Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture

Farhana Aleen and Nathan Clark

College of Computing
Georgia Institute of Technology
{farah, ntclark}@cc.gatech.edu

Abstract

Extracting performance from many-core architectures requires software engineers to create multi-threaded applications, which significantly complicates the already daunting task of software development. One solution to this problem is automatic compile-time parallelization, which can ease the burden on software developers in many situations. Clearly, automatic parallelization in its present form is not suitable for many application domains and new compiler analyses are needed address its shortcomings.

In this paper, we present one such analysis: a new approach for detecting *commutative functions*. Commutative functions are sections of code that can be executed in any order without affecting the outcome of the application, e.g., inserting elements into a set. Previous research on this topic had one significant limitation, in that the results of a commutative functions must produce identical memory layouts. This prevented previous techniques from detecting functions like `malloc`, which may return different pointers depending on the order in which it is called, but these differing results do not affect the overall output of the application. Our new commutativity analysis correctly identify these situations to better facilitate automatic parallelization. We demonstrate that this analysis can automatically extract significant amounts of parallelism from many applications, and where it is ineffective it can provide software developers a useful list of functions that *may be* commutative provided semantic program changes that are not automatable.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; D.1.3 [*Programming Techniques*]: Parallel Programming

General Terms Languages, Algorithms

Keywords Automatic Software Parallelization, Random Interpretation, Commutative Functions

1. Introduction

Increasing processor performance has always been the main driver of the computing industry. Processors that can provide more computation capability enable new applications; these applications spur consumers to purchase new computers, which feeds the industry in a positive feedback loop [9]. As computing evolves toward manycore architectures, the onus for improving application performance is increasingly shifting from processor designers to application developers. The application developer can no longer ignore the architecture they are targeting, and this significantly increases the difficulty of creating applications that make effective use of the performance provided by modern architectures.

Automatic compiler-based parallelization is one technique which can lessen the burden on application developers. In this scenario, the application developer creates code using sequential languages and the compiler attempts to identify concurrency automatically. This enables the application developer to effectively utilize manycore architectures, provided the compiler can parallelize the application well.

Because of its potential benefits, compiler-based parallelization is a well studied field of computer science [1, 3, 6, 13, 15, 20]. Unfortunately, the general consensus is that, while effective for certain domains of applications, compiler parallelization is generally not useful for most applications.

In contrast to this popular opinion, several recent papers [2, 18, 25] have shown that recent advances in compiler analysis and hardware support for speculation have enabled compiler-based parallelization to be surprisingly effective in domains conventionally thought beyond its reach. For example, Bridges et al. [2] report 454% average speedup over single threaded code on the SPECint2000 benchmarks, using a mixture of automatic parallelization with small code annotations.

One of the key enablers in Bridges' reported speedup was the *manual* identification of *commutative functions*. Commutative functions are sections of code that can be executed in any order relative to itself without affecting the outcome of the application. This means commutative functions can be parallelized without affecting program semantics. For ex-

```
class hash_set {
  private:
    vector<linked_list> set;
  public:
    void insert(int x) {
      unsigned bucket = x % set.size();
      set[bucket].insert(x);
    }
    bool is_member(int x) {
      return set[x % set.size()].is_member(x);
    }
    void remove(int x) {
      set[x % set.size()].remove(x);
    }
};
```
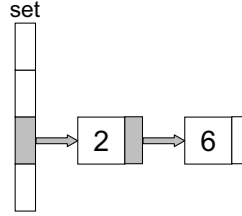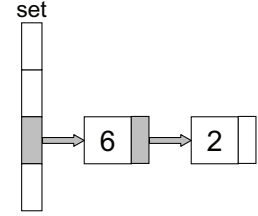
```
test_set.insert(2);
test_set.insert(6);
```

```
test_set.insert(6);
test_set.insert(2);
```

(a) (b) (c)

Figure 1. (a) Example class with commutative function `insert()`. (b) and (c) show executing the function with two different inputs creates different memory layouts, however the function is still commutative because the only functions that read the memory layout (`remove()` and `is_member()`) both produce the same results using both layouts.

ample, Figure 1 shows a simple data structure for a hash-set implemented as a vector of linked-lists. A human can easily recognize that insertions into the hash set can be performed in any order without affecting the program output. However there is no known compiler analysis that can detect this, even with perfect memory disambiguation. This is because executing the `insert()`s in different orders will result in different internal data structures, shown in Figure 1 (b) and (c). If the hash set is size four, and two elements are inserted into the same bucket, then the different insertion orders will create a different linked lists. The compiler recognizes that the semantics of the sequential code have been violated, and serializes these function calls.

