# Compiler Optimization via Superoptimization

LOUIS JENKINS, University of Rochester, USA

Program synthesis is a methodology for the synthesis, or creation, of programs from some high-level specification such as input-output examples. Such programs can not only be used for the sake of automation and convenience, but also efficiency. Compiler infrastructures such as LLVM provide reusable and general-purpose compiler optimizations, but these optimizations differ from superoptimization. Where compiler optimizations merely improve existing code, superoptimization seeks to synthesize an optimal replacement for the existing code. In the case of domain-specific compiler optimization, where the compiler leverages a priori knowledge about the semantics of the language that is being optimized, a superoptimizer will be at least as efficient in comparison. In this paper, I explore superoptimization and describe its more practical applications in compilers and programming languages.

Additional Key Words and Phrases: Synthesis, Superoptimization, Survey

**ACM Reference Format:**
Louis Jenkins. 2018. Compiler Optimization via Superoptimization. 1, 1 (December 2018), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION AND BACKGROUND

*Program synthesis* is a methodology for the *synthesis*, or creation, of programs from some high-level *specification* such as input-output examples. Program synthesis, interchangeably referred to as synthesis, is not restricted to a particular domain or area of Computer Science. As Gulwani[7] described at length, program synthesis is a multidisciplinary effort that can be segmented into three dimensions. The first dimension, being the interpretation of user intent into a high-level specification, is commonly handled by Human-Computer Interaction (HCI) and Natural Language Processing (NLP). The second dimension, being the leveraging of domain-specific knowledge in the selection or pruning of the program space of potential *candidates*, or programs that may satisfy the specification, can be handled by Information Extraction, Graphics, and Programming Languages. The third dimension, being the search over that particular search space can be intuitively handled via Logical Representation and Machine Learning (ML).

Superoptimization primarily focuses on the third dimension, where the superoptimizer will use the original code sequence and *test cases*, or input-output examples, as a specification in locating the *optimal* code sequence that *satisfies* that specification. A code sequence is optimal if it yields the lowest execution time to complete, and satisfies the specification if for all test cases it matches the expected output on all inputs. As the space of candidate programs is extremely large, it is often restricted in some way, normally to some subset of an Instruction Set Architecture (ISA). As superoptimizers focus on optimality, assembly is often used to avoid being hindered by abstraction. This does not mean that users must program in assembly to benefit from superoptimization, as higher-level programming languages are compiled down to assembly.

Author's address: Louis Jenkins, University of Rochester, 60 Crittenden Blvd, Apt 110, Rochester, NY, 14620, USA, louis.jenkins@cs.rochester.edu.


XXXX-XXXX/2018/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Superoptimization, originally coined by Massalin [15], started with techniques that would now be considered naive, which was an exhaustive enumerative search over potential candidate rewrites. Each rewrite would incrementally increase the number of instructions to be synthesized, with the assumption that shorter instruction sequences were faster than longer, hence yielding the optimal instruction sequence first. Massalin's superoptimizer was unfortunately very slow, only being able to produce 12 instructions, limited to 5 different instructions, within a feasible amount of time. Years later,Schkufza et al. [20, 21] introduced STOKE, a stochastic superoptimizer that made use of Markov-Chain Monte-Carlo (MCMC) sampling to find optimal code sequences, and used in conjunction with simulated annealing, called Metropolis-Hastings, to find its way out of local maxima. This approach lead to significant improvements in the available search space, allowing STOKE to search over 400 instructions for the same instruction length as Massalin's. This was also thanks to improvements to Satisfiability Modulo Theories (SMT) solvers, such as Z3 [5], which are able to determine the equivalence of different functions. As well, the rise of CounterExample Guided Inductive Synthesis (CEGIS) [22] lead to optimization in how often the SMT solvers, which are used as oracles, are invoked, as it enables synthesizers to learn by counterexamples provided by the oracle to filter out more candidates.

Finally, there are immediate applications of superoptimization, being the incorporation of superoptimization into compiler infrastructures such as LLVM[14]. Souper[19] and GreenThumb[17] are two such applications. While the number of applications in compiler optimization to-date are rather limited, there are some observations that can be made towards its applications in the Partitioned Global Address Space, which is discussed in section 3.

