# Quantifying the Semantic Gap Between Serial and Parallel Programming

Xiaochun Zhang*
*Huawei*
dennis.zh@live.com

Timothy M. Jones
*University of Cambridge*
timothy.jones@cl.cam.ac.uk

Simone Campanoni
*Northwestern University*
simone.campanoni@northwestern.edu

2021 IEEE International Symposium on Workload Characterization (IISWC) | 978-1-6654-4173-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/IISWC53511.2021.00024

***Abstract*—Automatic parallelizing compilers are often constrained in their transformations because they must conservatively respect data dependences within the program. Developers, on the other hand, often take advantage of domain-specific knowledge to apply transformations that modify data dependences but respect the application's semantics. This creates a semantic gap between the parallelism extracted automatically by compilers and manually by developers. Although prior work has proposed programming language extensions to close this semantic gap, their relative contribution is unclear and it is uncertain whether compilers can actually achieve the same performance as manually parallelized code when using them. We quantify this semantic gap in a set of sequential and parallel programs and leverage these existing programming-language extensions to empirically measure the impact of closing it for an automatic parallelizing compiler. This lets us achieve an average speedup of 12.6× on an Intel-based 28-core machine, matching the speedup obtained by the manually parallelized code. Further, we apply these extensions to widely used sequential system tools, obtaining 7.1× speedup on the same system.**

## I. Introduction

One reason that automatic parallelization is still an open research problem, despite decades of research and impressive recent performance demonstrations [1]–[16], is that developers extract parallelism in different ways to compilers. Often developers can take advantage of their knowledge of a program's semantics to understand the transformations that can be legally applied, given the application's domain. These may result in minor differences in the program's output or subtlely different techniques to perform the same algorithm. In effect, there exists a semantic gap between how the developer extracts parallelism and what a compiler can achieve that limits the performance of automatic parallelization. Because of this, several programming language extensions have been proposed to close this semantic gap, but they only have been evaluated *in isolation*, leaving uncertainty about their relative contribution. Moreover, no study has shown that compilers can actually generate the same manually parallelized code when these language extensions are used for irregular programs.

To shed light on this open problem, we have studied sequential and parallel versions of a range of applications written in C and C++ to determine how much parallelism compilers and developers are able to extract. To this end, we measure both manually defined and automatically generated thread-level parallelism (TLP) through both limit studies and measurements in a real system. Both studies show that for some applications, this gap is significant, severely hindering the speedups achievable through automatic parallelization. We show that this gap is composed of four common techniques in our workloads, termed the semantic-gap components, that developers use to incorporate semantic knowledge into their applications when parallelizing. We explore how each component contributes to the difference between automatic and manual parallelization at the limit by modeling an idealized multicore system. Our results show that, at the extreme, closing the semantic gap converts a 2× speedup into a 3,463× speedup and that often developers need to consider two or more semantic-gap components to realize the majority of the speedups available.

We then bridge the semantic gap by augmenting our applications with programming-language extensions, all previously proposed for other scenarios. Doing so allows us to empirically measure the impact of these extensions for unblocking parallelism within an automatic parallelizing compiler, achieving an average speedup of 12.6× on an Intel-based 28-core machine matching the speedup obtained by the manually parallelized code. We then consider a range of system tools, which are a compelling use-case for our extensions, being codes that are widely available, have existed for many years, and yet remain sequential. Closing the semantic gap through the same extensions brings an average speedup of 7.1× on the same system.

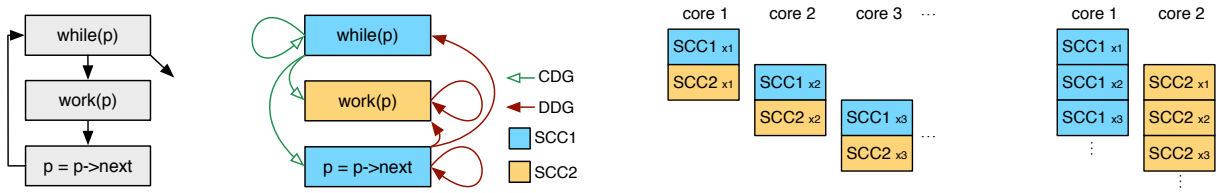
In summary, this paper makes the following research contributions.

- It evaluates the semantic gap between serial and parallel programming in widely-used benchmarks.
- It characterizes this gap highlighting its components and the contribution each makes to closing the gap.
- It empirically measures the impact of existing, unobtrusive language extensions to bridge the semantic gap between manual and automatic parallelization.
- It evaluates the performance a parallelizing compiler could achieve when the semantic gap is closed with oracle data-dependence information.

Next we describe how we model parallelism. Then, we describe the semantic gap we found on the benchmarks we have analyzed and its components.

*Work performed whilst at the University of Cambridge.

978-1-6654-4173-5/21/$31.00 ©2021 IEEE
DOI 10.1109/IISWC53511.2021.00024




(a) Original code (b) PDG (c) DoAcross (d) DSWP

Fig. 1: Parallelizing compilers split sequential code into SCCs. The function $work(p)$ produces an internal data dependence on itself. Two SCCs are identified in this example using the program dependence graph (PDG) made up of the control dependence graph (CDG) and data dependence graph (DDG). We evaluate DSWP and DoAcross by using SCCs as an unbreakable unit.

## II. Modeling Methodology

To understand why the TLP that developers currently extract manually is not within the reach of automatic parallelizing compilers, this paper measures both manually parallelized code as well as the largest amount of parallelism that can be extracted automatically. The former can be achieved using similar techniques to those in the literature [17]. The latter, instead, requires modeling because many parallelizing compilers have been proposed for automatically extracting TLP from a sequentially designed codebase. To model their limit, we focus on their common feature, which splitting sequential code into strongly-connected components (SCCs) [3], [5], [18] within the program dependence graph (PDG) [19]. Therefore, we model the parallelism extracted by parallelizing compilers by using SCCs of the original code as units of execution. This measures the TLP limit of both cyclic multithreading (CMT) (e.g., DoAcross [20], HELIX [3]) and pipeline multithreading (PMT) (e.g., DSWP [5], PS-DSWP [8]).

The observation that the code generated by parallelizing compilers can be modeled using SCCs sets this analysis apart from prior TLP and instruction level parallelism (ILP) analyses. We call our evaluation an *SCC analysis*. A TLP analysis measures the amount of *coarse-grained* parallelism *expressed* in the code and an ILP analysis measures the amount of *fine-grained* parallelism *extractable* from a sequential execution, whereas an SCC analysis, introduced in this paper, measures the amount of *coarse-grained* parallelism *extractable* from a sequential execution.

### A. Using SCCs as a Unit of Parallelism

To understand how compilers extract parallelism, we use an example of sequentially-designed code, shown in figure 1. The original C-like code is shown figure 1(a) as a control-flow graph (CGF), split into basic blocks. This is a linked-list traversal, where each iteration processes one element of the list, pointed to by $p$. Figure 1(b) shows the PDG of this program using the basic blocks as vertices, made up from the control dependence graph (CDG) on the left and data dependence graph (DDG) on the right, conservatively generated by LLVM's alias analysis. Two SCCs are created from this graph, as shown. The first (in blue) is due to the control dependence that the bottom vertex has on the top (since the $while(p)$ vertex determines whether $p = p\text{->}next$ gets executed), combined with the data dependence the top vertex has on the bottom, through the $p$ variable. The second (in orange) is due to the compiler identifying an internal data dependence within the $work$ function, meaning it has a data dependence with itself.