This paper presents a new and fully automatic method for testing if functions are commutative. We make the observation that *the results of a function only matter if they are written to or affect I/O*. Thus, it does not matter that the output of a function is different given two different execution orders; it only matters that the functions that read those outputs produce the correct results. Using the example in Figure 1 again, it is irrelevant that the linked list created when calling `insert()` in different orders produces two different memory layouts, because that information is state that is internal to the program. What does matter is that the only functions which use that internal state, `is_member()` and `remove()`, produce the same results no matter what the linked list looks like internally.

In order to automatically detect commutative functions, a candidate function is symbolically executed in two different orders to create an abstract representation of the result of the two execution orders. This symbolic result is then used as input to all functions that could potentially read the results, and those functions are symbolically executed. If the outputs of these reader functions are identical, then the initial function is commutative. This process can recursively test functions (first readers are symbolically executed, then the

readers of the readers, etc.) until the initial function is found to be commutative or the "internal" state affects I/O.

The main challenge of this technique is developing an efficient method for symbolic execution of the functions. Symbolically executing functions is known to have significant scalability issues, and so we introduce a new application of *random interpretation* [7]. Random interpretation is a randomized algorithm that can probabilistically prove program assertions with very low computational complexity. We demonstrate how commutativity testing with random interpretation is simple enough that it can be rapidly performed even on functions that write to heap variables, thus having many potential readers.

Using random interpretation for identifying commutative functions is both computationally simple, and very effective at finding commutative functions. This analysis can lead to significant increases in application parallelism with no additional programmer effort; and when the analysis does not automatically find parallelism, the compiler can present a set of simple, potential semantic changes to enable the compiler to automatically parallelize the code. This compiler/programmer co-development is the simplest way to achieve effective utilization of future architectures.

There are two primary contributions of this work. First, it develops a new method for testing the commutativity of functions. Assuming the compiler has full visibility of the application, this method can efficiently detect many functions that can be reordered without violating application semantics. Even without full visibility, the compiler can find significant parallelism and suggest places where commutativity may exist, pending analysis of external functions. Second, it presents a novel use of random interpretation. While the commutativity detection does *not* rely on using random interpretation, we demonstrate that this is a very efficient way to prove commutativity. We demonstrate that commutativity is an efficient technique that can automatically un-

cover significant amounts of parallelism in several popular benchmarks.

2. Overview of the Approach

Identifying coarse grained parallelism in applications is a critical problem in modern computing. Several researchers have noted that in many applications commutative functions are an excellent source of such parallelism [2, 16]. Some examples of common commutative functions that compilers are presently not able to identify are custom memory allocators (such as `xalloc()` in 197.parser) and pseudo-random number generators (such as `Yacm_random()` in 300.twolf). Bridges et al. showed that manually breaking these dependencies can result in significant application speedup on manycore architectures [2]. This paper presents an algorithm that *automatically* identifies such commutative functions, including these examples.

Previous proposals for automatically detecting commutative functions [16] only tested if memory contents would exactly match if the candidate function was executed in any order. However this test is too conservative to detect situations such as the `insert()` example in Figure 1. In this example, the memory contents do not match, but the resulting program would still be correct if the `insert()`s were reordered. The new commutativity test proposed by this paper is less conservative, checking only if the outputs of a program would be correct if the candidate function was executed in an arbitrary order with respect to itself.

Figure 2 shows the pseudo-code for detecting if a function is commutative. As a first step, two input sets, I_1 and I_2, and a random initial memory state, M, are generated for the candidate function. This function is then interpreted, i.e., abstractly executed, using the random inputs in two different orders: first execute using I_1 before I_2, and then use I_2 before I_1. These two execution orders will modify the initial memory state, M, and create two new memory states[1], $M_{1,2}$ and $M_{2,1}$. Note that these memory states contain not only the function outputs, but also all memory written by the function, such as heap and global variables. If $M_{1,2}$ and $M_{2,1}$ are equivalent then the function is commutative. This test is equivalent to the methods previously used for automatically detecting commutativity [16].