## 2 SUPEROPTIMIZATION

The goal of superoptimization is to *rewrite* a program such that it is an *optimal* equivalent of the original program. Optimality can be established by finding the global minima of a *cost function*. While optimality is normally applied to performance and execution time, it is not always the case and is application specific. To determine that these rewrites are correct, that is that they satisfy some *specification*, a *verifier* is needed as an oracle to determine whether the rewrite is indeed an equivalent of the original code. Specifications are sometimes given as input-output examples that will determine whether two functions match, but sometimes it can be given as a formal specification. One variant, called Symbol Execution, seeks to explore all *paths*, or conditional branches, and the constraints that will result in traveling down those paths. To determine equivalence, both the original and the rewrite must travel down the same paths given the same input. Symbolic Execution is one such verifier that can be used in practical real application in that it is able to execute code that cannot be represented as formal logic like what an SMT solver requires. It is not perfect, as it can only explore a finite number of paths, and if the constraints are too rigid they may take a while to find. As well side-effect inducing behavior of external libraries and system calls add difficulty, but they are not unsolvable.

### 2.1 Existing Applications

Souper [19] is a superoptimizer that was designed and created by Google to superoptimize LLVM Intermediate Representation (IR). Souper features its own IR that resembles a subset of LLVM IR that is control-flow free and restricted to integer and bit vectors. The construction of Souper IR from LLVM IR can be performed by using backwards data-flow analysis on functions returning integers until a non-supported instruction is found. Hence, Souper IR is only capable of optimizing the *tail* of integer-returning functions. However, given what limited instruction sequences it can optimize, it is sufficient for many real-world problems as it has claimed to discover 7,900 new optimizations that LLVM was unable to perform, with some even being incorporated into LLVM. While Souper's performance benefits are relatively low, estimated to be about 2% over normal compiler optimizations, it serves an interesting proof of concept. The restriction of being only able to evaluate integers, and the fact that it cannot attempt to synthesize and superoptimize instructions which operate on memory, floating-point numbers,

