tool relies on the symbolic access range analysis from Chapter 2 to emit directives capable of offloading memory regions to an external device.

4.1 Overview

We use the *Single Precision AX + Y* (SAXPY) kernel in Figure 4.1 (a) to illustrate the contributions of this chapter. This kernel is a standard function in Nvidia’s BLAS library¹. It simply performs a combination of multiplication by scalar plus addition between corresponding cells of two vectors. It runs in linear time on a sequential machine. However, it is $O(1)$ in the *Parallel Random-Access Machine* (PRAM) model, because there is no dependency between different iterations of the loop. In the high-performance computing jargon, the SAXPY loop is called a *doall*.

Figure 4.1 (b) shows a direct translation of SAXPY to C for CUDA. CUDA’s syntax is very similar to C’s; however, its semantics is substantially different. Part of it, lines 1-7, is meant to run on a GPU; the rest, lines 9-11, is meant to run on a host CPU. The code that runs on the GPU will be instantiated multiple times, once per each logical thread. In this case, we have one thread per each valid index in the input vectors. Even though C for CUDA is becoming commonplace among developers of parallel applications, having to worry about concurrent semantics and communication

¹<https://devblogs.nvidia.com/parallelforall/six-ways-saxpy/>

<pre> 1 void 2 saxpy_serial(int n, float alpha, float *x, float *y) { 3 for (int i = 0; i < n; i++) { 4 y[i] = alpha*x[i] + y[i]; 5 } 6 } </pre> <p>(a)</p>	<pre> 1 __global__ void 2 saxpy_parallel(int n, float alpha, float *x, float *y) { 3 int i = blockIdx.x * blockDim.x + threadIdx.x; 4 if (i < n) { 5 y[i] = alpha * x[i] + y[i]; 6 } 7 } 8 ... 9 // Invoke the parallel kernel: 10 int nblocks = (n + 255) / 256; 11 saxpy_parallel <<<nblocks, 256>>>(n, 2.0, x, y); </pre> <p>(b)</p>
--	--

Figure 4.1: (a) Standard C implementation of the Single Precision $AX + Y$ (SAXPY) kernel. (b) Same algorithm written in C for CUDA.

<pre> 1 void saxpy_serial(int n, float alpha, float *x, float *y) { 2 long long int tmp[2]; 3 tmp[0] = n - 1; 4 tmp[1] = ((tmp[0] > 0) ? tmp[0] : 0); // upper bound 5 6 char x_y_alias_free = ((x >= y + tmp[1] + 1) 7 (y >= x + tmp[1] + 1)); 8 9 #pragma acc data pcopy(y[0:tmp[1]]) \ 10 pcopyin(x[0:tmp[1]]) \ 11 if(x_y_alias_free) 12 #pragma acc kernels loop independent \ 13 if(x_y_alias_free) 14 for (int i = 0; i < n; i++) 15 y[i] = alpha*x[i] + y[i]; 16 } </pre> <p>(a)</p>	<pre> 1 void saxpy_serial(int n, float alpha, float *x, float *y) { 2 long long int tmp[2]; 3 tmp[0] = n - 1; 4 tmp[1] = ((tmp[0] > 0) ? tmp[0] : 0); // upper bound 5 6 char x_y_alias_free = ((x >= y + tmp[1] + 1) 7 (y >= x + tmp[1] + 1)); 8 9 #pragma omp target data map(to:x[0:tmp[1]]) \ 10 map(tofrom:y[0:tmp[1]]) \ 11 if(x_y_alias_free) 12 #pragma omp parallel for if(x_y_alias_free) 13 for (int i = 0; i < n; i++) 14 y[i] = alpha*x[i] + y[i]; 15 } </pre> <p>(b)</p>
---	--

Figure 4.2: (a) SAXPY annotated with OpenACC pragmas. (b) SAXPY annotated with OpenMP pragmas. The gray area denotes code created automatically.

between multiple devices still restricts the use of this language.

To make GPUs more accessible to the everyday developer, the high-performance computing community has designed a number of *annotation systems*. An annotation system is a meta-language that changes the semantics of a host language. In our setting, the host language is either C or C++, and the meta-language is either OpenACC or OpenMP. Figure 4.2 shows the sequential SAXPY kernel annotated with (a) OpenACC and (b) OpenMP pragma directives. Our DawnCC compiler inserts these pragmas, plus all the code necessary for them to work, automatically.

DawnCC is a source-to-source compiler: it reads an ordinary C program and produces a version of that program with annotations. In this process, DawnCC solves two problems. First, it recognizes *doall* loops. Second, it inserts primitives to copy data to and from the GPU. To deal with the first problem, the identification of *doall*

loops, we use standard compiler analysis techniques [Wolfe, 1995, Ch.6]. Because these techniques are already commonplace in the compiler’s literature, in this work we focus on the second problem.

To insert data movement directives, such as `pcopy` in Figure 4.2 (a) and `map` in Figure 4.2 (b), we need to infer the bounds of memory regions. In this example, memory regions are the arrays `x` and `y`. We use the range inference from Chapter 2 and the static analyses described in Section 4.2 to recover these limits. More importantly: we do it using symbols present in the program code itself. For instance, at line 3 of Figures 4.2 (a) and (b), we are using the symbol `n` to rebuild the limits of the arrays `x` and `y`. These limits not only give us a way to copy data around, but they also let us show that pointers do not overlap. In our example, the tests at line 6 of Figures 4.2 (a) and (b), built using the hybrid pointer disambiguation of Chapter 3, give us this information. Whenever either of the inequalities $y \geq x + n + 1$ or $x \geq y + n + 1$ are true, we are sure that the vectors `x` and `y` do not overlap. We can only assume that the loop is parallel once we are under this assumption. We emphasize that these annotations have been produced without any intervention from a user. How we perform such deed is the subject of the next section.

4.2 Static Analyses

In addition to the symbolic range inference of Chapter 2, DawnCC is built around a number of other static analyses, such as the dependence analysis of Ferrante et al. [1987]. In the rest of this section we explain how these compilation techniques work. Figure 4.3 shows how the different analyses are related to each other in the compilation flow of DawnCC.

4.2.1 Memory Disambiguation

The fact that we can infer the sizes of arrays gives us the possibility to disambiguate pointers, as shown in Chapter 3. This disambiguation lets us eliminate spurious dependencies in the source program due to pointer aliasing. Thus, it lets us potentially increase the number of parallel regions that we can find. As described in Section 3.3.1, if $B_1 = [l_1, u_1]$ and $B_2 = [l_2, u_2]$ are estimations for the regions dereferenced by two different pointers p_1 and p_2 , then we say that they will not overlap if:

$$p_1 + l_1 \geq p_2 + u_2 + 1 \text{ or } p_2 + l_2 \geq p_1 + u_1 + 1$$

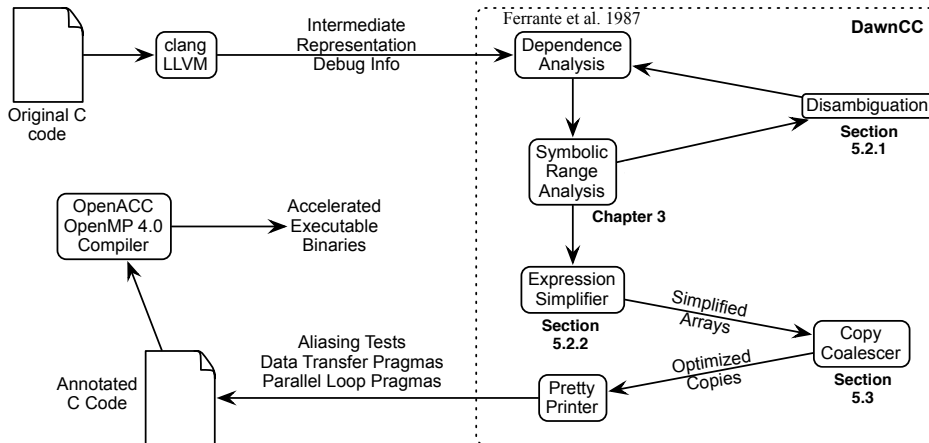


Figure 4.3: Overview of DawnCC.

This test ensures that the regions covered by offsets that use p_1 and p_2 as base pointers have empty intersection. For instance, if each element pointed by p_1 takes 8 bytes, p_2 cannot point to the 4th byte of the last memory position accessed through p_1 , whenever either of the above inequalities hold.

Both OpenACC and OpenMP give us the equipment necessary to use this information. Such equipment consists in *conditional directives*. A conditional directive only takes effect if a given predicate is valid at runtime. Thus, whenever we might have aliasing between different pointers within a loop, we use conditional tests to disambiguate them, as we illustrate in Example 5.

Example 5 *The loop in `saxpy_serial`, in Figure 4.1 (a), contains accesses to two pointers: x and y . Both have bounds $[0, n - 1]$. Thus, they will not alias if $x \geq y + (n - 1) + 1$, or if $y \geq x + (n - 1) + 1$. This test is implemented at lines 6 and 7 of Figures 4.2 (a) and (b). The conditional pragmas that use the result of this test appear at lines 9 through 12 of the annotated programs.*

4.2.2 From Source to IR, and Back Again

DawnCC is built on top of the LLVM [Lattner and Adve, 2004] compilation framework, whose intermediate representation is used as input for the static analyses presented in this section. We chose to perform our analyses on the *Intermediate Representation* (IR) of LLVM because we could, in this way, reuse already available analyses. However, such benefit comes with a challenge: There exists a gap between LLVM’s IR and the source code, and while we analyze the former, our annotations must, ultimately, be inserted in the latter. This section explains how we have bridged this gap, using analysis

```

1 void saxpy_3off(int n, float alpha, int *x, float *y) {
2   for (int i = 3; i < n; ++i)
3     y[i-3] = alpha*x[i] + y[i-3];
4 }

1 void saxpy_3off(int n, float alpha, int *x, float *y) {
2   char x_y_alias_free = ((x+3) >= (y + max(0,n-4) + 1)) || (y >= (x + max(3,n-1) + 1));
3
4   #pragma omp target map(x[3:max(3,n-1)-3+1], y[0:max(0,n-4)+1]) if(x_y_alias_free)
5   #pragma omp parallel for if(x_y_alias_free)
6   for (int i = 3; i < n; ++i)
7     y[i-3] = alpha*x[i] + y[i-3];
8 }

```

Figure 4.4: (a) example adapted from our original SAXPY code (Figure 4.1 (a)) and (b) a version of it annotated with OpenMP 4.0 pragmas to run on a GPU.

developed by Gleison Mendonça, from UFMG Compilers Laboratory [Mendonça et al., 2016]. While we focus on details of the LLVM’s IR as a means to provide the reader with concrete examples, our ideas fit scenarios made of high-level languages other than C/C++, and low-level languages other than LLVM bytecodes.