However, if those two memory states are not equivalent then the function may still be commutative, provided that all functions that read from $M_{1,2}$ and $M_{2,1}$ produce the same results. The idea is that as long as the different results created by changing execution order do not affect the program output, then the compiler is free to reorder the functions, and the only functions that could be affected are the ones

---

[1] Executing a function typically creates register values in addition to a transformed memory space, and this must be accounted for when testing commutativity. We omit this detail from Figure 2 because it makes the notation clearer, and supporting function outputs is a natural extension to the algorithm shown (simply treat the modified registers as an extension of the memory state).

Procedure: Commutativity_test(F)

1 Input: $F(I, M)$, a function which operates on a set of inputs, I, and a memory space, M
2 Generate a random memory space, M, and two random inputs, I_1 and I_2
3 $M_{1,2} = $ Rand_Interp(F, I_2,Rand_Interp(F, I_1, M))
4 $M_{2,1} = $ Rand_Interp(F, I_1,Rand_Interp(F, I_2, M))
5 **if** $M_{1,2} = M_{2,1}$ **then**
6 | F is Commutative
   end
7 **foreach** *function, F', which could read the memory spaces, $M_{1,2}$ or $M_{2,1}$* **do**
8 | **if** *Recursive_test($F', M_{1,2}, M_{2,1}$) = not_comm* **then**
9 | | F is not Commutative
   | **end**
   end
10 F is Commutative

Procedure: Recursive_test(F, M_1, M_2)
11 **if** *F is a system call or out of scope* **then**
12 | return not_comm
   end
13 **if** *F is already visited* **then**
14 | return comm
   end
15 Mark F visited
16 Generate a random input, I, for F
17 $M_1' = $Rand_Interp($F, I, M_1$)
18 $M_2' = $Rand_Interp($F, I, M_2$)
19 **if** $M_1' = M_2'$ **then**
20 | return comm
   end
21 **foreach** *function, F', which could read the memory spaces, M_1' or M_2'* **do**
22 | **if** *Recursive_test(F', M_1', M_2') = not_comm* **then**
23 | | return not_comm
   | **end**
   end
24 return comm

**Figure 2.** Testing commutativity of a function

that read the results of the reordered functions. Each function that could consume these results is interpreted using a random input operating on the two memory states initially produced, creating two new memory states, $M_{1,2}'$ and $M_{2,1}'$. If $M_{1,2}'$ and $M_{2,1}'$ are identical, then the fact $M_{1,2}$ and $M_{2,1}$ did not match is irrelevant, because the function which used those results produces the same output. If $M_{1,2}'$ and $M_{2,1}'$ did not match, then the check can be recursively performed until it is determined that the state produced by reordering the initial function affects program output.

Figure 3 makes this algorithm more concrete using the example from Figure 1 again. Two random values are generated and inserted into the hash table. We execute the two in-
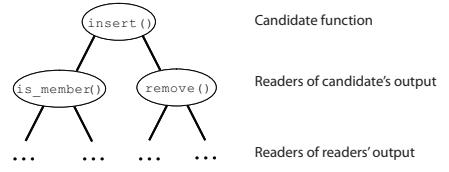
Generate abstract representations
of the results for the two execution orders
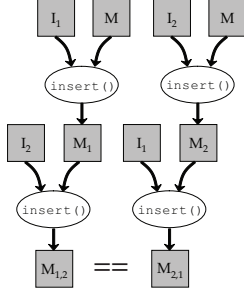
If the results do not match, check
if the readers of those results match
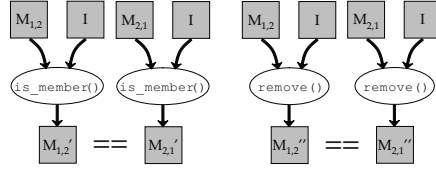
Recurse down the tree of readers until
reaching output or proving commutativity

Candidate function

Readers of candidate's output

Readers of readers' output

Figure 3. Pictorial representation of Figure 2 operating on the example from Figure 1

sertions in different orders and compare the resulting memory space. If the memory spaces are the same then we know with very high probability (quantified in Section 3.1) that `insert()` can be executed in any order without affecting program results. However, if the memory spaces do not match, which would happen if the random inputs hash to the same bucket, then the algorithm executes all the readers of the memory space written by `insert()`. In this case, `remove()` and `is_member()` read from the memory written by `insert()`. These two functions are executed using additional random inputs, and the algorithm checks if those functions produce the same results. They do, and thus we know that `insert()` can be reordered without affecting the program output. If the readers' output did not match, then the algorithm would check the readers of the readers' output recursively, and so on, until commutativity was proven or the results were passed passed beyond the scope of the compiler or used as program output.