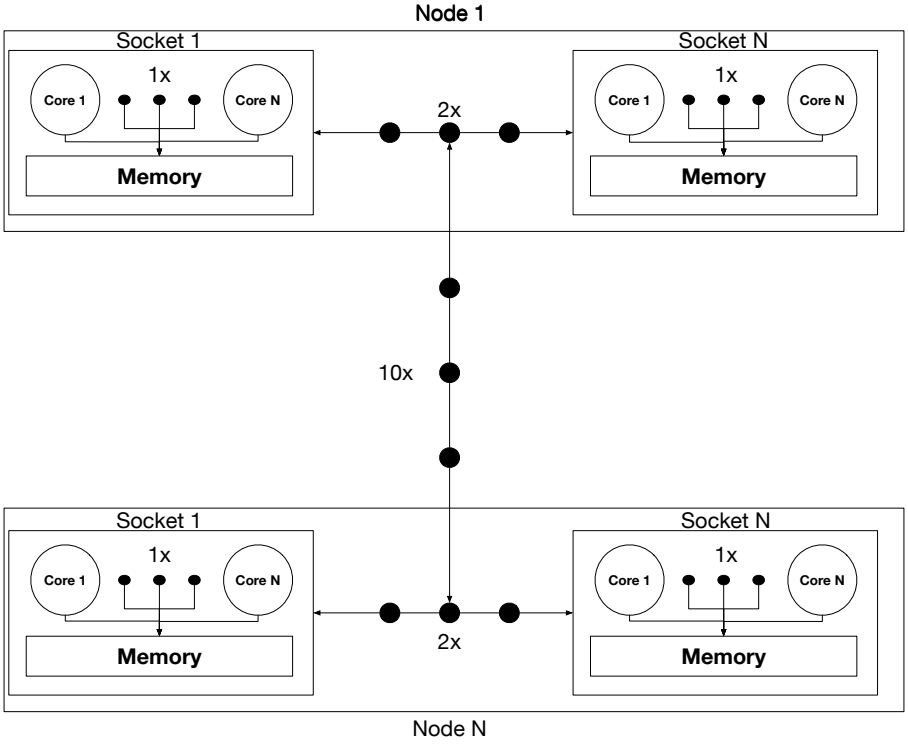


Fig. 1. Memory hierarchy of a supercomputer. Processors with associated memory are grouped into NUMA sockets. Compute nodes are often made up of multiple NUMA sockets, which establish a heirarchy of memory. Accessing memory that is local to a socket is given a multiplier of 1x, accessing memory that is on the same compute node in a different socket is given a multiplier of 2x, and accessing memory on another compute node is given a multiplier of 10x.

pointers, arrays, and so forth limits its utility. GreenThumb is a similar framework to Souper, but differs in that it is more generic and can be implemented for any ISA, not just LLVM.

## 3 COMPILE-TIME AGGREGATION UNDER THE PARTITIONED GLOBAL ADDRESS SPACE

With the eager advancements in computational power and with accordance to Moore's Law, the need to *distribute* computations across multiple processing elements has arisen. Each processing element (PE) consists of computational resources, such as processors, and memory that the computational resources are associated with. The need for such can be seen in the coming age of Non-Uniform Memory Access (NUMA, [12]) architectures, where groups of processors and associated memory are placed into *sockets*. *Locality* is established as a property of how close the computational resource is from memory, and maximizing locality can greatly impact performance. As an approximation, accessing data from L1 cache is an order of magnitude faster than accessing L2 cache which is an order of magnitude faster than accessing L3 cache. As a means of simplification, we can assume that each time we go down a part of the memory hierarchy, it will take an order of magnitude longer to access. The only exception to this generalization is accessing memory across NUMA sockets on the same *compute node*, which is estimated to be only 2x slower than accessing memory that is on the same socket. However, accessing memory in a different compute node is about an order of magnitude slower. Hence, the relative difference between accessing

memory in cache and accessing memory that is *remote*, which we restrict this term to mean memory that is hosted on another compute node, is 100x to 1000x slower.

The Partitioned Global Address Space (PGAS) is a memory model where memory is addressable as if it were single global address space, but partitioned in a way such that some of that memory is *local*. Where loads and stores are the primitives for accessing local memory, GETs and PUTs are the primitives for accessing remote memory. GETs and PUTs are implemented in one of two ways: 1) Remote Direct Memory Access (RDMA) which are handled by specialized hardware, or as Active Messages [23] which are requests with respective handlers that are handled by the CPU. *Aggregation*, the coarsening of the granularity of accesses of remote memory, is a very common optimization technique. There are many implementations of aggregation, such as the general-purpose active message library AM++[25], the language-specific Topological Routing and Aggregation Module (TRAM, [24]) for Charm++[10], the PGAS-specific Software Cache[6] and Locality-Aware Productive Prefetching Support (LAPPS, [11]), and the Chapel Aggregation Library (CAL, [9]) are all examples of efforts to provide aggregation support. While there are many examples of run-time support for aggregation, there is little support that, to the author's knowledge, provide aggregation through compile-time transformations.

As aggregation libraries generally complicate user code when made explicit such as in the case of AM++, or induce non-trivial performance overhead when performed implicitly such as for the Software Cache, compiler-based aggregation can provide a more transparent approach that comes with best of both worlds. If aggregation can be seen as a compile-time optimization, it only has a one-time overhead for compilation with the benefit of improved performance without the programmer having to do anything or make any changes to their code. One such example has been proposed by Hayashi et al. [8], where PUT and GET operations to adjacent memory to the same PE get aggregated into a single message. This is performed by transforming PUTs and GETs into individual loads and stores to a distinct *addrspace* in LLVM Intermediate Representation (IR). By modeling PUTs and GETs in this fashion, these PUTs and GETs can take advantage of certain compiler optimizations such as Loop-Invariant-Code-Motion. As well, these modeled loads and stores can more easily determine whether they are being performed on provably adjacent memory and can then be combined. After performing said optimizations, these modeled loads and stores can be lowered back to their original form as PUTs and GETs, benefiting from all of the optimizations that were applied. Unfortunately, PUTs and GETs to arbitrary locations and to different PEs are not aggregated, which means the coverage of such optimizations is very limiting despite it's novel achievements.