In low-level assemblies, array indexing expressions consist of *load* or *store* instructions. These instructions take as operand a memory address. Thus, finding the access bounds for a given array means finding the bounds for the target addresses of each load and store operation that can manipulate that array. Figure 4.4 (a) shows a small variation of the SAXPY code seen in Figure 4.1 (a). Figure 4.5 (a) outlines the LLVM instructions generated for the loop body² in Figure 4.4 (a), line 3. The instruction `GetElementPtr` computes the actual address of an element, given a base address and the element’s index. The *load* and *store* operations in lines 13 and 21 of Figure 4.5 (a) represent the accesses to array *y*. The *load* in line 4 was translated from the access to *x*. If we apply the techniques from Chapter 2 on this program, then we get the code in Figure 4.5 (b). These assembly instructions, when executed at runtime, yield the lowest and highest addresses referenced through arrays *x* and *y* in our example, i.e., their access bounds.

Converting the low-level representation of array access bounds back into source code is relatively easy: most operators in LLVM IR have a one-to-one mapping in C or can be emulated by a small set of C operations. Using a bottom-up recursive conversion strategy over the assembly in Figure 4.5 (b), we get the equivalent parenthesized C expressions in Figure 4.6 (a). While these expressions meet the goal of expressing

²Names preceded by % represent virtual registers created by the compiler

<pre> 1 ; read x[i] 2 %2 = sext i32 %i to i64 3 %3 = getelementptr i32, i32* %x, i64 %2 4 %4 = load i32, i32* %3, align 4 5 6 %5 = sitofp i32 %4 to float 7 %6 = fmul float %alpha, %5 8 9 ; read y[i-3] 10 %7 = sub nsw i32 %i, 3 11 %8 = sext i32 %7 to i64 12 %9 = getelementptr float, float* %y, i64 %8 13 %10 = load float, float* %9, align 4 14 15 %11 = fadd float %6, %10 16 17 ; write y[i-3] 18 %12 = sub nsw i32 %i, 3 19 %13 = sext i32 %12 to i64 20 %14 = getelementptr float, float* %y, i64 %13 21 store float %11, float* %14, align 4 22 23 br label %inc </pre> <p style="text-align: center;">(a)</p>	<pre> 1 ; lower bound for x: &(x[3]) 2 %1 = getelementptr i32, i32* %x, i64 3 3 %2 = bitcast i32* %1 to i8* 4 5 ; upper bound for x: &(x[max(3,n-1)]) 6 %3 = add i32 %n, -1 7 %4 = icmp sgt i32 %3, 3 8 %5 = select i1 %4, i32 %3, i32 3 9 %6 = add i32 %5, -3 10 %7 = zext i32 %6 to i64 11 %8 = shl nuw nsw i64 %7, 2 12 %9 = getelementptr i32, i32* %x, i64 3 13 %10 = ptrtoint i32* %9 to i64 14 %11 = add i64 %8, %10 15 %12 = inttoptr i64 %11 to i8* 16 17 ; lower bound for y: &(y[0]) 18 %13 = bitcast float* %y to i8* 19 20 ; upper bound for y: &(y[max(0,n-4)]) 21 %14 = add i32 %n, -1 22 %15 = icmp sgt i32 %14, 3 23 %16 = select i1 %15, i32 %14, i32 3 24 %17 = add i32 %16, -3 25 %18 = zext i32 %17 to i64 26 %19 = shl nuw nsw i64 %18, 2 27 %20 = ptrtoint float* %y to i64 28 %21 = add i64 %19, %20 29 %22 = inttoptr i64 %21 to i8* </pre> <p style="text-align: center;">(b)</p>
---	---

Figure 4.5: (a) LLVM instructions generated for the body of the loop in line 3 of Figure 4.4 (a); (b) LLVM assembly representing the symbolic access bounds of arrays x and y .

<p>(a)</p> <pre> lower bound for x: (void*)&(x[3]) upper bound for x: (void*)(((((int64_t)((((n-1)>3)?(n-1):3)-3))<<2)+((int64_t)&(x[3]))) lower bound for y: (void*)y upper bound for y: (void*)(((((int64_t)((((n-1)>3)?(n-1):3)-3))<<2)+((int64_t)y))) </pre>	<p>(b)</p> <table border="0" style="width: 100%;"> <tr> <td style="width: 50%;"> <pre> (1) (void*)(((((int64_t)(((((n-1)>3)?(n-1):3)-3))<<2)+((int64_t)&(x[3]))) </pre> </td> <td style="width: 50%;"> <pre> (5) (void*)((int64_t)&(x[max(n-1,3)])) </pre> </td> </tr> <tr> <td style="text-align: center;"> $\underbrace{\hspace{10em}}_{\text{max}(n-1,3)}$ </td> <td style="text-align: center;"> $\underbrace{\hspace{10em}}_{\text{void}^*}$ </td> </tr> <tr> <td style="width: 50%;"> <pre> (2) (void*)(((((int64_t)(max(n-1,3)-3))<<2)+((int64_t)&(x[3]))) </pre> </td> <td style="width: 50%;"> <pre> (6) (void*)&x[max(n-1,3)] </pre> </td> </tr> <tr> <td style="text-align: center;"> $\underbrace{\hspace{10em}}_{*4}$ </td> <td style="text-align: center;"> $\underbrace{\hspace{10em}}_{\text{(void}^*)x+\text{max}(n-1,3)}$ </td> </tr> <tr> <td style="width: 50%;"> <pre> (3) (void*)(((((int64_t)(max(n-1,3)-3))*4)+((int64_t)&(x[3]))) </pre> </td> <td style="width: 50%;"> <pre> (7) (void*)(x+max(n-1,3)) </pre> </td> </tr> <tr> <td style="text-align: center;"> $\underbrace{\hspace{10em}}_{\text{(void}^*)((\text{int64}_t)\&x[\text{max}(n-1,3)-3+3])}$ </td> <td style="text-align: center;"> $\underbrace{\hspace{10em}}_{\text{max}(n-1,3)}$ </td> </tr> <tr> <td style="text-align: center;"> $\underbrace{\hspace{10em}}_0$ </td> <td></td> </tr> </table>	<pre> (1) (void*)(((((int64_t)(((((n-1)>3)?(n-1):3)-3))<<2)+((int64_t)&(x[3]))) </pre>	<pre> (5) (void*)((int64_t)&(x[max(n-1,3)])) </pre>	$\underbrace{\hspace{10em}}_{\text{max}(n-1,3)}$	$\underbrace{\hspace{10em}}_{\text{void}^*}$	<pre> (2) (void*)(((((int64_t)(max(n-1,3)-3))<<2)+((int64_t)&(x[3]))) </pre>	<pre> (6) (void*)&x[max(n-1,3)] </pre>	$\underbrace{\hspace{10em}}_{*4}$	$\underbrace{\hspace{10em}}_{\text{(void}^*)x+\text{max}(n-1,3)}$	<pre> (3) (void*)(((((int64_t)(max(n-1,3)-3))*4)+((int64_t)&(x[3]))) </pre>	<pre> (7) (void*)(x+max(n-1,3)) </pre>	$\underbrace{\hspace{10em}}_{\text{(void}^*)((\text{int64}_t)\&x[\text{max}(n-1,3)-3+3])}$	$\underbrace{\hspace{10em}}_{\text{max}(n-1,3)}$	$\underbrace{\hspace{10em}}_0$	
<pre> (1) (void*)(((((int64_t)(((((n-1)>3)?(n-1):3)-3))<<2)+((int64_t)&(x[3]))) </pre>	<pre> (5) (void*)((int64_t)&(x[max(n-1,3)])) </pre>														
$\underbrace{\hspace{10em}}_{\text{max}(n-1,3)}$	$\underbrace{\hspace{10em}}_{\text{void}^*}$														
<pre> (2) (void*)(((((int64_t)(max(n-1,3)-3))<<2)+((int64_t)&(x[3]))) </pre>	<pre> (6) (void*)&x[max(n-1,3)] </pre>														
$\underbrace{\hspace{10em}}_{*4}$	$\underbrace{\hspace{10em}}_{\text{(void}^*)x+\text{max}(n-1,3)}$														
<pre> (3) (void*)(((((int64_t)(max(n-1,3)-3))*4)+((int64_t)&(x[3]))) </pre>	<pre> (7) (void*)(x+max(n-1,3)) </pre>														
$\underbrace{\hspace{10em}}_{\text{(void}^*)((\text{int64}_t)\&x[\text{max}(n-1,3)-3+3])}$	$\underbrace{\hspace{10em}}_{\text{max}(n-1,3)}$														
$\underbrace{\hspace{10em}}_0$															

Figure 4.6: (a) C code generated for the symbolic bounds in Figure 4.5 (b); (b) steps used to improve the readability of expressions that describes upper bound of x .

access bounds in C code, they have two shortcomings. First, they are hard to read, a fact that would decrease the maintainability of code automatically annotated. Second, expressions used in OpenAcc/OpenMP’s memory transfer directives need bounds given in terms of base addresses and integer access indexes. However, the expressions in Figure 4.6 (a) use pointer arithmetic to compute actual addresses. In what follows, we explain how we overcome these two problems.