We can classify parallelizing compilers to a first approximation according to how SCCs are scheduled between cores. CMT techniques schedule invocations of an SCC across cores (figure 1(c)). PMT techniques schedule them within the same core and schedule different SCCs on different cores (figure 1(d)). While these techniques are fundamentally different, their commonality is that all invocations of a single SCC are executed sequentially (within or across cores). Hence, we can measure the limit of all parallelization techniques by using an SCC as unbreakable unit of parallelism.

### B. SCC Limit Study

The performance of the code generated by parallelizing compilers is limited by the SCCs identified at compile-time. The conservativeness of dependence analysis (e.g., alias analysis) means some SCCs identified at compile time are unnecessary. For example, SCC2 in figure 1(b) is unnecessary when all elements of the list are unique because there is no internal data dependence in the `work()` function. Hence, state-of-the-art parallelizing compilers rely on speculative execution to overcome this limitation.

To model these latest compilers and at the same time make our analysis independent of the accuracy of a given dependence analysis, we identify SCCs based only on dynamic data dependences. To this end, we built a tool called *oracle-extractor*, which automatically computes the oracle information for data dependences that occur at run-time for a given input. We use these oracles to identify the minimum set of data dependences that must be satisfied to preserve the original program semantics. Oracle-extractor outputs these dependences as a dependence graph, which we call *minDDG*, which represents the best possible output of a data dependence analysis (i.e., only true dependences). Therefore, minDDG enables us to empirically measure the highest amount of parallelism that any compiler could extract starting from the sequentially-designed code. We use the SCCs found in minDDG to compute our SCC analyses.

We rely on minDDG to understand the difference between code generated by developers and parallelizing compilers, the semantic gap. Another gap that has been studied in the past to understand how to evolve programming languages is the analysis gap [21], [22]. This refers to the complexity of identifying a code property automatically (e.g., pointer aliasing) and it does not compare sequential and parallel programming. Hence, it cannot be used to answer our research question, so we focus on the semantic gap to compare the limits of parallelizing compilers with parallel programming.

### C. Measuring the Parallelism

We designed a tool called *parallelism-measurer* to measure both SCC and TLP limits. The former is performed by constraining all invocations of a given SCC to run sequentially. Instructions that do not belong to any SCC are modeled in the same way ILP and TLP limit studies [17] model all instructions: using a data-flow model constrained by dynamic data dependences. Finally, the execution between different SCCs and instructions that do not belong to any SCC is emulated using the same data-flow model: only data dependences identified at run-time serialize them.

Parallelism-measurer models the TLP limit of the manually-parallelized version of a benchmark. To do so, we have manually defined the DDG by studying the parallelization performed by developers of the target benchmark. We call it *manualDDG* and we use it as input to parallelism-measurer when we emulate the parallelized version of the code.

### D. Instrumentation Framework

Both of our tools are built on a modeling framework we designed around the LLVM IR. Our framework first compiles an application's source files to LLVM bitcode and combines them into a single monolithic bitcode file. We then perform function inlining so as to mimic the optimizations that an automatic parallelizing compiler would perform to provide more opportunities for transformation and optimize at level O3. From here we add instrumentation code and compile down to a native binary.

We developed an LLVM pass to insert instrumentation callbacks at key points in the code. IR instructions are stored alongside the binary in segments according to the callback points. We also create a mapping between the two, so that we can consult the IR whenever a callback is triggered. The callbacks are compiled down to machine code along with the rest of the program. When running the application, these statically-determined instrumentation points enable a runtime handler to examine the execution state of the application to gather profile information about instructions and data, and model different forms of execution.

The instrumentation points allow us to selectively track the data and control flows that we are concerned with. To profile the application and model parallelism, we instrument: *(i) Loop boundaries*, adding callbacks at the start of a new loop invocation, the end of a loop invocation, and the start of each loop iteration. This means that during execution we can track exactly which loop nest any dynamic instruction is currently within; *(ii) Memory accesses*, adding callbacks to track data dependences through memory. Dependences through LLVM SSA names can be tracked statically; *(iii) Library function calls*, placing callbacks beforehand so as to correctly handle data dependences when they are invoked.

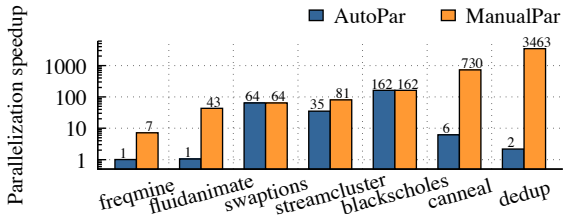



Fig. 2: Speedups of PARSEC benchmarks when performing automatic and manual parallelization (log-scale y axis).

## III. Semantic Gap

To understand why parallelizing compilers cannot extract the same amount of TLP as a programmer, we consider workloads from PARSEC [23] using simlarge inputs.

We studied their hot loops, in particular the way in which they have been parallelized manually by developers. Comparing that parallelism with the TLP extracted by compilers in an ideal situation (when relying on perfect run-time knowledge of dependences between instructions) allows us to understand how the TLP expressed by developers can be brought within a compiler's reach. We then identified the required semantic concepts to bridge this gap and previously proposed programming-language extensions to achieve this. We only consider a subset of applications from PARSEC as our tool-chain cannot compile all benchmarks (for more details, see section IV-A).

### A. Automatic versus Manual Parallelization

We performed a limit study to measure the difference between manual and automatic TLP extraction. To this end, we emulated an ideal machine with an infinite number of cores, no core-to-core communication latency, and infinite inter-core bandwidth. Section V relaxes these assumptions.

Figure 2 shows that only a fraction of the manually defined parallelism can be extracted automatically by compilers using the simlarge inputs to PARSEC, to keep emulation times manageable. Here, for ManualPar we used the manually parallelized version of each PARSEC benchmark; for AutoPar we either used the existing sequential versions (if present) or we compiled using a thread count of 1 to create a sequential version ourselves. Benchmarks such as *blackscholes* and *swaptions* represent the ideal case for compilers: they contain highly parallel loops where all of their iterations are independent of each other. However, for other applications there are clear differences. At the extreme, *dedup* has a speedup of 3,463× for the manually parallelized application, but only 2× using automatic parallelization.

*Freqmine* has the lowest speedup for manual parallelization of just 7.2×. This is partly due to the use of the simlarge inputs, with speedup rising to 24× with the larger, native inputs. However, it is still smaller than for other workloads stemming from unbalanced loop iterations due to its recursive tree traversals that are the core of the application.