### 3.1 Observations

Following on the idea of modeling PUTs and GETs as stores and loads to a different addrspace to enable optimization, perhaps more things can be modeled in terms of LLVM IR. Using the Chapel programming language [3, 4] as the prime example, aggregate types are represented as either a *class*, which is a heap-allocated object and is passed by its pointer, or a *record*, which is stack-allocated and passed by value or by reference. These aggregate types are represented in LLVM IR as a *struct*, which is one way the compiler is able to prove that such memory is adjacent to each other.[1] However, there are times when memory cannot be proven to be adjacent, but can still be benefit from aggregation. PUTs and GETs are handled sequentially which provide familiar semantics to accessing distributed memory, but they do not *need* to be sequential, they just need to complete prior to the first time that it is used. Hence, PUTs and GETs can be and arguably should be aggregated whenever possible whether they are adjacent or not.

Extending on the idea of combining adjacent PUTs and GETs into a single PUT and GET respectively, I would like to propose that non-adjacent PUTs and GETs should be combined into *scatter*, *gather*, and *scattergather*

[1] Another more common scenario is when you have a task migrating from one PE to another, the task on the destination PE can access variables in lexical scope of the task in the source PE. Variables can be accessed relative to the stack frame pointer, which can be used to prove loads and stores to these as being adjacent.

instructions. The scatter and gather will, in parallel, send a respective PUT and GET to multiple destinations, awaiting completion of all destinations rather than sequentially awaiting each individual destination, while scattergather will perform both scatter and gather at once, in parallel. If the network implements any of these operations, as is common in many recent network interconnects, they can be used in place of parallel individual PUTs and GETs.

## 3.2 Motivating Example

In the case of loop kernels, which are very much so common in high-performance computing, in particular for simulations and loop kernels. In many of these kernels, each iteration of a loop often accesses multiple arrays and in multiple sections, sometimes of which may be distributed over multiple PEs. As a simple example, imagine the case where the desire is to perform point-wise vector addition over K arrays. The arrays merely have to be of the same size, but they do not need to share the same distribution, that is they can be distributed over different PEs.

```
A = B + C + /* ... */ + Z;
```

Imagine that this is implemented by *zipping* over arrays $B$ through $Z$. This can be handled by zipping through the distributed domain of $A$, ensuring that parallelism and locality follows the shape of where the output is going.

```
forall i in A.domain {
        A[i] = B[i] + C[i]  + /* ... */ + Z[i];
}
```

This patterns is relatively common, and can be observed in the main kernel loop body of the HPC Challenge STREAMS benchmark [16].

```
C = A;
B = scalar * C;
C = A + B;
A = B + scalar * C;
```

The takeaway from this is that compile-time aggregation can directly benefit this case, whether it is in the extreme case of vector point-wise addition of K arrays, or vector point-wise addition of 2 arrays with a little vector multiplication added to the mix.

*3.2.1 Modeling of Chapel Arrays in LLVM IR.* Similar to how PUTs and GETs are modeled as loads and stores to separate address spaces, the goal is to model Chapel's arrays somehow in LLVM IR. Chapel's arrays are very complex and have a lot of internal machinery that can impede any optimization if not modeled in some way. Furthermore, LLVM requires arrays to have a constant known size but the domain of an array is not guaranteed to be constant, and so instead representing Chapel's arrays as LLVM arrays, they can be represented as pointers. Chapel's arrays have the domain as well as distribution tied to it's type, and so distributed arrays can be represented as pointers into some addrspace, similar to how PUTs and GETs are represented.