Simplifying Bound Expressions To make limit expressions more readable, we perform a series of static simplifications. Most of these simplifications take advantage of operations that are common to pointer arithmetic and address manipulation in C. We explain below the transformations that we perform, exemplifying some of them in Figure 4.5. This Figure shows, step by step, how the expression that computes the upper bound for array x (Figure 4.6 (a)) can be simplified. Even though we illustrate these transformations with C code, they are performed over the equivalent LLVM IR representing the expression (Figure 4.5 (b)). The list of simplifications is as follows:

- **conversion to *min* and *max* operations:** limit expressions usually involve a number of *minimum* and *maximum* operations. These, however, are usually represented as a *less than* or *greater than* comparison followed by a selection instruction in LLVM IR (e.g., lines 7-8 of Figure 4.5 (b)), or an equivalent conditional ternary operation in C. Reducing this representation to a simple *min* or *max* operation makes the code easier to read. Figure 4.6 (b) step (1) shows an example of this reduction.
- **static resolution of conditionals:** oftentimes our range analysis generates conditional operations based on relational expressions that can be trivially solved, e.g. $(n < n + 1)$. Whenever possible, we solve these conditionals statically, eliminating the remaining dead branch.
- **shift to *mul* and *div* conversion:** most compilers convert multiplication or division by powers of 2 into shift operations. Whenever possible, we undo this optimization, as explicit multiplications and divisions make it easier for us to identify pointer manipulation patterns. Step (2) of Figure 4.6 (b) illustrates this simplification.
- **simplification of array indexing:** computing the address of an element in a one-dimensional array involves (i) finding a base address and (ii) computing an offset. If we identify this pattern in a set of instructions, then we can replace it by the equivalent C expression using the brackets notation. Figure 4.6 (b)’s step (3) demonstrates this simplification when applied onto the array x .

- **constant propagation:** we solve any arithmetic operation that can be resolved statically. E.g., Figure 4.6 (b) step (4).
- **extraction of common subexpressions:** at times, the symbolic access bounds of different arrays use the same intermediate subexpressions. When this is the case, we extract such expressions to temporary variables.

From Symbolic Bounds to Annotations The expressions generated by our analysis compute the lowest and highest addresses that can be accessed in an array. To insert the final directives, however, we need the lowest and highest integer values used to index the array. After simplification, we often end up with a simple indexing expression (Figure 4.6 (b) step (6)) from which we can easily tell the integer index apart (steps (7) and (8)). Even for cases where the final bound expression is not as simple, we can still compute the limit index by subtracting the base address of the array and dividing the result by its type size. Once we obtain readable integer indexes for all the access bounds, we proceed to generate the annotated code seen in Figure 4.4 (b).

The aliasing test in line 2 of Figure 4.4 (b), which checks that arrays x and y do not overlap, can be obtained by inlining the symbolic access bounds that we have just computed into the restrictification inequality defined in Section 4.2.1. For data transfer pragmas, however, the lowest and highest index are not enough: mapping clauses in both OpenACC and OpenMP 4.0 determine the memory to be copied by specifying a start index and an integer length. This offset determines how many memory positions of an array should be transferred to the device, counting from the starting index. While the start index will be the same as the lowest access index that we got using our analysis, the length will be the number of memory positions between the access limits. In other words, the length is the difference between the highest and lowest access indexes plus one. The *map* clauses in line 4 of Figure 4.4 (b) show the resulting transfer ranges derived for our example.

4.3 Data Transfer Optimizations

Instead of mimicking the work of programmers, accurate static analyses let us go beyond what a human user can achieve with code annotation systems. The usual workflow followed by a developer when annotating large programs for GPU parallelization is to (i) reason about each loop nest in separate, (ii) decide if it should or not be sent to the external acceleration device and (iii) insert data transfer and parallel pragmas accordingly. This modus operandi is justified because loop nests contain most of the parallelization opportunities in a program, and are usually small enough to be

```

1 void corr(float *A, float *MEAN, float *STDEV, int m, int n) {
2   for (int i = 0; i < m; i++) {
3     for (int j = 0; j < n; j++)
4       MEAN[j] += A[i*n+j];
5
6     MEAN[j] /= n;
7   }
8
9   for (int i = 0; i < m; i++) {
10    for (int j = 0; j < n; j++)
11      STDEV[j] += (A[i*n+j] - MEAN[j]) * (A[i*n+j] - MEAN[j]);
12
13    STDEV[j] = sqrt(STDEV[j] / n);
14  }
15 }

```

(a)

```

1 void corr(float *A, float *MEAN, float *STDEV, int m, int n) {
2   #pragma omp target map(to: A[:m*n]) \
3     map(tofrom: MEAN[:m])
4   #pragma omp parallel for
5   for (int i = 0; i < m; i++) {
6     for (int j = 0; j < n; j++)
7       MEAN[j] += A[i*n+j];
8
9     MEAN[j] /= n;
10  }
11
12  #pragma omp target map(to: A[:m*n], MEAN[:m]) \
13    map(tofrom: STDEV[:m])
14  #pragma omp parallel for
15  for (int i = 0; i < m; i++) {
16    for (int j = 0; j < n; j++)
17      STDEV[j] += (A[i*n+j] - MEAN[j]) * (A[i*n+j] - MEAN[j]);
18
19    STDEV[j] = sqrt(STDEV[j] / n);
20  }
21 }

```

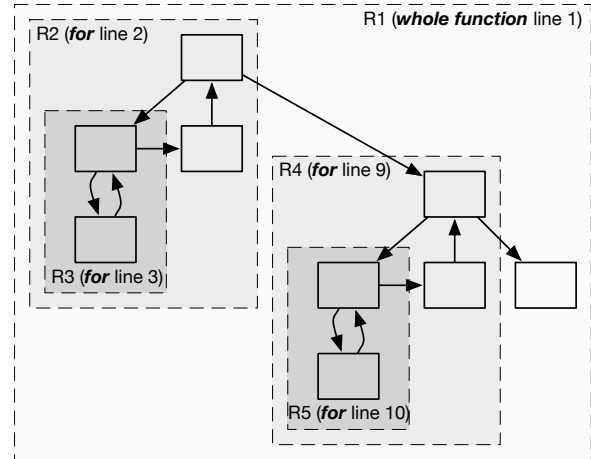
(b)

```

1 void corr(float *A, float *MEAN, float *STDEV, int m, int n) {
2   #pragma omp target data map(to: A[:m*n]) \
3     map(tofrom: MEAN[:m], STDEV[:m])
4   {
5     #pragma omp target
6     #pragma omp parallel for
7     for (int i = 0; i < m; i++) {
8       for (int j = 0; j < n; j++)
9         MEAN[j] += A[i*n+j];
10
11     MEAN[j] /= n;
12   }
13
14   #pragma omp target
15   #pragma omp parallel for
16   for (int i = 0; i < m; i++) {
17     for (int j = 0; j < n; j++)
18       STDEV[j] += (A[i*n+j] - MEAN[j]) * (A[i*n+j] - MEAN[j]);
19
20     STDEV[j] = sqrt(STDEV[j] / n);
21   }
22 }
23 }

```

(c)



(d)

Figure 4.7: (a) example program adapted from the *correlation* benchmark, (b) code produced following a simple per-loop annotation approach, (c) code produced using the *data environment* feature of OpenMP 4.0, and (d) CFG for the program divided into SESE regions.

amenable to human reasoning. Nevertheless, complex syntax and intricate iteration spaces might cause developers to use redundant annotations in the effort to parallelize programs. The function `corr` in Figure 4.7 (a), which was adapted from the *correlation* benchmark in the PolyBenchGPU suite, gives us the opportunity to illustrate some of these difficulties, as we explain in Example 6.

Example 6 *The approach described in Section 4.2.2, once applied onto function `corr`*

in Figure 4.7 (a), gives us the code in Figure 4.7 (b). This second version of *corr* executes all loop nests in the accelerator. As indicated by the directive in lines 2-3 of the new program, the contents of arrays *MEAN* and *A* are sent to the device before the loop in lines 5-10 of Figure 4.7 (b). *MEAN*, the output array, is brought back after that loop finishes. For the second loop nest, *MEAN*, *A*, and *STDEV* are sent to the GPU and *STDEV* is brought back. This gives us a total of seven transfer operations.

Annotating loop nests as completely separate objects may cause a developer to miss optimization opportunities. One such opportunity is the reuse of memory transfer operations across different nests of loops. Data transfer operations may impose a prohibitive overhead when offloading code to a GPU [Gregg and Hazelwood, 2011]. Not sending to the external device the contents of arrays used solely as computation output, or not bringing back input data can reduce this overhead. In this work we go one step beyond: we coalesce data transfers of loops that operate on the same data. For instance, in Figure 4.7 (b) both parallel loop nests operate over the array *MEAN*; thus, it would be desirable to keep this memory region in the external device during the execution of the whole function, rather than bringing it back between the loops, contrary to what has been shown in Example 6. To achieve this end, both OpenMP 4.0 and OpenACC allow the user to explicitly define a *data environment* in the target device. A data environment is a syntactic region that determines a set of memory mappings between host and device, which are valid for any parallel region within the environment block. In C/C++, data environments are *scoped blocks*, i.e., a region delimited by braces. Example 7 shows the benefits of this optimization.

Example 7 Figure 4.7 (c) contains a data environment ranging from line 4 till line 22. In this new version of *corr*, the pragma in lines 2-3 states that arrays *MEAN*, *STDEV*, and *A* must be sent to the GPU, and that *MEAN* and *STDEV* must be brought back. Transfers to the GPU happen, semantically, at line 4. Transfers from the GPU happen at line 22.