This algorithm is general enough to test whether it is legal to reorder arbitrary sections of code, not just commutative functions, however, there are far too many possibilities to test all possible combinations. Identifying which sections of code to test is a significant research challenge that we leave for future work.

Automatically parallelizing commutative functions is a fairly straightforward process. Thread spawns are inserted whenever overlapping function calls are detected. Shared variables, such as the private variable `set` in Figure 1, are either privatized [25] or protected with mutex locks. To be explicitly clear, *this paper is only targeting functions that are executed consecutively* that could be parallelized, such as `f(x); f(y);` or `for(; i < limit; i++) f(i);`. It does not identify instances of parallelization that arise from recursive function calls, e.g., `f(i) { return f(i - 1) + f(i - 2); }`. Parallelizing recursive commutative functions is more challenging, and we refer the reader to Rinard and Diniz' work for a detailed discussion on that process [16].

```
int global_1, global_2;

void a(int* d) {
  int tmp = global_2 % 2;
  if(tmp == 0) {
    (*d) = global_1;
  } else {
    (*d) = -(global_1 + 7 + tmp);
  }
  global_1++;
}

int b(int e, int f) {
  return e - f;
}

int c(int g) {
  if(g < 0)
    return -g;
  return g;
}

int main() {
  int foo, bar;
  a(&foo);
  a(&bar);
  printf("%d", c(b(foo, bar)));
  return 0;
}
```

Figure 4. Program with commutative function `a()`

3. Commutativity Detection Details

Now that the proposed algorithm has been described at a high-level, this section presents many of the details that make implementation difficult. This section focuses on the two main technical challenges of this process: checking the equivalence of the function executions, and identifying the set of functions that read another function's output.

3.1 Equivalence Checking with Random Interpretation

One of the main difficulties with identifying commutative functions is *proving* that the execution order of a function does not matter for all possible inputs. To accomplish this, the function must be symbolically executed in two different
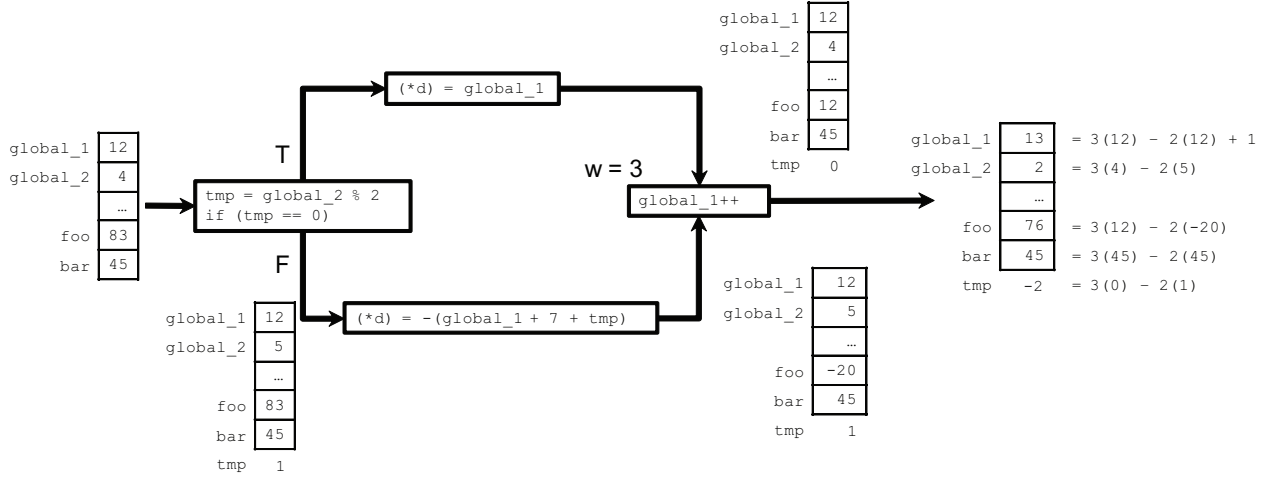
Figure 5. Using random interpretation on function `a()` from Figure 4