These results show that although there is bountiful performance available in many of these workloads, automatic parallelization is unable to take advantage of it. Since data dependences place a limit on the parallelism available, these results show that the developers have managed to break certain dependences, yet the compiler is unable to replicate this. To understand why, we performed an in-depth study on these PARSEC benchmarks to identify situations where the developers have taken advantage of semantic knowledge about the application or algorithm to transform the code. Current compilers would never be able to apply these techniques, since the dependences are real. We call these techniques the semantic-gap components between the compiler's and programmer's understanding of the program and describe each one in the following section. As figure 2 shows there is no semantic gap in *blackscholes* and *swaptions*, we do not analyze them further. Although that only leaves five benchmarks to analyze, section V demonstrates that these are sufficient to capture the semantic-gap components across a wide range of workloads.

### B. Semantic Gap Breakdown

We discuss each semantic-gap component individually, illustrating them through code examples taken from our workloads and describing how they can be closed through previously proposed programming-language extensions.

*1) Algorithmic Options: canneal, dedup*

This component encompasses techniques that programmers use to extract parallelism by exploiting their knowledge of how the application works. In essence, this means that they break data dependences in a manner that preserves the overall intent of the algorithm while possibly sacrificing accuracy.

Figure 3 shows the algorithmic options semantic gaps for *depup* and *canneal*. In *canneal* (figure 3(a)) the loop at line 68 repeatedly selects an element at random, evaluates it against an existing element, and potentially swaps them. Therefore each iteration depends on the previous, since one of the elements remains the same between consecutive iterations. This creates an SCC that limits the parallelism of any automatic parallelizing compiler. In the parallel version, instead, this dependence is broken to avoid the related SCC: each thread starts with different random elements. Only the programmer knows that the algorithm does not depend on the preservation of this dependence because this information is not encoded in the source code. In fact, this dependence (and SCC), due to the reuse of an element, is an optimization added by the developer to improve cache behavior.

In *dedup* (figure 3(b)) input is split into variable-sized chunks based on Rabin fingerprints [24]. In the sequential

```
68 for (int i = 0; i < _moves_per_thread_temp; i++){
     ...
70   a = b;
71   a_id = b_id;
72   b = _netlist->get_random_element(&b_id,a_id,&rng);
     ...
87 }
```

(a) canneal (annealer_thread.cpp)

```
     // Sequential and parallel versions
 667 int offset = rabinseg(chunk->uncompressed_data.ptr,
         chunk->uncompressed_data.n, rf_win, rabintab,
         rabinwintab);
     ...
     // Parallel version
1108 int offset = rabinseg(
         chunk->uncompressed_data.ptr+ANCHOR_JUMP,
         chunk->uncompressed_data.n-ANCHOR_JUMP,
         rf_win_dataprocess, rabintab, rabinwintab);
```

(b) dedup (encoder.c)

```
class Dedup_chunkSize:Tradeoff_options{
  int64_t getMaxIndex(){ return 100; }
  auto getValue(int64_t i){ return 64 * i; }
  int64_t getDefaultIndex() { return 0; }
};
tradeoff DedupAlgorithmicOption_chunkSize {
  {Dedup_chunkSize};
};
```

(c) Programming-language extension used in *dedup* [13]

```
1107 auto chunkSize = DedupAlgorithmicOption_chunkSize;
1108 int offset = rabinseg(
         chunk->uncompressed_data.ptr+chunkSize,
         chunk->uncompressed_data.n-chunkSize,
         rf_win_dataprocess, rabintab, rabinwintab);
```

(d) Closing the algorithmic option semantic gap in *dedup*

Fig. 3: Algorithmic options.

version, input is read into a large buffer and chunks identified starting at its first byte, with each subsequent chunk starting at the byte following the previous. If the end of the buffer is reached without finding a fingerprint, unallocated data is moved to the start of the buffer and more input is read. Fingerprinting then restarts from the beginning of the buffer. This process causes a cross-iteration dependence, and therefore an SCC, in the loop that splits the input into chunks. Unfortunately, this SCC sequentializes the chunking.

The parallel version, however, avoids this SCC by breaking the related loop-carried dependence. This alters the output of the program but still creating a valid compressed file. Here the input is first split into coarse-grained fragments by searching for a fingerprint at a fixed offset (`ANCHOR_JUMP`) from the start of the input buffer. Each large fragment is passed to a different thread for chunking in the same manner as the sequential version of the program. Even though a fingerprint may not be found at the end of each fragment, the remaining

```
 960 long bsize = points->num/nproc;
 961 long k1 = bsize * pid;
 962 long k2 = k1 + bsize;
     ...
1039 //my *lower* fields
1040 double* lower = &work_mem[pid*stride];
1041 //global *lower* fields
1042 double* gl_lower = &work_mem[nproc*stride];
     ...
1044 for ( i = k1; i < k2; i++ ) {
       ...
1067   int assign = points->p[i].assign;
1068   lower[center_table[assign]] +=
          current_cost - x_cost;
       ...
1070 }
     ...
1079 for ( int i = k1; i < k2; i++ ) {
1080   if( is_center[i] ) {
         ...
1083     for( int p = 0; p < nproc; p++ ) {
1084       low += work_mem[center_table[i]+p*stride];
1085     }
1086     gl_lower[center_table[i]] = low;
         ...
1095   }
1096 }
```

(a) streamcluster (streamcluster.cpp)

```
1042 reassoc double* gl_lower = &work_mem[nproc*stride];
```

(b) Programming-language extension used in *streamcluster*

Fig. 4: Reassociativity.

bytes are treated as a chunk and sent to the next stage for processing. Therefore, the algorithm for input splitting is slightly different in the serial and parallel versions, leading to occasional changes in how the input is handled and different, but correct, outputs.

*Programming-language extension:* We leverage the trade-off interface described in [13], which makes algorithmic-specific trade-offs visible to a compiler tool-chain. For example, figure 3(c) shows the additional code to add to the original codebase to express that the chunk size of *dedup* can be changed if compilers have the need for it. Moreover, the original code needs to be changed to specify where the chunk size option is used (shown in figure 3(d)). Our parallelizing compiler uses the information encoded by the trade-off interface as a degree of freedom to remove dependences.

*2) Reassociation: streamcluster*

When developers parallelize a loop by means of a reduction, they break a read-after-write dependence by exploiting an associative operation. Reductions exploit the commutative properties of mathematical operations, such as calculating a running sum during execution of the loop. Reductions on scalars are automatically handled by many parallelizing compilers. The semantic-gap component of reassociation enables a reduction for types where it is unsafe—that is, reordering the operations may change the program's output.

Figure 4(a) shows reassociativity used in *streamcluster*. Each element of the `gl_lower` array is a sum of the cost of moving a point in the stream from one center to another, but its type is a `double`, which is unsafe to use in reductions. However, the developer has parallelized this code through a reduction by using an intermediate array, `lower`, for each thread (partial results collated in line 1068) and combining them in line 1086, accepting the imprecision that may result.

In fact, this example demonstrates not only the reassociativity semantic-gap component, but also an analysis-gap component too. Parallelizing compilers will automatically handle reductions on safe scalar values, but this example shows an array, where each element can be reduced.