The reimplementations of Example 6, seen in Example 7, saves two data transfer operations. This example shows a specific case of a general optimization, which we call *Coalescing of Data Transfer Operations*. The goal of this optimization is to automatically encompass as many different parallel loops as possible in the same data environment; hence, reducing the amount of data that needs to be transferred between host and device. There are two main steps involved in this process: (i) finding which loops should be surrounded by the same data environment (Section 4.3.1), and (ii) inserting the actual copy block in the annotated source file (Section 4.3.2).

4.3.1 Deciding which Loops can be Merged into Common Transfer Blocks

To decide which loops should have their transfer operations coalesced, we first divide the program’s control flow graph into SESE regions, as explained in Section 3.3.3. Example 8 describes an instance of this division. To identify which of these regions we can analyse, we resort to the symbolic range analysis seen in Chapter 2, plus classic data dependence analysis [Ferrante et al., 1987]. These two techniques, once combined, let us find the regions that we can optimize. We can analyze a region if we are able to define the access bounds of all arrays used within its scope. An array has bounded accesses if we can determine symbolic ranges for all the expressions used to index it within the region of interest. A region of interest contains only parallel loops, which we can run on the target accelerator. We want to enclose the largest of such regions on a data environment. Example 9 provides some insight on this observation.

Example 8 *Figure 4.7 (d) shows the structure of the CFG for the program in Figure 4.7 (a), with SESE regions highlighted (regions represented by a single basic block or that are not meaningful to the example are not represented in the Figure).*

Example 9 *Going back to our running example of Figure 4.7, every array access has limits known right at the start of region R1, the largest region in the function (Figure 4.7 (b)). DawnCC places data transfer operations around this region; hence, arriving at the code seen in Figure 4.7 (c), which is semantically equivalent to the original program seen in Figure 4.7 (a).*

4.3.2 Carrying on with Coalescing into Source Code

As our intent is to insert annotations in the original source code, the intermediate representation over which we run our static analyses should be as close to the input program’s text as possible. To this extent, when generating a program’s CFG we disable any optimization that can move code across different points of said CFG. One particular optimization, however, cannot be disabled in most cases: the conversion to Static Single Assignment Form (SSA) [Cytron et al., 1991]. A number of mainstream compilers, such as Clang, GCC, ICC, HotSpot, and Mozilla’s IonMonkey, will convert a program to SSA form when generating its intermediate representation. The virtual names inserted in this process make it harder to map instructions in a program’s CFG to its original text – a fact that Example 10, adapted from PolyBenchGPU’s `fdtd_2d`, clarifies.

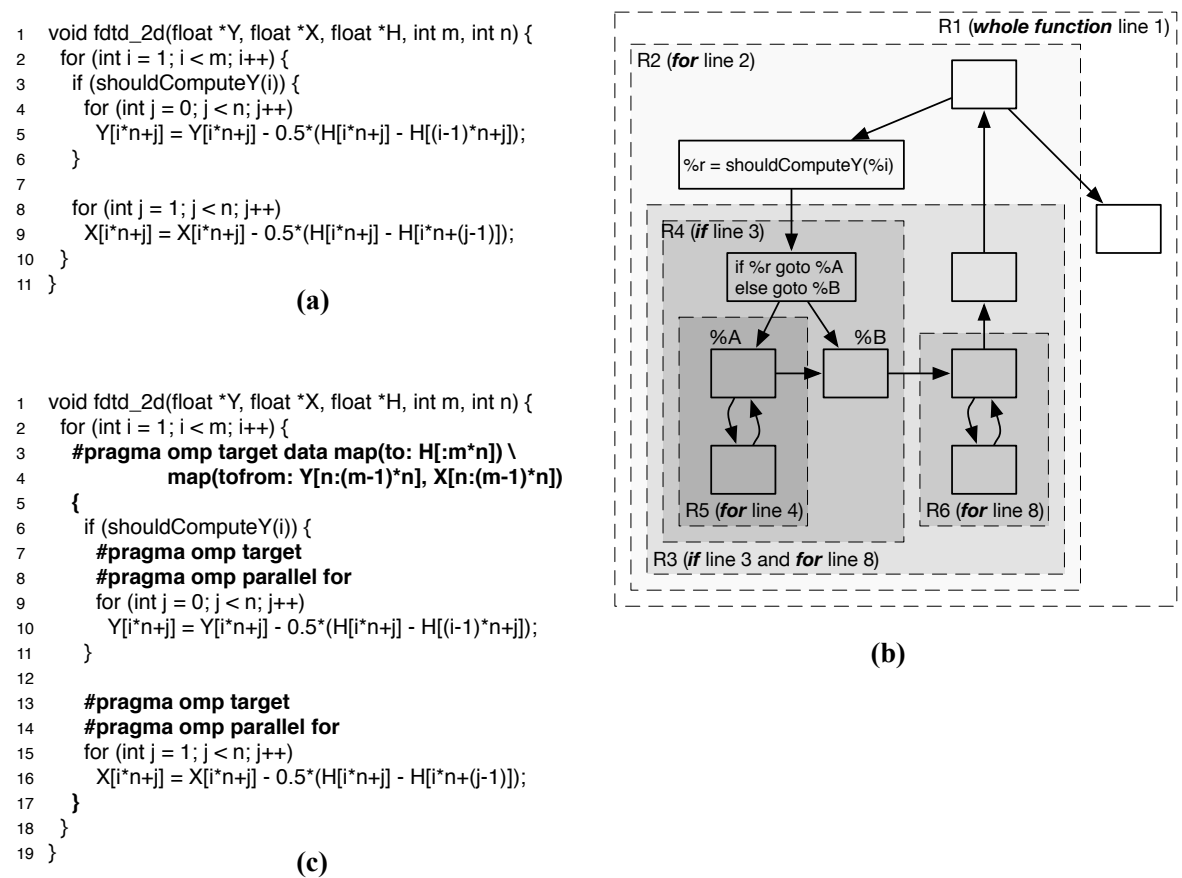


Figure 4.8: (a) Example program adapted from the *fdtd_2d* benchmark, (b) SESE regions in the program. (c) Unsafe annotations.

Example 10 Figure 4.8 (b) shows the SESE regions of function *fdtd_2d*, seen in Figure 4.8 (a). The call to *shouldComputeY* in line 3 can yield side-effects. Thus, dependence analysis tells us that the outer loop might not be parallel. The two inner loops in lines 4-5 and 8-9, however, are stencil kernels that can be annotated to run in the accelerator. In the CFG, the largest region for which we have full symbolic range information (R3) goes from right before the if statement in line 3 to right after the for loop in lines 8-9. However, the call to *shouldComputeY* is syntactically inside our target region in the program’s text (lines 3 through 9), but falls outside of it in the program’s CFG, due to the virtual name *%r* inserted during the SSA transformation.

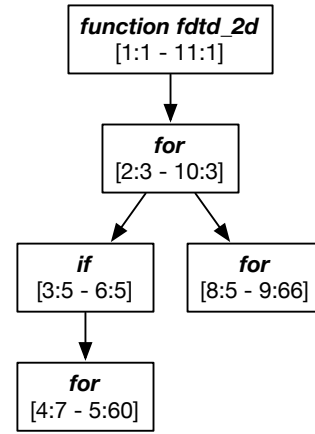
Example 10 shows that a SESE region in a program’s CFG in SSA form may not directly map to a SESE region in the original program text. In Example 10, surrounding region *R3* with a data transfer block in the source program (as seen in Figure 4.8 (c)) could change its semantics. This modification happens if the call to *shouldComputeY* modifies the values stored in arrays *H*, *Y* or *X* after they have been

```

1 void fdttd_2d(float *Y, float *X, float *H, int m, int n) {
2   for (int i = 1; i < m; i++) {
3     if (shouldComputeY(i)) {
4       #pragma omp target map(to: H[(i-1)*n:2*n]) \
5         map(tofrom: Y[i*n:n])
6       #pragma omp parallel for
7       for (int j = 0; j < n; j++)
8         Y[i*n+j] = Y[i*n+j] - 0.5*(H[i*n+j] - H[(i-1)*n+j]);
9     }
10
11    #pragma omp target map(to: H[i*n:n]) \
12      map(tofrom: X[i*n:n])
13    #pragma omp parallel for
14    for (int j = 1; j < n; j++)
15      X[i*n+j] = X[i*n+j] - 0.5*(H[i*n+j] - H[i*n+(j-1)]);
16  }
17 }

```

(a)



(b)

Figure 4.9: (a) Annotations produced by DawnCC for program in Figure 4.8 (a), (b) scope tree.

sent to the GPU.

To ensure that our coalescing strategy is correct under SSA conversion, we resort to a simple restriction: the data environment created to cover an analyzable SESE region goes from the point right before its first parallel loop (including its preheader) to the point right after its last parallel loop. SSA conversion does not change the structure of loops nor moves computation across them. Thus, all computation inside the data environment defined by the above restriction will be the same in both a program’s text and in its CFG. Example 11 provides some intuition on why such approach is safe.

Example 11 *If we were to surround region R3, in Figure 4.8 (b), with a data transfer block, then this region should go from the loop in line 4 to the end of the loop in lines 8-9 in Figure 4.8 (a). Thus, this data transfer block does not include the call `shouldComputeY`. This exclusion holds in both the source text (Figure 4.8 (a)) and the CFG (Figure 4.8 (b)).*

Despite being safer, our augmented coalescing approach still suffers from one problem: it is not always syntactically valid to surround different loops within the same region with a data-transfer block. Inserting a block around both loops in Example 11 is not syntactically valid. The beginning of the block would fall inside the `if` in line 3 of Figure 4.8 (a) and its end would lay outside. To avoid such problems, we use a last verification step. This check uses a data structure that we call a *scope tree*.