orders and shown to be equivalent. Symbolically proving the equivalence of two sections of code is an undecidable problem in its general form, and very complex, even when the application is simplified in ways that guarantee no false negatives are introduced [4]. Symbolic interpretation is simply not scalable to realistically complex functions, which makes this technique infeasible for commutativity analysis.

On the other end of the equivalence testing spectrum is random testing. By repeatedly executing the code using randomly generated inputs, it is possible to determine that a function *might* be commutative. Obviously this method is also not useful, because it is unlikely that the random values would exercise all execution paths in the functions. Any commutativity detected could potentially be a false positive, and lead to incorrect program transformations, which is unacceptable for a compiler.

Random interpretation is a hybrid technique, which combines the simplicity of testing with random inputs, while enabling the compiler to achieve correctness very close to 100%. The reader should note that the commutativity testing algorithm *does not rely on random interpretation*; other fast equivalence testing procedures, such as fractal symbolic analysis [12], could be used just as easily. We chose random interpretation because it is both fast and simple, but this does not necessarily mean that other equivalence testing procedures are impractical.

To give the reader a better idea of how random interpretation works, Figure 4 shows an example program with commutative function `a()` and Figure 5 shows the process of randomly interpreting function `a(&foo)`. The first step in the algorithm is to generate random values for memory used by `a()`, in this case `global_1`, `global_2`, `foo`, and `bar` are set in this way, shown on the left of Figure 5. This state is used to execute the first block of code, creating a value of 0 for `tmp`. The `if` statement evaluates to true, and the taken

branch, at the top of Figure 5, is evaluated yielding the state at the top middle of this figure. You can see that the value of `foo` has been changed to 12, and `tmp` is added to the state.

Up to this point, the algorithm has performed the same as random testing. However, unlike random testing, random interpretation requires that all possible control flow paths are executed. Taking the fall-through path using the initially generated state is nonsensical though, since `tmp` must be 1 along the fall-through path. To properly evaluate the fall-through path, the initial state is cloned and then adjusted to values that will execute along the fall-through path. To accomplish this, the branch condition is solved as an equation, and those results are propagated up through the dataflow graph to ensure that the state is consistent. In this example, `tmp` must equal 1, and so `global_2` is adjusted from 4 to 5. Once the initial state is changed, `tmp` is recomputed which will force interpretation along the fall-through path. This creates the state at the bottom left of Figure 5.

If the equation cannot be solved, e.g., if the branch condition were `if(1 == 0)`, then that control flow path is not interpreted, as it is impossible to encounter during normal execution. After the fall-through path is executed `foo` is updated with the value -20, shown at the bottom right. At this point we have two sets of state that represent different control flow paths through function `a(&foo)`.

Now the different control flow paths merge, meaning the states must be combined. If the states are not combined, the size of the state will exponentially explode, which is the main problem with most equivalence checkers. When control paths merge, i.e., wherever an SSA phi node would be inserted [5], the values of the two states are combined using an *affine join operation*.

The affine join operation creates a weighted average of the values computed on the different paths of execution, using the equation $\phi_w(v_1, v_2) = wv_1 + (1 - w)v_2$, where
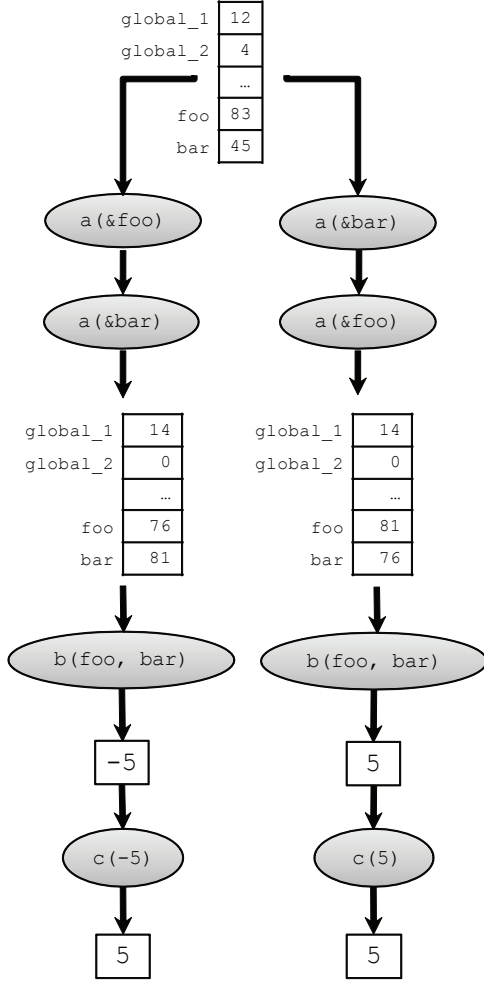
Figure 6. Proving that the reordering of `a()` in Figure 4 is commutative