*Programming-language extension:* We introduce a new type modifier, `reassoc`, which enables compilers to realize that they can change the associativity of the annotated variables of that type. For example, *streamcluster*'s array `gl_lower`, shown in figure 4(a), can be declared using this new modifier. The resulting code is shown in figure 4(b). The compile then knows it can change the associativity of the elements of the array `gl_lower`, allowing them to close and reduce each array element.

Our modifier `reassoc` is different from the LLVM option introduced in *clang 6* "`--reassoc`". This option enables the compiler to consider reassociation for all floating point variables, but is unfortunately not useful for parallelizing compilers because it is both too strict and too loose. It is too strict because we need reassociation for complex data structures like arrays rather than just scalar variables. It is too loose because enabling reassociation for all variables often leads to a change in output quality. Compiling *streamcluster* with this flag reduced output quality by 2% but using our `reassoc` modifier only for `gl_lower` (as performed by the parallel version of the benchmark) preserves the output quality.

*3) Data-Structure Duplication: canneal, freqmine*

Programmers often duplicate intermediate data structures to privatize the writes to them, allowing threads to run without synchronization. Knowing when this transformation is safe to apply and how to perform a deep copy of an object is something that existing compilers cannot infer. Data-structure duplication captures this semantic-gap component, whereby read-after-write dependences are broken without affecting the underlying algorithm.

In figure 5(a), for *canneal*, each thread executes the `Run` function and creates its own local copy of a pseudo random number generator (PRVG) (an `Rng` object). This PRVG contains a cross-iteration read-after-write dependence when it is used in a later loop, because it calculates the next random number as a function of the previous. The programmer breaks this dependence by duplicating and privatizing the object in each thread. The PRVG duplicate is generated by resetting its seed. This is a safe operation only because of the randomness of the implemented algorithm.

*Freqmine* (figure 5(b)) creates thread-local copies of structures used for memory management (`local_fp_tree_buf` and `local_fp_buf`), as well as those used when performing tree traversal during data mining (i.e., `local_list` and

```
50 void annealer_thread::Run(){
     ...
55   Rng rng; //store of randomness
     ...
93 }
```

(a) canneal (annealer_thread.cpp)

```
1343 #pragma omp parallel for schedule(dynamic,1)
1344 for(sequence=upperbound - 1; sequence>=lowerbound;
         sequence--)
       ...
1349   int thread = omp_get_thread_num();
       ...
1351   memory *local_fp_tree_buf = fp_tree_buf[thread];
1352   memory *local_fp_buf = fp_buf[thread];
1353   stack *local_list = list[thread];
1354   int *local_ITlen = ITlen[thread];
       ...
1424 }
```

(b) freqmine (fp_tree.cpp)

```
41 class Rng : public Duplicable {
     public:
      Rng duplicate () override {
        auto newCopy = new Rng(time(NULL));
        return newCopy;
      }
      ...
   }
```

(c) Programming-language extension used in *canneal*

Fig. 5: Data-structure duplication.

`local_ITlen`). For the memory-management structures, the read-after write dependence occurs due to keeping track of free memory objects, which is obviously unnecessary in support structures such as these. The other two objects have read-after-write dependences due to increasing and decreasing the pointer to the top of the stack. This dependence must exist within a thread, but not across threads, which perform independent traversals of a tree, beginning from the same starting point.

*Programming-language extension:* We create a new interface understood by compilers called `Duplicable`. Objects that implement this interface must implement `duplicate()`, which performs a class-specific deep copy of an object. The modified code for `canneal` is shown in figure 5(c).

Compilers can use the static type of objects to check which implement the `Duplicate` interface. Then, a compiler can duplicate objects (one per thread) by injecting code at compile-time to invoke the `duplicate()` method at run-time whenever it is safe to do so. Compilers can prove this transformation is safe by analyzing the PDG at compile time: an object used within a loop body that is always written in a given iteration before it is read within the same iteration can be safely duplicated across the generated threads. This test is sufficient to recognize this opportunity for all benchmarks considered by this paper.

```
127 do {
128   val = Get();
129 } while(!atomic_cmpset_ptr((ATOMIC_TYPE *)&p,
          (ATOMIC_TYPE)val, (ATOMIC_TYPE)x));
```

(a) canneal (AtomicPtr.h)

```
485 pthread_mutex_t *ht_lock = hashtable_getlock(cache,
       (void *)(chunk->sha1));
486 pthread_mutex_lock(ht_lock);
    ...
498 if (hashtable_insert(cache, (void *)(chunk->sha1),
       (void *)chunk) == 0) {
      ...
500 }
    ...
507 pthread_mutex_unlock(ht_lock);
```

(b) dedup (encoder.c)

```
732 pthread_mutex_lock(&mutex[index]
       [ipar % MUTEXES_PER_CELL]);
733 cell->density[ipar % PARTICLES_PER_CELL] += tc;
734 pthread_mutex_unlock(&mutex[index]
       [ipar % MUTEXES_PER_CELL]);
```

(c) fluidanimate (pthreads.cpp)

```
1257 #pragma omp critical
1258 {
1259   if (current < released_pos) {
1260     released_pos = current;
1261     fp_node_sub_buf->freebuf(MR_nodes[current],
           MC_nodes[current], MB_nodes[current]);
1262   }
1263 }
```

(d) freqmine (fp_tree.cpp)

```
1257 commutative
1258 {
1259   if (current < released_pos) {
1260     released_pos = current;
1261     fp_node_sub_buf->freebuf(MR_nodes[current],
           MC_nodes[current], MB_nodes[current]);
1262   }
1263 }
```

(e) Programming-language extension used in *freqmine* [25]

Fig. 6: Commutative dependences.

*4) Commutative Dependences: canneal, dedup, fluidanimate, freqmine*

Programs often contain dependent operations that can be applied in any order while preserving program semantics. In other words, the operations are commutative and their relative execution order is not important.

Figure 6 shows the commutative dependences semantic-gap components we found in *canneal*, *dedup*, *fluidanimate*, and *freqmine*. In each, critical sections are defined that allow only one thread at a time to complete the enclosed operation. This

provides atomic update of state, to avoid data corruption, but commutativity of the operations since there is no ordering defined on thread access to the critical sections.

In *canneal* (figure 6(a)) two elements are swapped when profitable using an atomic operation to avoid data corruption. The algorithm is designed to allow multiple threads to swap the same elements multiple times. *Dedup* inserts chunks of data to compress into a global hash table (figure 6(b)), where ordering within the hash table is unimportant. In *fluidanimate*, in figure 6(c), particles that are close enough in the simulated 3D space influence each other's position and velocity, which is a symmetric operation, making the order in which they are accessed unimportant. Finally, in *freqmine*, after performing mining for each item, the algorithm frees auxiliary data structures (figure 6(d)), also able to be performed in any order.

The semantic-gap component of reassociation is similar to commutativity, but differs in that it applies to a particular instance of a variable, whereas the commutative-dependences semantic-gap component applies to a region of code that can be executed in any order.

*Programming-language extension:* This is closed by the attribute `commutative` attached to single-entry-single-exit code regions (SESE) [26] (e.g., functions, loops) [25]. Compilers can use this information in the same way that has been previously proposed [25] but generalizing its applicability to all SESE code regions rather than only functions. For example, figure 6(e) shows the modified code of *freqmine* using this programming language extension to close its semantic gap.