*3.2.2 Aggregation.* In the case of vector point-wise addition via zipping K arrays, one can imagine that the original LLVM IR for the body of the loop would look like such.

```
%B_p = getelementptr ... ; ref to B[i]
%B_v = load i64 addrspace(100)* %B_p
%C_p = getelementptr ... ; ref to C[i]
```

```
%C_v = load i64 addrspace(100)* %C_p
; ...
%Z_p = getelementptr ... ; ref to Z[i]
%Z_v = load i64 addrspace(100)* %Z_p
; Sum B_v to Z_v into value %val
%A_p = getelementptr ... ; ref to A[i]
store i64 %val, i64* %A_p
```

The above would result in K sequential GETs, but it does not need to be that way. Assuming that locating the element can be done without any communication, which is true with Chapel's standard distributions such as Block and Cyclic, the loads can be reordered such that they are placed in proximity, after locating all of the elements. Even in the case that locating the elements requires some communication, there will still be a benefit. In Chapel, indexing into an array is handled by first locating a *reference* to the element and returning that, in which case is written to or read from by the user. If obtaining the reference requires communication, the communication from sequentially accessing the reference can still be avoided.

```
%B_p = getelementptr ... ; ref to B[i]
%C_p = getelementptr ... ; ref to C[i]
; ...
%Z_p = getelementptr ... ; ref to Z[i]
%B_v = load i64 addrspace(100)* %B_p
%C_v = load i64 addrspace(100)* %C_p
; ...
%Z_v = load i64 addrspace(100)* %Z_p
; Sum B_v to Z_v into value %val
%A_p = getelementptr ... ; ref to A[i]
store i64 %val, i64* %A_p
```

Given that these loads are grouped together, they can then be performed in a single gather instruction.

```
%B_p = getelementptr ... ; ref to B[i]
%C_p = getelementptr ... ; ref to C[i]
; ...
%Z_p = getelementptr ... ; ref to Z[i]
; Gather into a single array of size K for K inputs
%val_p = gather %B_p, %C_p, ..., %Z_p
; Sum the array val_p into value %val
%A_p = getelementptr ... ; ref to A[i]
store i64 %val, i64* %A_p
```

All of this can be handled as a peephole optimization, but then it would be restricted to transformations for which it can statically prove. Much more aggressive transformations can be performed via peephole superoptimization.

### 3.3 Peephole Superoptimization

While the approach using peephole optimization has the potential to greatly impact the performance of code that centers around arrays, it would not be able to optimize for general-purpose routines and objects. For example, the optimization discussed earlier only works due to the first-class treatment that arrays and domains get in Chapel,

but would fail to do anything in other PGAS languages which do not provide such first-class treatment or have such meta-information available at compile time. As well, if the user were to write code that can obviously be improved with aggregation, if the accesses were not provably adjacent or to arrays, they would need to perform their own explicit aggregation which is undesirable.

Peephole superoptimization, originally a term coined by Bansal and Aiken [1], is the application of superoptimization to peephole optimization. In [1], peephole superoptimizations are generated from patterns learned by *harvestng* loop-free instruction sequences from a training dataset, being the optimized output from a compiler; using *canonicalization* to rename registers and constants relative to the order in which they are first used; and then *fingerprinting* enumerated candidate instruction sequences for reuse. While this approach was originally designed for the x86 Instruction Set Architecture (ISA), it can possibly be generalized to work with any ISA such as LLVM IR. For constructing the training data, heavily optimized locality-aware applications can be harvested with the fingerprint of the canonicalized template stored in the optimization database.

Chapel's *multiresolution design philosophy* [2] supports providing user with access to lower-level abstractions and language features when necessary, such as for performance reasons. It would not be against the design of Chapel to expose explicit calls to scatter, gather, and scattergather, and so this can enable a learning peephole superoptimizer to learn more. This act of learning patterns from harvested training data and reusing those patterns to replace suboptimal instruction sequences with optimal instruction sequences can be useful outside of compile-time aggregation. For example, given that multiresolution design philosophy allows the user to access lower-level abstractions and constructs, once enough applications make use of this pattern, that pattern can then be harvested and applied to the higher-level code. In the end, the issue of seeking out lower-level to combat a lack of performance at a higher-level may resolve itself over time without further intervention from the developers and the language may be able to maintain itself.