v_1 and v_2 are the values being merged, and w is a randomly generated weight. In Figure 5, a random weight 3 is chosen to merge the values. After the merge, $\mathtt{foo} = (3*12)+((1-3)*-20)$, and $\mathtt{bar} = (3*45)+((1-3)*45)$, for example. By combining the two states, the random interpreter is able to *capture behavior from all paths of execution*, but without requiring the exponential book keeping associated with symbolic execution. After the state is fully merged, `global_1` is incremented yielding the final state from randomly interpreting `a(&foo)`, shown at the right of Figure 5.

This final state is then used as input to randomly interpret `a(&bar)` producing a new state, shown on the left of Figure 6. In parallel with this, the initial state fed to `a(&foo)` is used to randomly interpret `a(&bar)` executing before `a(&foo)`, shown on the right of Figure 6. These two states do not match because the values in `foo` and `bar` are reversed; previous work would stop checking for commutativity at this point [16].

The algorithm proposed in this work continues by evaluating the lone function, `b()`, that reads `foo` and `bar`, to determine if the state differences affect program output. Using the two different states, `b()` will evaluate to -5 and 5, which still do not match, and so `b()`'s reader function `c()` is evaluated. As shown in Figure 6, the outputs of `c()` do match, meaning the two calls to `a()` can be reordered without affecting program outputs. When parallelizing these functions the compiler must guarantee that all writes to shared variables, `global_1` in this example, are protected by a critical section. This example demonstrates how random interpretation can be used to check the equivalence of different execution orders.

3.1.1 Why Random Interpretation Works

The intuition behind the soundness (i.e., the lack of false positives and negatives) of random interpretation is that the affine join operation creates a superposition of all paths of execution. Thus, *any linear relationship of variables which existed before the join will still exist after the join*. On the other hand, *if any one of the paths of execution violate the linear relationship, then superimposing the violating path will destroy the linear relationship* of all the other paths.

It should be noted that there are a some bad choices for randomly generated inputs and weights that may lead the random interpreter to produce an incorrect result (i.e., falsely proving that a relationship among the variables exists or does not exist). For example, a weight of 1 or 0 will effectively destroy the state from one execution path at an affine join point, which could mask a relationship. However, in the general case, there is at most one bad random value per join in the program [7]. Provided that weights are selected uniformly from the set of integers on a 64-bit machine, *this bounds the chance of error to at most* $\frac{\#joins}{2^{64}}$, which in practice is a negligible probability. Previous work empirically found that the number of joins in a program increases only linearly in the number of program statements, with a linear coefficient between 0.5 and 5.2 [5]. This means that, in the worst case, falsely assuming an unusually large 1000 statement function was commutative would happen with a probability of $\frac{5.2*1000}{2^{64}} \approx 2.8 * 10^{-16}$. If greater accuracy is required, the random interpretation can be run multiple times with different random numbers, and the chance of error will *exponentially decrease* to $(\frac{\#joins}{2^{64}})^{\#runs}$. This probability can be made arbitrarily small until it is less likely than a compiler bug or less likely than an alpha particle strike corrupting the results during execution.

Using random interpretation for commutativity detection is a novel application of this recently proposed program analysis technique. Despite its very high accuracy, random interpretation is simple enough to avoid the scalability problems traditionally endemic to equivalence testing.