Ensuring Safety with Scope Trees A scope tree is a tree-like data structure in which nodes represent statements in the source program that can create SESE regions (statements such as *for*, *if*, and *switch cases*). Each node contains its text range: the coordinates in the source code (line and column of the character in the text) that represent its first and last character. For instance, the *if* statement in Figure 4.8 (a) ranges from line 3 column 5 to line 6 column 5. This data structure can be constructed using a C parser. The main property of a scope tree, which makes it useful to our purposes, is that the range of any node falls either completely inside or completely outside another node’s range, i.e., it has the *property of balanced parentheses*: if node c_1 is a child of node c_0 , then the lines that c_1 cover lay within the lines that c_0 represents. Example 12 illustrates this property.

Example 12 *Figure 4.9 (b) shows the scope tree for the function `fdtd_2d`, seen in Figure 4.9 (a). This tree has the property of balanced parentheses. For instance, the only node `if` spans lines 3 to 6. Its child, a node `for`, spans lines 4 to 5.*

We say that a data environment block is safe to be inserted when it preserves the property of balanced parentheses. This additional verification tells us that enclosing the loops in region *R3* of Figure 4.8 (b) is not syntactically safe, as we explained above. The largest regions of our example that follow all desired properties and meet all safety restrictions are now *R5* and *R6* of Figure 4.8 (b). Therefore, in this example, each parallel loop should be annotated with separate data transfer operations. DawnCC produces, in this case, the code seen in Figure 4.9 (a). Notice that the existence of a well-defined scope tree is not essential for DawnCC: if this data structure is not available or cannot be generated, the tool will use uncoalesced memory transfers to parallelize code.

4.4 Experiments

We have evaluated our techniques using two different compilers and architectures. In what follows we describe our methodology and discuss our results.

Benchmarks. We tested our analyses on the version of PolyBenchGPU used by Grauer-Gray et al. [2012]. Each benchmark in PolyBenchGPU comes with five different sizes of input; we present dynamic numbers for the three largest ones, namely *medium*, *large* and *huge* (or *extra large*).

Hardware We experimented with the following setups:

- **Desktop:** Intel Xeon CPU E5-2620, with 6 cores of 2.00GHz and 16 GB of RAM (DDR2), running Linux Ubuntu 12.04 3.2.0, equipped with a GPU model GeForce GTX 670, with 2 GB of RAM (CUDA Compute Capability 3.0).
- **Phone:** Exynos7420 AArch64 Processor with 4GB of RAM running Android 5.1.1 and equipped with a GPU model ARM Mali-T760 with 913 MB of RAM and 8 parallel compute units.

Compilers We have used these compilers to translate annotated code to binaries:

- **gpuclang:** gpuclang version 2.0 (based on Clang 3.5.0). Runs on the phone setup, and translates OpenMP to parallel code.
- **pgcc:** PGI C Compiler version 16.1 64bit. Runs on the desktop setup, and translates OpenACC to parallel code.

4.4.1 Runtime Results

Desktop setup (DawnCC+pgcc [OpenACC]). Figure 4.10 shows the relative runtime of annotated code compared against the same code, without our annotations. Both binaries are produced by pgcc -O3. We run each program five times; bars show averages. When probing the GPU’s execution time, we include the time to transfer data to and from the GPU. Variance is negligible; hence, we will not provide error intervals. We observe very large speedups in four benchmarks: 2MM, 3MM, COVAR and GEMM. These are embarrassingly parallel applications, which benefit substantially from the SIMD execution model of a GPU. We have also observed slowdowns in six benchmarks. These slowdowns happen in benchmarks that run for very short times. To emphasize this point, we show absolute runtimes for largest inputs next to each benchmark in Figure 4.10.

Phone setup (DawnCC+gpuclang [OpenMP]). Figure 4.11 shows the speedup that we obtain when comparing the annotated code, compiled with gpuclang, against Clang v3.5 -O3, on the phone setup. Each program has been executed five times, and bars show averages. In this setup we observe speedups in more benchmarks, although we have not gotten results as dramatic as those seen in the desktop setup. Again, GEMM is the benchmark where we got the more noticeable gains. The less impressive speedups are due to the fact that the difference, in terms of number of available cores, between the Mali GPU and the Exynos CPU is smaller than the difference between the GTX GPU and the Xeon CPU.

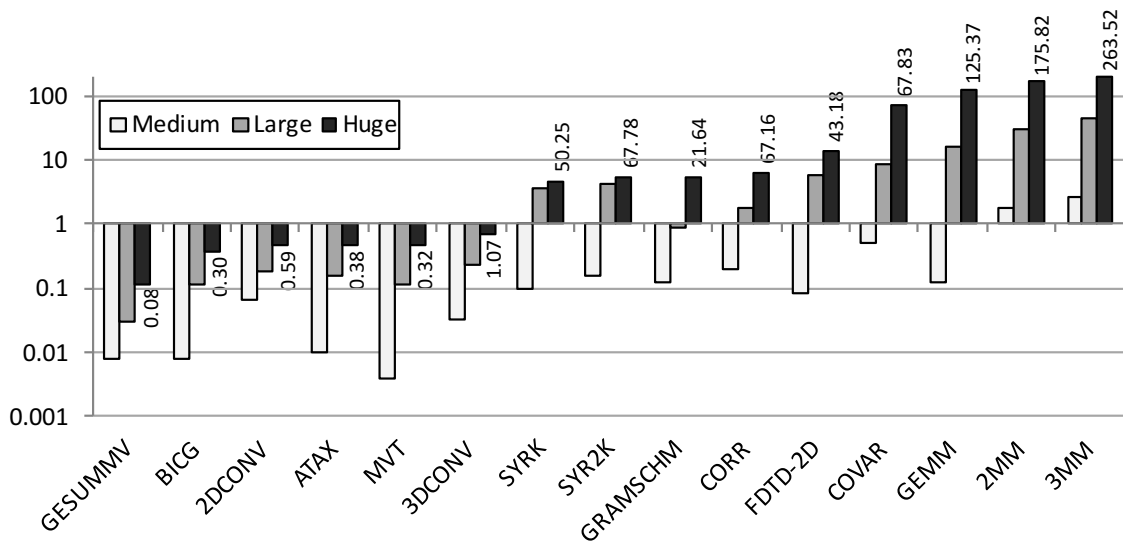


Figure 4.10: **Desktop (DawnCC+pgcc vs pgcc)** Speedup due to the annotations inserted by DawnCC, compared to execution of sequential code on the Desktop setup. Both programs, original and annotated, have been compiled with pgcc. Y-axis show speedup, in number of times. The higher the bar, the better. Numbers represent absolute runtime, in seconds, of benchmarks with largest inputs, running on the CPU.

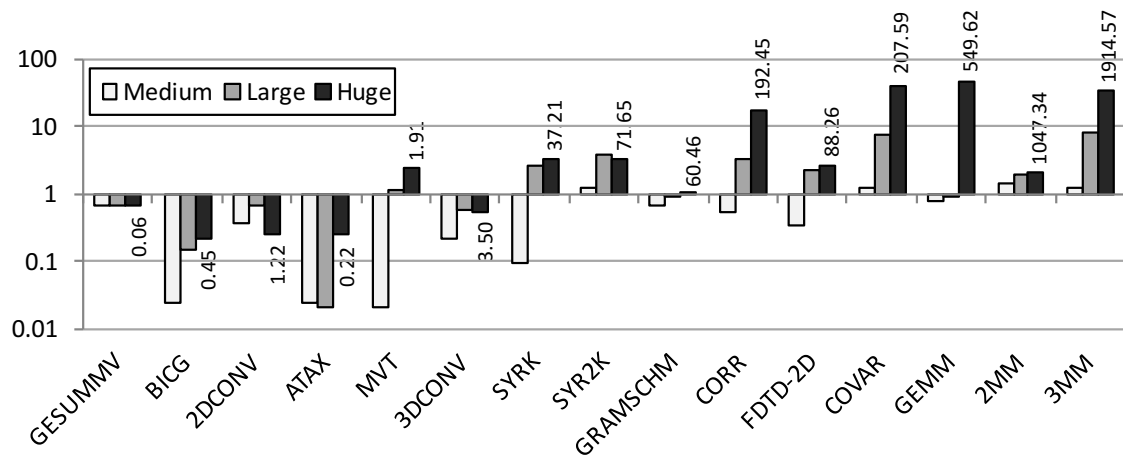


Figure 4.11: **Phone (DawnCC+gpuclang vs gpuclang)** Comparison between the code that DawnCC has annotated with OpenMP pragmas and compiled with gpuclang, and the benchmarks without the annotations. Y-axis show speedup, in number of times. The higher the bar, the better. Numbers represent absolute runtime, in seconds, of benchmarks with largest inputs, running on the CPU.

4.4.2 The Impact of Copy Coalescing

Figure 4.12 shows results produced by the copy coalescing optimization described in Section 4.3. The reduction in the number of pragmas indicate how many regions in the

Benchmark	# of pragmas		# of copies		Runtime (sec)		
	Orig.	Opt.	Orig.	Opt.	Orig.	Opt.	Speedup
GESUMMV	1	1	7	7	•	•	•
BICG	3	1	10	7	0.80	0.70	14%
2DCONV	1	1	3	3	•	•	•
ATAX	3	1	10	6	0.81	0.70	16%
MVT	2	1	8	7	0.66	0.63	5%
3DCONV	1	1	3	3	•	•	•
SYRK	2	1	5	3	10.89	10.87	0.2%
SYR2K	1	1	4	4	•	•	•
GRAMSCHM	1	1	6	6	•	•	•
CORR	4	4	14	14	•	•	•
FDTD-2D	1	1	7	7	•	•	•
COVAR	3	1	9	6	0.92	0.90	2%
GEMM	1	1	4	4	•	•	•
2MM	2	1	8	7	1.08	1.06	2%
3MM	3	1	12	10	1.33	1.32	1%
Total	29	18	110	94			

Figure 4.12: Results produced by the copy coalescing optimization of Section 4.3 in the Desktop setup. “Orig.” denotes the original program, and “Opt.” its optimized version. Runtime is the GPU’s, for the largest input size. The bullets in the last two columns indicate benchmarks where no coalescing could be performed.

code had their transfer operations coalesced. The number of copies account for how many arrays are transferred to the accelerator in each version. DawnCC has been able to eliminate redundant copies in seven out of the 15 benchmarks available. Manual inspection of the untouched benchmarks reveal the absence of further opportunities for copy coalescing. In two benchmarks, ATAX and COVAR, DawnCC could produce a transfer block that surrounds the entire program kernel. In the latter case, three loop nests, containing seven loops, have been placed within a single transfer block. The final result of this optimization is performance, as the two right columns in Figure 4.12 show.