While the end result may sound like a pipe dream, it can at least be kick-started by finding communication patterns. To do so, we must first establish the *search space* that will be searched, the *cost function* for a given candidate rewrite, and *verification* to determine equivalence of a rewrite and the original program..

*3.3.1 Search Space.* The space of which candidates are derived from consist of loop-free code sequences between memory fences, as these are safe to be reordered by the compiler. Furthermore, the space is pruned of candidates that violate a data dependency, significantly reducing the overall search space and improving potential performance. As our goal is to group together loads and stores, we do not need to insert or remove code, and instead the only operation that needs to be performed would be moving a load or store instruction. Due to the implicit nature of remote memory access in Chapel, sequences of fine-grained remote memory access is more common than one would expect, and so even though the amount of code that can be safely superoptimized is limited to pointer dereferencing, it should supply an ample amount of coverage. Indeed, because return values from functions are first stored into a temporary, it forms a data dependency and so will be pruned away.

The rationale is that since both the compiler and CPU is free to reorder loads and stores that are not across a memory fence, we can group together loads and stores to potentially remote memory. These grouped loads and stores can then be turned into a single scatter or gather instruction. As a further optimization, a scatter and gather can be combined into a single *scattergather* that will handle both in an asynchronous manner, further reducing latency.

*3.3.2 Cost Function.* To find the optimal rewrite, each candidate must have associated with it a cost. To define this cost, for each individual load or store to a global pointer that cannot be aggregated, we give it the cost $c$, which is given the same cost as an individual scatter, gather, and scattergather call. The cost function ensures that the synthesizer will select candidates that result in the fewest number of loads, stores, scatter, gather, and scattergather calls which we define as *optimal* in terms of latency. To prevent getting caught in a local minima, a Markov-Chain Monte-Carlo (MCMC) algorithm such as Metropolis-Hastings can be employed.

*3.3.3 Verification.* UC-KLEE [18] is a symbolic execution verification tool that determines the equivalence of two routines written in C. To use the tool, some modifications may be required to work directly on LLVM bitcode, but in the worst case a decompiler can be used to reconstruct the C program. UC-KLEE is able to determine, down to the last bit, if two routines given the same input state will yield the same output state, including the state of heap, stack, registers, and global variables. This is done by using *shadow memory* where metadata is kept about the state of memory, and instrumentation is performed to update and check the state of this shadow memory to determine certain information. For example, UC-KLEE treats all data as raw bytes, but the first time a symbolic variable is dereferenced, it will mark that memory location as memory to be compared across routines later. Unfortunately, UC-KLEE has some severe limitations: it will only keep track of a finite input size and will only check a finite amount of code paths in both routines, but it should serve as an intermediate solution for finding easy-to-prove superoptimizations for candidates. In particular, UC-KLEE will come into play for when data dependencies, such as those obtained from side-effect inducing functions, are moved closer to other loads and stores.

Alive[13] is a viable alternative or augmentation to UC-KLEE in that it proves correctness of peephole optimizations such as this via use of SMT solvers. Alive is a domain-specific language that models the LLVM IR to match its look-and-feel while providing extensions such as type-inference and polymorphic transformations. Alive is capable of proving the equivalence of the original program and a rewrite by satisfying three conditions: 1) An operation defined in the original should be defined in the target, 2) An operation should only produce *poison values*, which is undefined behavior under certain conditions, in the rewrite if it does so in the original, and 3) The results of instructions in both the original and rewrite must match when conditions 1 and 2 hold.

Unlike UC-KLEE, Alive is capable of proving absolute correctness of code that it can model, but it is limited to verification of that model. In the Alive language, it currently only supports integers, pointers, arrays, and void types, which limits the number of peephole optimizations to sequences of code that consist of just those types. Luckily, this particular peephole optimization, being the reordering of loads and stores, does not require much more as only code inside of memory barriers are considered and hence neither atomicity or thread-safety need to be considered. However, as Alive lacks support for floating points, which are relatively common in scientific computing, the symbolic execution of UC-KLEE can be utilized as not just additional verification, but for verifying code that Alive cannot. Hence the two can work well together.