3.2 Limitations of the Algorithm

The proposed algorithm can automatically identify several types of commutative functions beyond related work, but it also has several limitations that leave room for future improvement. First, as previously mentioned, this algorithm was not designed to exploit the parallelism from recursive commutative functions. Additionally, random interpretation cannot determine equivalence of floating point computations. This is because combining various paths of execution with an affine join can potentially create rounding errors, although this limitation could be overcome through use of an infinite precision floating point library [19].

Another limitation is that memory allocations are assumed to always succeed or that their failures are not important for semantic correctness. For example, the hash set insertion from Figure 1 were reordered and one of the insertions failed, parallelizing the insertions can potentially change which one fails. One can envision scenarios where this is incorrect behavior.

It should be noted that random interpretation can never prove commutativity with 100% accuracy. To overcome this, the algorithm could be modified to use random interpretation to find *likely* commutative functions, and feed those candidates to a more heavy-weight equivalence checker to prove commutativity.

A last limitation is that the algorithm requires full vision of the program. Without access to all functions it is impossible, in the general case, to identify all possible reader functions of another function's outputs.

Despite the simplicity of the example in Figure 4, random interpretation *can* handle complicated program constructs such as recursive functions and loops. For example, loop constructs simply extend the state duplication with adjustment already described in Figure 5; loops need to be iterated over a specific number of times to ensure that relationships between the variables have stabilized. Details on these extensions can all be found in previous work [7].

One final thing that random interpretation does not handle well is recursive data structures, such as the linked list in Figure 1. Rhetorically, how does one affine join two different linked lists in memory? In order to correctly interpret these data structures, the implementation used in this paper does *not* merge memory state for heap-allocated data structures. Instead, when different heap-allocated data structures are merged, each copy is retained and interpreted separately. This means that the random interpreter's state does exponentially increase in the presence of heap objects. When the data structures grow too large, the algorithm simply aborts interpretation and assumes the function is non-commutative. This drawback could potentially be overcome by utilizing shape analysis, but we leave that to future work.

Even with these drawbacks, Section 4 demonstrates that this algorithm can identify a significant amount of parallelism in legacy applications.

```
void * xalloc(int size) {
    char * old_end_of_array;
    old_end_of_array = end_of_array;
    end_of_array += size;

    if ((end_of_array-start_of_array) > MEMORY_LIMIT) {
        exit(1);
    }
    return old_end_of_array;
}
```

Figure 7. Custom memory allocator adapted from one used in 197.parser

3.3 Identifying Readers of the Function Outputs

A second difficulty with the commutativity algorithm presented is the challenge of identifying which functions read the results of another function. It is necessary to accurately determine this in order to test whether or not an altered memory space created by two different execution orders semantically matters in the program. If too few readers are identified then the compiler might falsely believe that a function is commutative and perform incorrect program transformations. On the other hand, if too many readers are identified, then it is possible that one of the false-readers will use the memory to compute program output and incorrectly invalidate a true commutative function.

Identifying the readers of a function's results is simply a matter of dataflow analysis. For example identifying that $c()$ reads $b()$'s output in Figure 4 is straightforward. However, identifying readers of functions with side-effects requires that the compiler have a large scope (i.e., being able to analyze as much of the application as possible), and high-quality points-to analysis. Points-to analysis determines the set of all possible addresses that a pointer could potentially refer to. When this information is available, then the compiler can easily identify which functions read from areas written by the candidate function.

There are obvious tradeoffs to be made in points-to analysis, as complex analysis reduces the size of points-to sets at the expense of increased runtime. A thorough discussion of these tradeoffs is left to the significant body of related work, e.g., [8], but this section will discuss one analysis which is particularly important for detecting frequently occurring patterns in commutative functions.

One simplification that many points-to analyzers make is treating all heap references as a single object. Thus they assume that any pointer that is assigned from a dynamic allocation site refers to the same set of addresses. This enables points-to analysis to be performed very quickly, however, this simplification is unacceptable for use in commutativity analysis. Figure 7 shows an example of why that is. This figure shows source code for a custom memory allocator similar to one that appears in 197.parser from the SPECint2000 benchmark suite. During program initialization 197.parser allocates a very large array. While the program is running, parser calls `xalloc()` to get a free section of the heap to

perform computations with. If the points to analysis treated the entire heap as a single object, then the commutativity detection algorithm would need to check the vast majority of the application as a potential reader. This ends up preventing the discovery that xalloc() is in fact commutative. Heap-sensitivity is the single most critical characteristic of points-to analysis needed to enable effective commutativity detection.