### C. Compiler Extension

The programming language extensions described alongside each of the semantic-gap components are used by parallelizing compilers either to remove or to annotate dependences in the program's PDG. Figure 7 shows the algorithm to implement in a parallelizing compiler to modify a PDG. The function `modifyCodeAndGeneratePDG` is invoked when a parallelizing compiler's IR is generated and before the SCCs of the PDG are computed. This function takes the whole program IR and it modifies the code to generate a PDG without the dependences that can be removed and with the annotations to the remaining dependences.

For example, let us consider the loop-carried data dependence from `dedup` created by reading the input buffer sequentially across loop iterations (figure 3(b)). This dependence has distance 1 (i.e., the dependence is between adjacent loop iterations), which sequentializes all instances of the related SCC. When this dependence is considered by the algorithm, it checks whether it can be removed (line 4). Because this dependence cannot be removed, the algorithm tries to annotate this dependence by altering the code (line 6). Given the algorithmic-option extension (figure 3(c)), the compiler is free to choose an alternative chunk size, increasing the dependence distance, and enabling some iterations of the `dedup` loop to be run in parallel. This is a degree of freedom that the compiler now controls and it is used to generate the same TLP as developers have described manually in *dedup*.

```
1 PDG modifyCodeAndGeneratePDG(Code program)
2   pdg = generatePDG()
3   for (auto dep : pdg->edges()){
4     if (alterCodeToRemoveDep(program, pdg, dep))
5       continue
6     annotateDependence(program, pdg, dep)
7   }
8   return pdg
```

Fig. 7: Algorithm to modify the PDG by using the proposed programming-language extensions.

## IV. Limit Study

Next we empirically evaluate the performance implications of the four semantic-gap components.

### A. Experimental Setup

*Platform:* Oracle-extractor and parallelism-measurer are built on LLVM 3.7.1 [27]. Our evaluations were performed using a dual socket Dell PowerEdge R730 server with two Intel Xeon E5-2695 v3 Haswell processors running at 2.3GHz and capable of 9.60GT/s on the QPI interface. Each processor has 14 cores with 2-way hyperthreading, and a 35MB of last-level cache. The cores are supported by 256GB of main memory in 16 dual rank RDIMMs at 2133MHz. Finally, the OS is Red Hat Enterprise Linux Server 6.7.

*Benchmarks:* We evaluated the applications included in the latest PARSEC benchmark suite, version 3.0, representing a wide range of modern workloads. We chose PARSEC because each benchmark has been implemented twice: sequentially and manually parallelized, targeting multicore architectures. Unfortunately, we could not consider *vips* and *bodytrack* because they did not compile using the vanilla clang compiler and the binary generated by clang for *ferret* produced incorrect outputs. Moreover, we could not generate a single whole-program bitcode file for *facesim*, *raytracer*, or *x264*; therefore, we could not analyze them with our tools. Unless otherwise stated, simlarge inputs were used, although section IV-D shows that these are big enough for limit studies.

*Measuring the impact of a semantic-gap component:* We model the impact of programming language extensions to measure the importance of each semantic-gap component by manually identifying the SCCs included in minDDG (section II) that have been affected by it. To do so, we compare SCCs of minDDG with the manually parallelized code (e.g., its critical sections) then tag them with the list of semantic-gap components used for their counterparts in the manually parallelized code. Parallelism-measurer parses these SCC tags and emulates the parallelism model we found in the manually parallelized version of the benchmark. For example, consider *freqmine*, shown in figure 6(d). The minDDG generated by oracle-extractor contains an SCC between the code in lines 1259–1262. The code generated by any parallelizing compiler will preserve the execution order of the identified SCC, but the manually parallelized version of the benchmark does not.

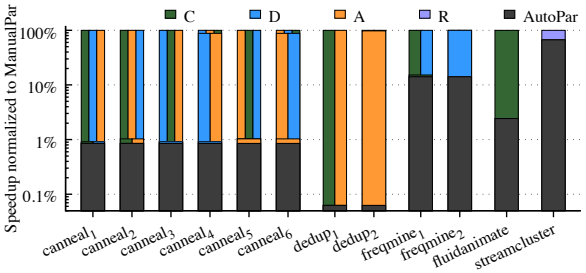



Fig. 8: Breakdown of extracted parallelism when closing semantic-gap components. Horizontal separation of a bar indicates the increasingly closed components. Vertical separation indicates components closed together. Abbreviations are C: Commutative dependences, D: Data duplication, A: Algorithmic options, and R: Reassociation, also used in following figures.

Therefore we tagged this SCC as a commutativity semantic-gap component. Parallelism-measurer parses this tag and emulates the parallel execution of the related code without preserving the execution order between the invocations of this tagged SCC. We use this solution to turn on and off each semantic-gap component.

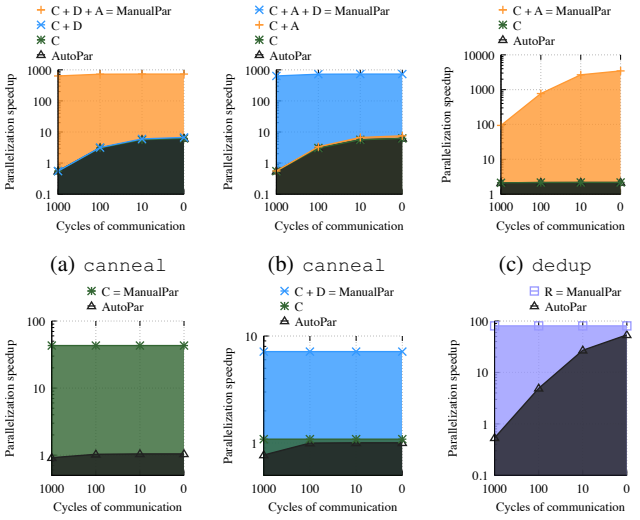
*Permutations of semantic-gap components:* We need to consider all possible permutations of semantic-gap components to understand the importance of a given component. This is because the impact of closing a given semantic-gap component with a programming-language extension depends on the baseline code we start from (i.e., which other semantic-gap components have been closed already before the current one). Our empirical evaluations found that this is important in most of the benchmarks we studied.

### B. Semantic-Gap Breakdown

Most parallelism expressed by developers is out of reach for compilers. Figure 8 shows the obtained speedups relative to the manually parallelized version of the code for whole-program execution, but only running loops parallelized manually by developers in parallel. Closing the algorithmic options component (i.e., A) is necessary for *dedup*: not closing it will block the adoption of parallelizing compilers for this workload. Similarly, closing the data structure duplication component (i.e., D) is necessary for *canneal*. Notice that for this benchmark closing A alone is not enough. Furthermore, closing the reassociation component (i.e., R) makes automatic parallelization of *streamcluster* possible even if a need for it is not as pronounced as for the other components. Finally, notice that for *canneal* A and D work well together, but show little impact when closed alone.