*3.3.4 Expected Goal.* Given the restriction of the search space to loop-free code sequences between memory fences, the amount of pruning of candidates that violate data dependencies, and the relatively simple scheme of moving instructions, this should result in high performance. Furthermore, the problem is embarrassingly parallel and can be sped up and even distributed. The benefit of automatic aggregation without any run-time modification could prove to be a notable scientific contribution, in particular for the realm of high-performance computing.

## 4 CONCLUSION

Superoptimization is a particular area of program synthesis where the optimal code sequence is produced by rewriting of the original code sequence. Superoptimization has the potential to yield some very interesting and fruitful applications, in particular for compiler optimization. In this paper, I discussed the applications of superoptimization to LLVM, the ever-growing compile infrastructure and an ever-growing user-base. As well, I presented a potential application of superoptimization under the Partitioned Global Address Space memory model, in particular for the Chapel programming language, that can serve to significantly speed up distributed applications by aggregating PUTs and GETs into scatter, gather, and scattergather instructions.

## REFERENCES


[1] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

[2] Bradford L Chamberlain et al. 2007. Multiresolution languages for portable yet efficient parallel programming. (2007).

[3] Bradford L Chamberlain, David Callahan, and Hans P Zima. 2007. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.

[4] Bradford L. Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, and Greg Titus. 2018. Chapel Comes of Age: Making Scalable Programming Productive. *Cray Users Group* (2018).

[5] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[6] Michael P Ferguson and Daniel Buettner. 2015. Caching Puts and Gets in a PGAS language runtime. In *2015 9th International Conference on Partitioned Global Address Space Programming Models (PGAS)*. IEEE, 13–24.

[7] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1836089.1836091

[8] Akihiro Hayashi, Jisheng Zhao, Michael Ferguson, and Vivek Sarkar. 2015. LLVM-based Communication Optimizations for PGAS Programs. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*. ACM, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/2833157.2833164

[9] Louis Jenkins, Marcin Zalewski, and Michael Ferguson. [n. d.]. Chapel Aggregation Library (CAL). ([n. d.]).

[10] Laxmikant V Kale and Sanjeev Krishnan. 1993. CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, Vol. 28. ACM, 91–108.

[11] Engin Kayraklioglu, Michael P Ferguson, and Tarek El-Ghazawi. 2018. LAPPS: Locality-Aware Productive Prefetching Support for PGAS. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 3 (2018), 28.

[12] Christoph Lameter. 2013. NUMA (Non-Uniform Memory Access): An Overview. *Queue* 11, 7, Article 40 (July 2013), 12 pages. https://doi.org/10.1145/2508834.2513149

[13] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

[14] Nuno P Lopes and John Regehr. 2018. Future Directions for Optimizing Compilers. *arXiv preprint arXiv:1809.02161* (2018).

[15] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS II)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 122–126. https://doi.org/10.1145/36206.36194

[16] John D McCalpin et al. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 1995 (1995), 19–25.

[17] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Greenthumb: Superoptimizer construction framework. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 261–262.

[18] David A Ramos and Dawson R. Engler. 2011. Practical, Low-effort Equivalence Verification of Real Code. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 669–685. http://dl.acm.org/citation.cfm?id=2032305.2032360

[19] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *arXiv preprint arXiv:1711.04422* (2017).

[20] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. *SIGPLAN Not.* 48, 4 (March 2013), 305–316. https://doi.org/10.1145/2499368.2451150

[21] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stochastic Program Optimization. *Commun. ACM* 59, 2 (Jan. 2016), 114–122. https://doi.org/10.1145/2863701

[22] Armando Solar Lezama. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html

[23] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. 1992. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*. ACM, New York, NY, USA, 256–266. https://doi.org/10.1145/139669.140382

[24] Lukasz Wesolowski, Ramprasad Venkataraman, Abhishek Gupta, Jae-Seung Yeom, Keith Bisset, Yanhua Sun, Pritish Jetley, Thomas R Quinn, and Laxmikant V Kale. 2014. Tram: Optimizing fine-grained communication with topological routing and aggregation of messages. In *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 211–220.

[25] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. 2010. AM++: A generalized active message framework. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 401–410.