Figure 4.13 (a) shows the percentage of speedup that we obtain with copy coalescing. These numbers are modest for most of the benchmarks, but they refer to only the time to copy data between host and device. The higher the asymptotic complexity of the kernel, the lower will be the gains produced by copy coalescing, because the copy cost is amortized on the program’s runtime cost. Nevertheless, if we consider only the time to move data, then the results produced by copy coalescing are more noticeable, as Figure 4.13 (b) shows. This figure outlines only the percentage of time saved to copy

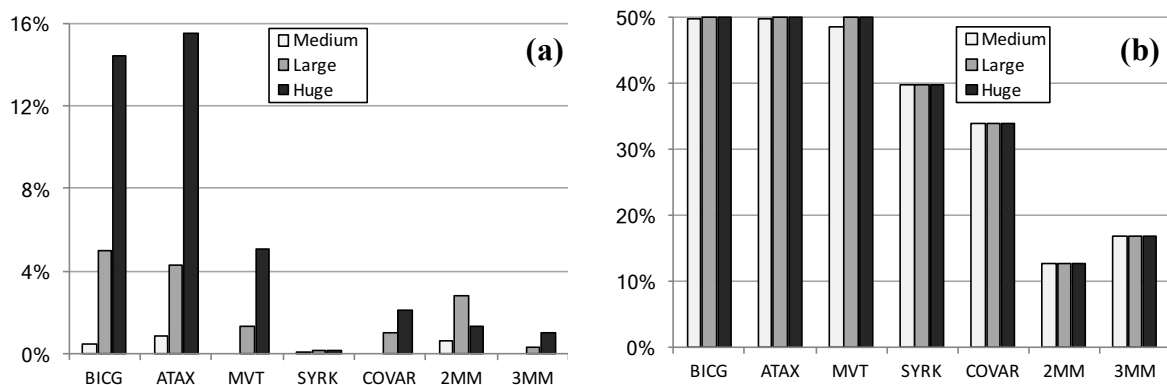


Figure 4.13: (a) Performance improvement due to the copy coalescing optimization discussed in Section 4.3. Bars show percentage of speedup of optimized over non-optimized code. The higher the bar, the better. (b) Percentage of copy time saved. X% is the reduction in the time spent copying data. The higher the bar, the better.

data. In a few benchmarks, e.g., *Atax*, *Bicg* and *Mvt*, copy coalescing could reduce in almost 50% the time spent in memory transfers.

4.4.3 DawnCC vs manual code annotation

Breno Campos, from the UFMG Compilers Lab., produced a manually annotated version of each benchmark. We compared the code produced by our tool against this version and against UniBench, a publicly available version of PolyBenchGPU annotated with OpenMP 4.0 pragmas. The only difference between the manual annotations and the code transformed automatically refers to annotating the loops that initialize the data structures used in each benchmark. Performing this initialization on the accelerator is not worthwhile for these benchmarks, on account of the time to transfer data. Therefore, the expert developer chose to leave these routines untouched, whereas DawnCC has annotated all of them. However, these procedures run for a very short time, compared with the execution time of each kernel. In the end, we have not been able to measure significant differences between the runtime of manually and automatically annotated programs. DawnCC has even correctly annotated each data that is only read on the GPU as read-only, which avoids transferring them back to the CPU once processed on the accelerator. Furthermore, DawnCC has been able to produce pointer disambiguation tests, a task that the human developer has not performed. Such omission is due to the fact that dealing with memory indexing expressions is a tedious and error prone task.

Limitations of our method can be more noticeable in the presence of less regular

input programs. The symbolic range analysis used here is only capable of deriving limits for array operations, which means that a human user would perform better at annotating code that relies heavily in custom or more dynamic data structures. Additionally, our analysis cannot estimate bounds in the presence of function calls, which could be overcome with techniques such as interprocedural bounded regular section analysis [Havlak and Kennedy, 1991]. DawnCC is also limited to *doall* parallelism, whereas an experienced developer could identify and take advantage of more refined work division patterns, such as reduction operations. We emphasize, however, that the majority of the programs that can take advantage of an external acceleration device are regular. As an example, PolyBenchGPU encompasses a set of core algorithms widely targeted by hardware acceleration techniques and DawnCC is able to annotate all kernels in it.

Chapter 5

Related Work

The symbolic access range analysis that we present in Chapter 2 is based on the work of Rugina and Rinard [2000] on bound estimation for memory regions. In their paper, Rugina and Rinard present a static region analysis capable of inferring symbolic bounds for array accesses and investigate its use on the elimination of memory safety checks. Contrary to our method, their technique restricts the set of symbolic bounds that can be generated to linear expressions, whereas our analysis allows for different arithmetic and type operators. Rugina and Rinard’s approach also uses integer linear programming to derive symbolic bounds, which can be too expensive for many applications and larger program sizes.

Nazaré et al. [2014] largely extended and investigated the use of symbolic range analysis in different contexts. In their work, they provide a more formal description to the problem of inferring symbolic ranges and are able to handle significantly more complex expressions. Our inference analysis represents a simpler version of Nazaré et al.’s approach, while still keeping acceptable precision, as we show experimentally. A key difference of our work is that we propagate range information backwards in the program graph, as done by Rugina and Rinard, whereas Nazaré et al.’s technique focuses on propagating information from allocation points to array accesses, in a forward fashion. Additionally, Nazaré et al. proposes a more refined renaming strategy to achieve a sparser analysis, while we rely solely on SSA form, restricting our range inference to interval loops to achieve a similar result. Campos et al. [2016] provide a thorough investigation on how this difference affects the impact of the analysis in other code optimization passes. When considering analysis clients, both Nazaré et al. and Rugina and Rinard focus on static applications, like the elimination of array bound checks. Here, however, we investigate in detail dynamic uses of symbolic range inference, with the generation of bound expressions intended to be evaluated at runtime. As the nov-

elty of our work lies on these applications, i.e., memory disambiguation and automatic offloading, we provide an overview of previous work in these areas in the sections that follow.

5.1 Literature on Memory Disambiguation

Profiling Based Alias Analysis. One of the inspirations for our work on pointer disambiguation is the study performed by Mock et al. [2001], in which they have shown that the pointer information produced by state-of-the-art static alias analyses is markedly worse than the actual behavior observed at runtime. This work has also inspired other groups [Guo, 2006; Da Silva and Steffan, 2006], which, differently from us, use profiling to determine the probability that aliasing happens in practice. In this case, a recovery mechanism is necessary to preserve the semantics of the program in face of actual aliasing. Lin et al. [2003] provide some examples of different ways to recover from wrong speculation.

This kind of speculative *modus operandi* has seen use in several other works [Ceze et al., 2006; Fernández and Espasa, 2002; Huang et al., 1994; Da Silva and Steffan, 2006], which have in common the fact that a profiler is used to derive alias information. This information is then made available to the program’s runtime environment. For instance, Huang et al. [1994] have applied speculative pointer disambiguation to solve dependencies in the context of a very long instruction word (VLIW) machine. They assume that there is no dependence between some memory references that overlap with low probability. If this dependence is observed at runtime, then they rollback the execution. They also mention that an alternative would be to branch to a different version of the code, as we do. In Huang’s case, dependencies are resolved in hardware. Similar approaches, implemented at the software level, have been proposed by Fernández and Espasa [2002] and Da Silva and Steffan [2006]. Chen et al. [2004] have designed several enhancements on the basic speculative methodology to make it faster and more accurate.

All these previous efforts apply optimizations assuming that pointers do not alias at runtime: profiling enables code optimizations whenever it indicates that aliasing happens with low probability. However, our approach is not speculative. Instead of speculating, we replicate code, and use the runtime checks to decide where to branch. Furthermore, contrary to us, none of these previous works tries to generate disambiguation checks statically.

Hybrid Pointer Disambiguation for Parallelism. Rus et al. [2003] generate stati-

cally checks that disambiguate pointers at runtime. This technique to disambiguate pointers differs from our work because the test that Rus *et al.* generate is a necessary condition to enable loop parallelization. This constraint may lead to complex dynamic checks that need to be simplified. Oancea and Rauchwerger [2012] have proposed several ways to perform this simplification within a framework of logical inference rules. They use, for instance, variable elimination in a Fourier-Motzkin style. Posterior work [Oancea and Rauchwerger, 2015] has adopted similar ideas to disambiguate induction variables that can be converted to a closed form. The method that we advocate in our work has the primary goal of overcoming the limitations of the static analyses commonly applied on low-level intermediate program representations. We use a lightweight approach that allows us to: (i) take advantage of existing compiler infrastructure (e.g. relational analysis); (ii) let the compiler make the decision of which optimization to perform (e.g. among all possible polyhedral transformations); (iii) generate simpler run-time checks; and (iv) improve the practicality of the static alias analyses approaches which mainstream compilers use.

Memory Aliasing in Research Compilers. Research compilers such as Pluto [Bondhugula et al., 2008] or PPCG [Verdoolaege et al., 2013b] ignore memory aliasing when applying state-of-the-art loop optimizations for cache locality, parallelism, vectorization or accelerator usage. As a result, their transformations may change the program behavior in case aliasing actually happens and an external alias analysis is not used. Thus, in practice they may cause general code transformation to be unsound for existing C/C++ codes. Outside of a user controlled benchmark environment, pointer disambiguation techniques such as the ones presented in this work are required to maintain program correctness and enable the use of advanced loop optimizations.