### C. Sensitivity Analysis

Multicores are used in different domains, from low-end (mobiles and laptops with 8–16 cores) to high-end (HPC with thousands of cores across nodes) domains. Moreover, different architectures (Intel, ARM, IBM Power chips with




(a) `canneal` (b) `canneal` (c) `dedup`

(d) `fluidanimate` (e) `freqmine` (f) `streamcluster`

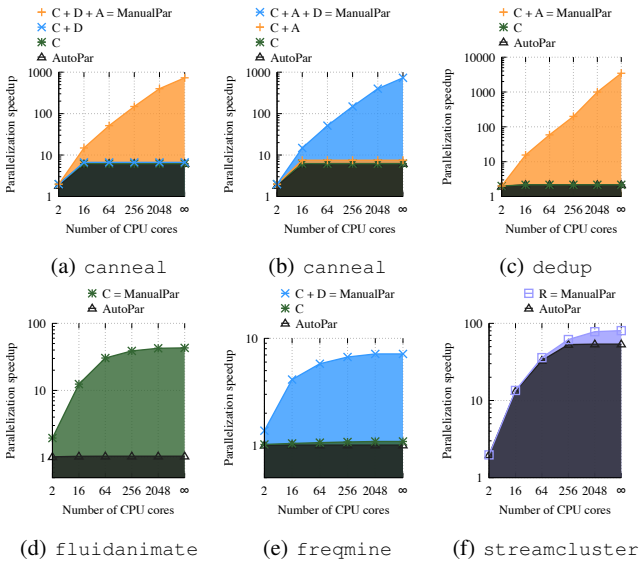
Fig. 9: Sensitivity analysis for the inter-core latency.

or without scalar operand networks [28]) and system configurations (single-node single-socket, single-node multi-socket, multi-node multi-socket) have significantly different latencies to communicate between cores. To study how the impact of the semantic gap changes through this heterogeneous computing spectrum, we extend our analysis, sweeping the number of cores and inter-core latencies.

*Inter-core latency:* To understand how the impact of the semantic gap changes with the inter-core latency, we performed a sensitivity analysis while keeping an unbounded number of cores. Figure 9 shows this analysis. We considered four situations with related inter-core latencies: 1,000 cycles for emulating inter-server parallelization, 100 cycles for multi-core parallelization within the same commodity CPU, 10 cycles for multi-core parallelization in specialized architectures with scalar operand networks (e.g., TRIPS [29], HELIX-RC [1]), and 0 cycles to measure the limits.

Some benchmarks (e.g., *canneal* and *streamcluster*) lose all of the automatic parallelization benefits when the inter-core latency is 1,000 cycles. However, they tolerate high latencies when the whole semantic gap is closed. Closing the semantic gap reduces inter-core communication, making it so infrequent that these benchmarks become latency insensitive. The only benchmark that remains sensitive to latency when the semantic gap is closed is *dedup*, where threads require frequent communication even when manually parallelized.

Closing the semantic gap becomes even more essential when today's inter-core latencies are assumed. *Streamcluster* is the empirical evidence of this. For this benchmark, AutoPar is close to ManualPar when 0 latency is assumed (see figures 8 and 9(f)). However, when the latency increases, AutoPar degrades dramatically leading to single digit speedups for today's commodity multicore latencies.




(a) `canneal` (b) `canneal` (c) `dedup`

(d) `fluidanimate` (e) `freqmine` (f) `streamcluster`

Fig. 10: Sensitivity analysis for the core count.

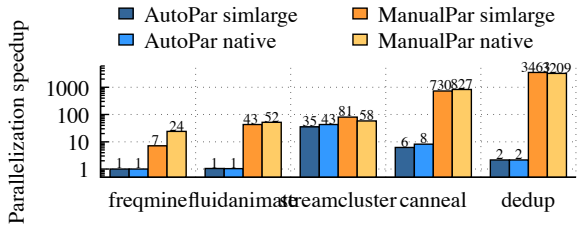



Fig. 11: Parallelism using different sizes of input, comparing large-scale simulation and realistic application input.

*Number of cores:* Figure 10 shows how the semantic gap changes with the number of cores. We considered six situations: 2 cores to model mobile phones, 16 cores for single socket servers, 64 cores to model four socket servers, 256 and 2,048 cores for small and large clusters, and an infinite number of cores to measure the parallelization limits. The semantic gap constrains parallelizing compilers to target only low-end devices (e.g., a couple of cores) for most benchmarks. This highlights the importance of closing the semantic gap to go beyond a few cores for automatic parallelization. Benchmarks *canneal*, *fluidanimate*, *dedup*, and *freqmine* highlight this constraint. On the other hand, *streamcluster* shows important scalability even without closing the semantic gap. However, as previously mentioned (see figure 9), this benchmark still requires the semantic gap to be closed to handle typical inter-core latencies.

### D. Altering Input Size

To ensure that the parallelism analyses shown in this paper are independent with the input size, we computed the performance limits of our benchmarks for much larger inputs than those used in previous sections. (Native inputs require between 130× and 3,000× more instructions than simlarge.) Figure 11 shows that the performance limits of both AutoPar and ManualPar remain almost identical when the input size increases significantly.

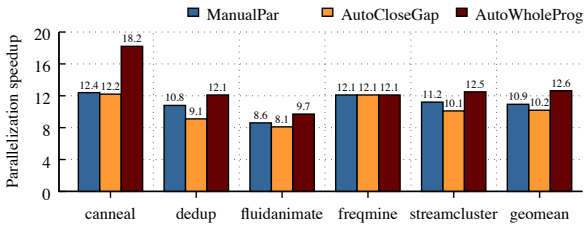



Fig. 12: Performance obtained by closing the semantic gap in PARSEC on a 28-core machine.

## V. Real-System Experiments

This section evaluates the performance obtained when using the programming-language extensions described in section III, running on a real multicore machine, described in section IV-A.

*1) The extended HELIX parallelizing compiler:* We extended the HELIX parallelization [3] to measure the impact of the programming language extensions from section III to parallelizing compilers. First, we replaced the program dependence analysis with the use of minDDG (only true data dependences, see section II-B). This was necessary because applying a parallelization technique automatically requires closing the analysis gap (section II-B), which is outside the scope of this paper. Second, we rely on the algorithm in figure 7 to modify the PDG generated from minDDG.

We applied the extended HELIX to the original codebase modified using the programming language extensions shown in section III. The extended HELIX parallelization generates OpenMP code, which is then compiled with the original clang (version 7) compiler.

*2) PARSEC benchmarks:* We applied the extended HELIX to the modified PARSEC benchmarks to generate two parallel binaries. The first one is generated by constraining the parallelizing compiler to consider parallelizing only the loops that have been manually parallelized by the PARSEC developers. We call it *AutoCloseGap*. We use the AutoCloseGap binary to compare the parallelism generated by parallelizing compilers and the one defined by developers. The second binary, instead, is generated by allowing the parallelizing compiler to consider parallelizing all loops of the original codebase. We call it *AutoWholeProg*. Here we used the programming-language extensions only for the loops that the developers had already parallelized (as in AutoCloseGap), but then additionally allowed our parallelizing compiler to parallelize other loops (but without use of programming-language extensions). This binary highlights the true value of enabling a parallelizing compiler to define parallelism instead of requiring a developer to do it: parallelizing compilers can parallelize all loops if

beneficial, developers cannot do it due to strict development-time constraints.

Parallelizing compilers can generate an equivalent amount of parallelism that developers define manually today. Figure 12 shows that the speedups obtained by AutoCloseGap and ManualPar are similar, almost identical. This is because closing the semantic gap allowed the automatic generation of the parallelism defined by developers. The differences come from the way parallelism is extracted. For example, for *streamcluster*, AutoCloseGap obtains a speedup of 10.1× compared to 11.2× for ManualPar. In every loop invocation in AutoCloseGap, the code wakes up the threads used in the OpenMP parallel loop, which requires a kernel call. However, in ManualPar, threads are always awake, so this overhead does not exist. In fact, this overhead is not inherent in the techniques we have described for closing the semantic gap, just in the way we have transformed the serial code to extract parallelism for AutoCloseGap. In particular, it is due to the choice of generating OpenMP code rather than generating threads that stay awake. The OpenMP runtime does not keep threads awake between parallelized loops and, therefore, AutoCloseGap pays extra overhead.

Closing the gap for parallelizing compilers enables the generation of more scalable parallel code than the one defined manually by developers. Figure 12 shows that the speedups obtained by AutoWholeProg (12.6× average) are significantly higher than ManualPar (10.9×). There are two reasons for this. First, ManualPar includes several loops that are kept sequential; AutoWholeProg does not. Second, ManualPar includes many loops that are executed sequentially within a single thread reducing the amount of nested parallelism; AutoWholeProg does not.

*3) System tools codebase:* This paper identified semantic-gap components only in five benchmarks, but these findings generalize across a wide range of applications. To demonstrate this, we applied the extended HELIX to the codebase of several frequently used system tools from the Linux/Unix environment, yielding an additional 16 workloads. These codebases have been sequentially designed and developed decades ago. They run sequentially despite their daily use in most systems and despite us having been in the multicore era for more than a decade. This suggests that it is unlikely that system tools will be redesigned only to use the many cores we now have in a single chip. Rather, it is more realistic to add only a few lines of code to the original codebase and then rely on an automatic parallelizing compiler to extract the necessary TLP. This is what we have done. We have modified recent stable versions of the programs shown in figure 13 to use the programming language extensions from section III. Finally, we applied the extended HELIX to generate their parallel binaries. We chose custom inputs for each application to ensure an execution time of at least five minutes for the sequential baseline.

Programming-language extensions of section III are necessary to enable parallelizing compilers to extract significant TLP from many system tools most of us use daily. Figure 13 shows the performance obtained on our platform by the parallel binaries generated with (AutoWholeProg, 7.1×) and without (AutoPar, 1.2×) the programming-language extensions. These language extensions are necessary to parallelize the considered programs, and each extension was used at least once across all system tools. We only added a few tens of lines of code for each of these programs, which include those containing several thousand to over sixty thousand lines of code in their original codebase (maximum 60,384 in *gawk* — GNU AWK version 4.2.1), typically taking around 30 minutes for each application.

Although we used all programming-language extensions across the suite of tools, they were not all equally useful for each workload. For example, closing the algorithmic options semantic-gap component yielded 71% of the total performance improvement in *sort*, but could not be applied to *cat*, *cp*, *grep*, or *sum*. Closing the data duplication semantic-gap component resulted in 90% of the performance improvements for *grep*, the final 10% coming from identifying commutative dependences. For this application, only two semantic-gap components needed to be closed, whereas for programs such as *chmod*, *chgrp*, and *chown*, all four semantic-gap components needed to be closed to achieve maximum performance.

## VI. Related Work

From the beginning of the multi-core era, the research community has discussed the role of compilers for TLP extraction, a debate still ongoing [30]. This paper studied the limits of compilers to highlight the semantic gap between what developers exploit to parallelize their code manually and what can be encoded and extracted from sequential codebases from compilers. To the best of our knowledge, only the commutative-dependences semantic-gap component has been exploited by existing work in the literature [25], [31], [32].

Multi-threaded programming is still a great challenge for most developers. Although being widely used, Pthread [33] makes programmers spend significant effort on organizing the work-flow of threads. To ease the burden of accurate thread configuration, models, such as OpenMP [34], MPI [35], TBB [36], Cilk [37], Cilk++ [38], have been proposed. Some emphasize synchronization and others loop-level task construction. Other approaches like CUDA [39] and OpenCL [40] emphasize novel hardware interfaces, whereas Prabhu et al. [32] propose language constructs for speculation. However, none make parallel programming as simple as serial programming.

Automatic TLP extraction from sequential programs has a rich history. Some techniques distribute the dynamic invocations of a loop dependence SCC across parallel processing units (e.g., cores, servers) [1], [3], [4], [20], [41]–[47]. Other techniques keep all invocations of a single SCC within the same processing unit and distribute different SCCs across units (e.g., cores) [5], [6], [8], [18], [48], [49]. In all of these approaches, parallelism is limited by the SCCs extracted from the dependence graph. This paper studies the limits of an SCC-based approach to highlight the limitations that

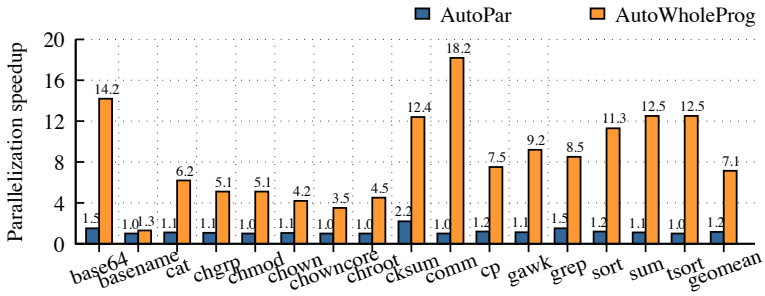



Fig. 13: Performance obtained by closing the semantic gap in systems tools on a 28-core machine.

compilers have with the current expressiveness available in today's programming languages, like C++.

Some of the limitations of automatically extracting TLP from sequential code, created by lack of expressiveness in programming languages, have been recently studied, in particular the impact of the commutativity component [25], [50], [51]. This paper studies all components of the semantic gap we found in modern workloads. Others considered satisfying dependences in alternative ways [13], [52]. These latest approaches, however, focus only on specific classes of workloads, allowing them to use domain-specific knowledge. This paper aims for a broader set of workloads and, therefore, it does not make this assumption and it does not rely on domain-specific knowledge.

Finally, several studies have been performed to identify (manual or automatic) parallelization opportunities [31], [53]–[58] or to measure the limits of their parallelism [17], [59]–[63]. None of these studies, however, studies why dependences exist and they did not compare the detected parallelism with that exposed by developers in manually parallelized versions of the considered workloads. This paper is the first one to perform this study.

## VII. Conclusions

Using a novel SCC analysis, based on measuring the amount of coarse-grained parallelism that is extractable from a sequential execution, we have identified four techniques that developers use to break data dependences when manually parallelizing their applications, using semantic knowledge. We take advantage of previously proposed, simple programming language extensions to convey this information to a compiler, allowing automatic parallelization to surpass the performance of manual parallelization. Our work shines a light on the limitations of today's parallelizing compilers and points firmly towards further research into programming language design to capture program semantics for the benefit of tomorrow's tools.

## Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/P020011/1. This work was also made possible by support from the National Science Foundation via the awards CCF-1908488, CCF-2107042, CCF-2118708, CNS-1763743, PPoSS-2119069, and PPoSS-2028851. Additional data related to this publication is available in the data repository at https://doi.org/10.17863/CAM.76224.

## References

[1] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, "Helix-rc: an architecture-compiler co-design for automatic parallelization of irregular programs," in *ISCA*, 2014.

[2] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "Helix-up: Relaxing program semantics to unleash parallelization," in *CGO*, 2015.

[3] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "Helix: automatic parallelization of irregular programs for chip multiprocessing," in *CGO*, 2012.

[4] S. Campanoni, T. M. Jones, G. Holloway, G.-Y. Wei, and D. Brooks, "Helix: Making the extraction of thread-level parallelism mainstream," *IEEE Micro*, vol. 32, no. 4, pp. 8–18, 2012.

[5] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *MICRO*, 2005.

[6] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August, "Decoupled software pipelining creates parallelization opportunities," in *CGO*, 2010.

[7] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke, "Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 166–176, 2009.

[8] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *CGO*, 2008.

[9] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 177–187, 2009.

[10] D. Liu, Z. Shao, M. Wang, M. Guo, and J. Xue, "Optimal loop parallelization for maximizing iteration-level parallelism," in *CASES*, 2009.

[11] A. Matni, E. A. Deiana, Y. Su, L. Gross, S. Ghosh, S. Apostolakis, Z. Xu, Z. Tan, I. Chaturvedi, D. I. August, and S. Campanoni, "NOELLE Offers Empowering LLvm Extensions," 2021.

[12] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "Perspective: A sensible approach to speculative automatic parallelization," in *ASPLOS*, 2020.

[13] E. A. Deiana, V. St-Amour, P. A. Dinda, N. Hardavellas, and S. Campanoni, "Unconventional parallelization of nondeterministic applications," in *ASPLOS*, 2018.

[14] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, "Parallel-stage decoupled software pipelining," in *CGO*, 2008.

[15] T. Oh, S. R. Beard, N. P. Johnson, S. Popovych, and D. I. August, "A generalized framework for automatic scripting language parallelization," in *PACT*, 2017.

[16] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "Posh: a tls compiler that exploits program structure," in *PPoPP*, 2006.

[17] E. Fatehi and P. Gratz, "Ilp and tlp in shared memory applications: A limit study," in *PACT*, 2014.

[18] A. Aiken and A. Nicolau, "Perfect pipelining: A new loop parallelization technique," in *ESOP*, 1988.

[19] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM TOPLAS*, vol. 9, no. 3, pp. 319–349, 1987.

[20] A. R. Hurson, J. T. Lim, K. Kavi, and K. Lee, "Parallelization of doall and doacross loops-a survey," *Advances in Computers*, vol. 45, pp. 53–103, 12 1997.

[21] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August, "Practical and accurate low-level pointer analysis," in *CGO*, 2005.

[22] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers, "A limit study of javascript parallelism," in *IISWC*, 2010.

[23] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[24] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, 1987.

[25] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *MICRO*, 2007.

[26] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *PLDI*, 1994.

[27] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[28] M. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal, "Scalar operand networks," *IEEE Transactions on Parallel and Distributed Systems*, 2005.

[29] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore, "TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP," in *ACM TACO*, 2004.

[30] D. August, K. Pingali, D. Chiou, R. Sendag, J. Y. Joshua *et al.*, "Programming multicores: Do applications programmers need to write explicitly parallel programs?" *IEEE Micro*, vol. 30, no. 3, pp. 19–33, 2010.

[31] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, "How much parallelism is there in irregular applications?" in *PPoPP*, 2009.

[32] P. Prabhu, G. Ramalingam, and K. Vaswani, "Safe programmable speculative parallelism," in *PPoPP*, 2010.

[33] IEEE and T. O. Group, *IEEE Std 1003.1-2017, The Open Group Base Specifications Issue 7*, 2018.

[34] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[35] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 1998.

[36] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*, 2007.

[37] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[38] C. E. Leiserson, "The cilk++ concurrency platform," in *DAC*, 2009.

[39] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," in *SIGGRAPH*, 2008.

[40] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[41] C.-Z. Xu and V. Chaudhary, "Time stamp algorithms for runtime parallelization of doacross loops with dynamic dependences," in *TPDS*, 2001.

[42] D.-K. Chen and P.-C. Yew, "On effective execution of nonuniform doacross loops," in *TPDS*, 1996.

[43] ——, "Redundant synchronization elimination for doacross loops," in *TPDS*, 1999.

[44] K. Ebcioglu and A. Nicolau, "A global resource-constrained parallelization technique," in *ICS*, 1989.

[45] K. S. McKinley, "Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors," in *ICS*, 1994.

[46] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August, "Automatic speculative doall for clusters," in *CGO*, 2012.

[47] S. Campanoni, T. Jones, G. Holloway, G. Y. Wei, and D. Brooks, "The helix project: Overview and directions," in *DAC*, 2012.

[48] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative parallelization using software multi-threaded transactions," in *ASPLOS*, 2010.

[49] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *PACT*, 2007.

[50] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *PLDI*, 2011.

[51] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *PLDI*, 2007.

[52] Z. Zhao, B. Wu, and X. Shen, "Challenging the "embarrassingly sequential": Parallelizing finite state machine-based computations through principled speculation," *SIGPLAN Not.*, vol. 49, no. 4, pp. 543–558, 2014.

[53] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1088–1098, 1988.

[54] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *CGO*, 2009.

[55] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: rethinking and rebooting gprof for the multicore age," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 458–469, 2011.

[56] G. Tournavitis and B. Franke, "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information," in *PACT*, 2010.

[57] W. Thies, V. Chandrasekhar, and S. Amarasinghe, "A practical approach to exploiting coarse-grained pipeline parallelism in c programs," in *Micro*, 2007.

[58] R. Eigenmann, J. Hoeflinger, and D. Padua, "On the automatic parallelization of the perfect benchmarks (r)," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 1, pp. 5–23, 1998.

[59] M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge, "The limits of instruction level parallelism in spec95 applications," *SIGARCH Comput. Archit. News*, vol. 27, no. 1, pp. 31–34, 1999.

[60] D. W. Wall, "Limits of instruction-level parallelism," in *ASPLOS*, 1991.

[61] K. B. Theobald, G. R. Gao, and L. J. Hendren, "On the limits of program parallelism and its smoothability," *SIGMICRO Newsl.*, vol. 23, no. 1-2, pp. 10–19, 1992.

[62] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, and D. Connors, "Chip multi-processor scalability for single-threaded applications," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 44–53, 2005.

[63] N. Murphy, T. Jones, R. Mullins, and S. Campanoni, "Performance implications of transient loop-carried data dependences in automatically parallelized loops," in *CC*, 2016.