Memory Aliasing in Static Compilers. Static compilers such as `icc`, IBM XL, `gcc` or Clang all support some kind of loop versioning to enable optimizations such as vectorization in the presence of pointer aliasing. For closed source compilers it is difficult to understand how general their code versioning support indeed is, but the experiment in Figure 3.1 suggest that at least `icc`, even for simple examples, does not always apply loop versioning. Looking at the source code of `gcc` and Clang, we can confirm that even their latest development versions¹ only allow versioning of simple, innermost loops, as it is required for loop vectorization and other transformations that focus on innermost loops. Pointer disambiguation for complex loop nest or fully

¹The upcoming `gcc` 5.3 and Clang 3.7 releases

dynamic pointer disambiguation approaches, as we presented them, have not been used by the static compilers that we analyzed.

5.2 Literature on Automatic Program Annotation and Offloading

Annotation Systems. Annotation systems, such as OpenMP 4.0 [Jaeger et al., 2015], OpenSs [Meenderinck and Juurlink, 2011] and OpenACC [Wienke et al., 2012] are a simple, yet powerful alternative to the development of high-performance software. Such systems are not a programming language *per se*; rather, they work as a *meta-language*, which, once combined with a host-language, typically Fortran, or C, let developers imbue standard syntax with parallel semantics. The emergence of such systems, has led to a resurgence of interest in parallelizing compilers. OpenAcc, for instance, has been a target of several different compilers, such as AccUll [Reyes et al., 2012], Omni OpenAcc [Tabuchi et al., 2016], ipmacc [Lashgar et al., 2014], OpenARC [Kim et al., 2015; Lee and Vetter, 2014] and pgcc [Ghike et al., 2014]. Similarly, OpenMP 4.0 is already supported by several mainstream compilers, including gcc 4.9.0 (for C/C++), gcc 4.9.1 (for Fortran), icc 15.0 (C/C++/Fortran) and LLVM’s Clang 3.7, which offers partial support to OpenMP 4.0 for C/C++. Our tool, DawnCC, is not an OpenACC or OpenMP 4.0 compiler; hence, it does not compete against the technologies that we have just mentioned. Instead, DawnCC works one level up: inserting the annotations that will be later translated by an OpenAcc or OpenMP 4.0 compliant compiler.

Automatic Parallelization. There exists an enormous corpus about the automatic parallelization of software. For an overview about the classic techniques, we recommend the book of Michael Wolfe [Wolfe, 1995]. However, the goal of our work is not to parallelize programs; instead, we want to give programmers the tools to benefit from latent parallelism already available in code. Thus, our interval analysis is needed only when we use pragmas to offload code to an external device. We do not need data copy directives when using OpenMP to parallelize for simple multithreading, for instance. The research community has not yet tackled the problem of inserting copy directives in programs, even though we have today a rich literature about automatic generation of OpenMP code, which is represented by work such as Lee et al. [2009].

Automatic Offloading. A number of optimization frameworks based on the polyhedral model have been used for automatic generation of OpenMP and GPU code [Ver-

doolaeghe et al., 2013a; Baskaran et al., 2010]. These tools generate data copy library calls by inspecting the iteration domains of arrays used within parallel loops. In practice, the symbolic limits generated by such frameworks and the ones generated by the analysis chosen for this work present similar results, as we show on Chapter 3, each having specific advantage scenarios. For instance, the method applied here can handle non-affine regions of code, while the analyses implemented in polyhedral-based tools usually generate simpler interval expressions, by performing static simplification, which comes at the cost of a higher compilation time.

Chapter 6

Conclusion

This dissertation summarized two years of research work dedicated to extending the reach of compiler optimizations through the use of better memory-related information. We derived a precise, yet simple analyses capable of inferring symbolic ranges for memory operations, which we described in Chapter 2. The result of this analysis, bound expressions that can be solved at runtime, allowed us to design new code optimizations, as well as improve existing ones. We tackled the long-standing problem of memory disambiguation for static compilers, by producing a hybrid pointer disambiguation technique whose performance compares to or overcomes that of other state-of-the-art analyses. This method allows LLVM to produce faster binaries in many different cases. We then combined both our range analysis and disambiguation technique to produce **DawnCC**, a framework that annotates code for GPU execution. During the implementation of this framework, we put time and effort in the solution of other hard problems, such as (i) replicating in high-level C code the result of analysis design to operate over intermediate representation and (ii) generating human-readable symbolic expressions. The annotations produced by **DawnCC** lead to speedups in different scenarios. We believe that these contributions, published in respected Computer Science venues, represent a meaningful advancement for the code analysis and optimization field. The source code of all techniques presented here is available for download, as described in Section 6.2.

6.1 Contributions

Due to the myriad of subjects it touches in the code optimization field, this work has a series of key contributions:

- We present a backwards hybrid symbolic analysis of memory access ranges and evaluate its precision in a widely used benchmark suite.

- We propose a lightweight pointer disambiguation technique and demonstrate that it matches or surpasses the precision of more expensive methods, i.e, purely dynamic and polyhedron-based, implemented by two other research groups.
- We describe the implementation of DawnCC, a framework to insert code offloading annotations.
- We present a strategy to reduce the amount of data transferred between host and device in programs annotated for GPU execution.

6.2 Publications and Software

The contributions presented in this master’s dissertation were the result of two fruitful years of research work, published in a number of respected Computer Science venues:

- **A case for a fast trip count predictor [Alves et al., 2014] - IPL’14:** describes a precursor of our symbolic analysis, capable of inferring ranges for induction variables of simple interval loops. This technique was mainly directed to innermost loops.
- **Runtime Pointer Disambiguation [Alves et al., 2015] - OOPSLA’15 (*artifact evaluated*):** description of our symbolic range analysis, hybrid pointer disambiguation technique, and comparison against two other disambiguation methods: the first, purely dynamic, was developed by Fabian Gruber, from INRIA-France. The second, was developed by Johannes Doerfert and Tobias Grosser, the main maintainers of Polly.
- **Restritificação [Campos et al., 2015b] - SBLP’15 (*best paper*):** though not being its main focus, this paper includes a comparison of our technique to a static forward disambiguation method and includes new experiments regarding the impact of our analysis in loop transformations.
- **Restrictifier: a tool to disambiguate pointers at function call sites [Campos et al., 2015a] - CBSOFT Tools’15 (*best paper*):** includes a description of the implementation of our analysis in the LLVM compilation infrastructure and how client code transformations can use it to improve their precision.
- **Restrictification of Function Arguments [Campos et al., 2016] - CC’16:** presents a variation of our range analysis and disambiguation method, capable of operating over whole functions rather than isolated regions. This improved version also presents a *best-effort* mode, where symbolic ranges can be computed even in the presence of procedure calls and other sources of imprecision. An

additional comparison with methods developed by Henrique Nazaré and Victor Campos from UFMG is also presented.

- **Paralelização Automática de código com diretivas OpenACC [Moreira et al., 2016] - SBLP'16:** prototype version of DawnCC, capable of generating simple OpenACC directives.
- **Automatic Insertion of Copy Annotation in Data-Parallel Programs [Mendonça et al., 2016] - SBAC-PAD'16:** full description of the analyses and architecture of DawnCC, including a set of optimizations to reduce data transfer overhead and the ability to generate both OpenACC and OpenMP pragmas.
- **DawnCC: Automatic Annotation for Data Parallelism and Offloading - Submitted to ACM TACO:** extended version of the SBAC-PAD paper for a journal. Includes new optimization strategies to reduce the amount of data transferred between host and device, partly implemented by Breno Campos from UFMG, and a set of static simplifications capable of generating human-readable symbolic expressions. The latter being developed by Gleison Mendonça, also from UFMG.

Thanks to the diligent work of Breno Campos from the UFMG Compilers Lab., the implementation of the techniques described in this work are available for public use in the following addresses:

- <http://github.com/periclesroalves/runtime-pointer-disambiguation>: implementation of our symbolic access range analysis on top of LLVM as well as the code for our hybrid pointer disambiguation technique.
- <http://github.com/gleisonsdm/DawnCC-Compiler>: source code of the DawnCC framework, including all of its composing analyses.
- <http://cuda.dcc.ufmg.br/dawn/>: online interface for the DawnCC compiler. At this address the reader can upload its own source code and automatically annotate it for GPU execution, using either OpenMP 4.0 or OpenMP.

6.3 Future Work

There are many small improvements that can be made to different parts of our work. For our symbolic access range analysis, the use of interprocedural techniques such as bounded regular section analysis [Havlak and Kennedy, 1991] could improve the precision in the presence of procedure calls. In our hybrid pointer disambiguation, better

static simplification strategies could be implemented to reduce the runtime overhead of the dynamic checks that it generates. Finally, **DawnCC** currently recognizes only *doall* parallelism. The analyses on top of which the framework is built could be extended to take advantage of other parallel patterns, such as *reduction*, for which both OpenMP 4.0 and OpenACC offer special directives.

Bibliography

- Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley.
- Akritidis, P., Costa, M., Castro, M., and Hand, S. (2009). Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *SSYM*, pages 51--66. USENIX.
- Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., and Pereira, F. M. Q. a. (2015). Runtime pointer disambiguation. In *OOPSLA*, pages 589--606. ACM.
- Alves, P., Rodrigues, R., Sousa, R., and Pereira, F. M. Q. (2014). A case for a fast trip count predictor. *IPL*, 10(1016):8.
- Andersen, L. O. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen.
- Appel, A. W. and Palsberg, J. (2002). *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition.
- Baskaran, M. M., Ramanujam, J., and Sadayappan, P. (2010). Automatic C-to-CUDA code generation for affine programs. In *CC*, pages 244--263. Springer.
- Bayer, R. (1972). Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290--306.
- Bentley, J. L. and Friedman, J. H. (1979). Data structures for range searching. *ACM Comput. Surv.*, 11(4):397--409.
- Bertolli, C., Antao, S. F., Eichenberger, A. E., O'Brien, K., Sura, Z., Jacob, A. C., Chen, T., and Sallénave, O. (2014). Coordinating GPU threads for OpenMP 4.0 in LLVM. In *LLVM-HPC*, pages 12--21. IEEE.

- Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices*, 43(6):101--113.
- Campos, V., Alves, P., and Pereira, F. (2015a). Restrictifier: a tool to disambiguate pointers at function call sites. In *CBSoft Tools*. SBC.
- Campos, V., Alves, P., and Pereira, F. (2015b). Restritificação. In *SBLP*. SBC.
- Campos, V. H. S., Alves, P. R. O., Santos, H. N., and Pereira, F. M. Q. (2016). Restrictification of function arguments. In *CC*, pages 163--173. ACM.
- Ceze, L., Tuck, J., Torrellas, J., and Cascaval, C. (2006). Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, pages 227--238. IEEE.
- Chabbi, M. and Mellor-Crummey, J. (2012). DeadSpy: a tool to pinpoint program inefficiencies. In *CGO*, pages 124--134. ACM.
- Chen, T., Lin, J., Dai, X., Hsu, W.-C., and Yew, P.-C. (2004). Data dependence profiling for speculative optimizations. In *Compiler Construction*, pages 57--72. Springer.
- Cormen, T. H. (2009). *Introduction to algorithms*, chapter III.14, pages 348--355. MIT press.
- Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, K. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451--490.
- Da Silva, J. and Steffan, J. G. (2006). A probabilistic pointer analysis for speculative optimizations. In *ASPLOS*, pages 416--425. ACM.
- Duck, G. J. and Yap, R. H. C. (2016). Heap bounds protection with low fat pointers. In *CC*, pages 132--142. ACM.
- Fernández, M. and Espasa, R. (2002). Speculative alias analysis for executable code. In *PACT*, pages 222--231. IEEE.
- Ferrante, J., Ottenstein, J., and Warren, D. (1987). The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319--349.
- Ghike, S., Gran, R., Garzarán, M. J., and Padua, D. A. (2014). Directive-based compilers for GPUs. In *LCPC*, pages 19--35.

- Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., and Cavazos, J. (2012). Auto-tuning a high-level language targeted to GPU codes. In *InPar*, pages 1–10. IEEE.
- Gregg, C. and Hazelwood, K. (2011). Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*, pages 134–144. IEEE.
- Grosser, T., Groeslinger, A., and Lengauer, C. (2012). Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 04(22):X.
- Grosser, T., Verdoolaege, S., and Cohen, A. (2015). Polyhedral AST generation is more than scanning polyhedra. *TOPLAS*, 37(4):12:1–12:50.
- Guo, P. J. (2006). *A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ programs*. PhD thesis, MIT.
- Hall, M. W. (1991). *Managing interprocedural optimization*. PhD thesis, Rice University.
- Havlak, P. and Kennedy, K. (1991). An implementation of interprocedural bounded regular section analysis. *Trans. Parallel Distrib. Syst.*, 2(3):350–360.
- Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.
- Hind, M. (2001). Pointer analysis: Haven’t we solved this problem yet? In *PASTE*, pages 54–61. ACM.
- Horwitz, S. (1997). Precise flow-insensitive may-alias analysis is NP-hard. *TOPLAS*, 19(1):1–6.
- Huang, A. S., Slavenburg, G., and Shen, J. P. (1994). Speculative disambiguation: a compilation technique for dynamic memory disambiguation. In *ISCA*, pages 200–210.
- Jaeger, J., Carribault, P., and Pérache, M. (2015). Fine-grain data management directory for OpenMP 4.0 and OpenACC. *Concurr. Comput. : Pract. Exper.*, 27(6):1528–1539.
- Karrenberg, R. and Hack, S. (2011). Whole-function vectorization. In *CGO*, pages 141–150. IEEE.

- Kennedy, R., Chan, S., Liu, S.-M., Lo, R., Tu, P., and Chow, F. C. (1999). Partial redundancy elimination in SSA form. *TOPLAS*, 21(3):627–676.
- Kim, J., Lee, S., and Vetter, J. S. (2015). An OpenACC-based unified programming model for multi-accelerator systems. In *PPoPP*, pages 257–258. ACM.
- Kulkarni, P. A., Whalley, D. B., Tyson, G. S., and Davidson, J. W. (2006). Exhaustive optimization phase order space exploration. In *CGO*, pages 306–318. IEEE.
- Landi, W. and Ryder, B. G. (1991). Pointer-induced aliasing: A problem classification. In *POPL*, pages 93–103. ACM.
- Lashgar, A., Majidi, A., and Baniasadi, A. (2014). IPMACC: open source OpenACC to CUDA/OpenCL translator. *CoRR*, abs/1412.1127:1–14.
- Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.
- Lee, S., Min, S.-J., and Eigenmann, R. (2009). OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *PPoPP*, pages 101–110. ACM.
- Lee, S. and Vetter, J. S. (2014). OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing. In *HPDC*, pages 115–120. ACM.
- Lin, J., Chen, T., Hsu, W.-C., Yew, P.-C., Ju, R. D.-C., Ngai, T.-F., and Chan, S. (2003). A compiler framework for speculative analysis and optimizations. In *PLDI*, pages 289–299. ACM.
- Liu, Y., Yang, C., Liu, F., Zhang, X., Lu, Y., Du, Y., Yang, C., Xie, M., and Liao, X. (2016). 623 tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores. *IJHPCA*, 30:39–54.
- Margiolas, C. and O’Boyle, M. F. P. (2014). Portable and transparent host-device communication optimization for GPGPU environments. In *CGO*, pages 1–10. ACM.
- Meenderinck, C. and Juurlink, B. (2011). Nexus: Hardware support for task-based programming. In *DSD*, pages 442–445. Springer.
- Mendonça, G., aes, B. G., Alves, P., Pereira, M., Araújo, G., and ao Pereira, F. M. Q. (2016). Automatic insertion of copy annotation in data-parallel programs. In *SBAC-PAD*, pages 1–8. ACM.

- Metzger, R. and Stroud, S. (1993). Interprocedural constant propagation: an empirical study. *LOPLAS*, 2(1-4):213--232.
- Mock, M., Das, M., Chambers, C., and Eggers, S. (2001). Dynamic points-to sets: A comparison with static analysis and potential applications in program understanding and optimization. In *PASTE*, pages 66--72. ACM.
- Moreira, K. C. A., Mendonça, G. S. D., Guimarães, B. C. F., Alves, P. R. O., and Pereira, F. M. Q. (2016). Paralelização automática de código com diretivas OpenACC. In *SBLP*. SBC.
- Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. (2009). SoftBound: Highly compatible and complete spatial memory safety for C. In *PLDI*, pages 245--258. ACM.
- Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Gonnord, L., and Pereira, F. M. Q. (2014). Validation of memory accesses through symbolic analyses. In *OOPSLA*, pages 791--809. ACM.
- Necula, G. C., McPeak, S., and Weimer, W. (2002). Ccured: Type-safe retrofitting of legacy code. In *POPL*, pages 128--139. ACM.
- Oancea, C. E. and Rauchwerger, L. (2012). Logical inference techniques for loop parallelization. *SIGPLAN Not.*, 47(6):509--520.
- Oancea, C. E. and Rauchwerger, L. (2015). Scalable conditional induction variables (CIV) analysis. In *CGO*, pages 213--224. ACM.
- Pereira, F. M. Q. and Berlin, D. (2009). Wave propagation and deep propagation for pointer analysis. In *CGO*, pages 126--135. IEEE.
- Pouchet, L.-N. (2014). Polybench/C: the polyhedral benchmark suite. Available online.
- Reyes, R., López-Rodríguez, I., Fumero, J. J., and de Sande, F. (2012). AccULL: An OpenACC implementation with CUDA and OpenCL support. In *Euro-Par*, pages 871--882. Springer.
- Rugina, R. and Rinard, M. (2000). Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *SIGPLAN Not.*, 35(5):182--195. ISSN 0362-1340.

- Rus, S., Rauchwerger, L., and Hoefflinger, J. (2003). Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *International Journal of Parallel Programming*, 31:251–283.
- Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: a fast address sanity checker. In *USENIX ATC*, pages 28–28. USENIX Association.
- Surendran, R., Barik, R., Zhao, J., and Sarkar, V. (2014). Inter-iteration scalar replacement using array SSA form. In *Compiler Construction*, pages 40–60.
- Tabuchi, A., Kimura, Y., Torii, S., Matsufuru, H., Ishikawa, T., Boku, T., and Sato, M. (2016). Design and preliminary evaluation of omni openacc compiler for massive MIMD processor PEZY-SC. In *OpenMP: Memory, Devices, and Tasks (IWOMP)*, pages 293–305. Springer.
- Verdoolaege, S. (2010). isl: An integer set library for the polyhedral model. In *ICMS*, pages 299–302. Springer.
- Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F. (2013a). Polyhedral parallel code generation for CUDA. *TACO*, 9(4):54:1–54:23.
- Verdoolaege, S., Juega, J. C., Cohen, A., Gómez, J. I., Tenllado, C., and Catthoor, F. (2013b). Polyhedral parallel code generation for CUDA. *TACO*, 9(4).
- Wienke, S., Springer, P. L., Terboven, C., and an Mey, D. (2012). OpenACC - first experiences with real-world applications. In *Euro-Par*, pages 859–870. Springer.
- Wolfe, M. (1996). *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1st edition.
- Wolfe, M. J. (1995). *High Performance Compilers for Parallel Computing*. Addison-Wesley.
- Yang, Y., Xiang, P., Kong, J., and Zhou, H. (2010). A GPGPU compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97. ACM.
- Zhang, Q., Lyu, M. R., Yuan, H., and Su, Z. (2013). Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *PLDI*, pages 435–446. ACM.
- Zheng, X. and Rugina, R. (2008). Demand-driven alias analysis for C. In *POPL*, pages 197–208. ACM.