4. Experimental Evaluation

The proposed commutativity detection algorithm was implemented in the Trimaran compiler system [21]. Random interpretation was implemented as described in [7] and operates on the compiler's intermediate representation. In order to experimentally evaluate the system, commutativity analysis was applied to the MediaBench [11] and SPECint2000 suites. The benchmarks from these suites omitted from the results are missing due to issues within the compiler infrastructure unrelated to the commutativity detection algorithm. The algorithm runtime experiments were performed on a Linux-based system with a 2.4 GHz Intel Core2 Quad CPU and 2GB of RAM.

4.1 Pointer Analysis

The default pointer analysis in our compiler is fairly conservative. It is capable of differentiating global variable and stack accesses, but it assumes all heap references point to the same object. This artificially inflates the number of reader functions that need to be checked for any potentially commutative function that writes to the heap. Function pointers are also handled very conservatively, meaning that if the compiler cannot identify the complete set of functions a pointer could reference using intra-procedural dataflow analysis, it assumes all functions could be referenced by that function pointer. Empirically speaking, the conservative function pointer analysis was not a significant limitation for the benchmarks we examined.

To test the sensitivity of our results to the pointer analysis algorithm, we also generated results using the Fulcra pointer analysis framework [14]. Fulcra is inter-procedural, context-sensitive pointer analysis algorithm, that has support for differentiating heap objects. This analysis takes slightly more compilation time (at most a few minutes), but greatly reduces the number of potential readers for a given function's outputs.

One significant weakness in the proposed commutativity detection algorithm is that the pointer analysis requires full vision of the application to accurately identify the set of reader functions. However, there are a few trends/techniques that mitigate this somewhat. First, the increasing use of object-oriented programming with privatized data semantically limits which functions can read another function's outputs. Thus if the compiler has vision of all of the class methods, that is often sufficient to prove commutativity of a member function. Second, many reader functions are common library functions that are known to not violate commutativity. For example, if the results of a custom memory allocator are passed to memset(), the compiler could "special-case" memset() and know that it will not produce outputs that violate commutativity, thus precluding the need to examine that function. Finally, even when the compiler does not have full vision of the application, this analysis is useful as a way to let the programmer know what could *potentially* be automatically parallelized given some user specified guarantees about external functions not using the results. Having the developer work with the compiler in this way enables fast and effective software creation.

4.2 Algorithm Runtime

Figure 8 shows the runtime of commutativity testing every function in several benchmarks. These numbers should be considered an upper bound on the runtime of the analysis, because many functions are never called consecutively in the target application, and thus would not benefit from commutativity-based parallelization. The "Non-Recursive" bars show the runtime for commutativity testing without examining readers (i.e., if the memory state differs after interpreting the function in two different execution orders, it is assumed to be not commutative and analysis ends). "Depth 1" and "Depth 1 (Fulcra)" interpret the function and its readers, if necessary, using the default and Fulcra pointer analyses, respectively. Similarly "Depth 2" examines the readers, and the readers-of-readers if necessary. We found that recursing beyond depth of two produced very little additional benefit.

In general, the commutativity analysis runtime is sufficiently fast. The worst case runtime for any of the applications was under 5 minutes, and using the higher quality pointer analysis greatly reduced this number across all applications. Better pointer analysis reduces the number of false readers, and as this result suggests, the primary contributor of runtime is repeated random interpretation of large functions. It should be noted that evaluating the extra readers did *not* affect whether a function was found to be commutative. We found that with both Fulcra and the more naive pointer analysis, the same number of commutative functions were identified.

Looking at specific examples, one application that took a relatively long time to analyze is 186.crafty. This application stored a significant amount of data in global variables that were accessed by many functions. This means that many functions had a large number of readers, and many of the readers were very long, e.g., Evaluate(). As previously mentioned, 197.parser also took a relatively long time because of its custom memory allocator, which created a significant number of readers, particularly with heap-insensitive pointer analysis.

On average, the algorithm required less than a minute to test the commutativity of all functions in an application. This result demonstrates that random interpretation is very fast
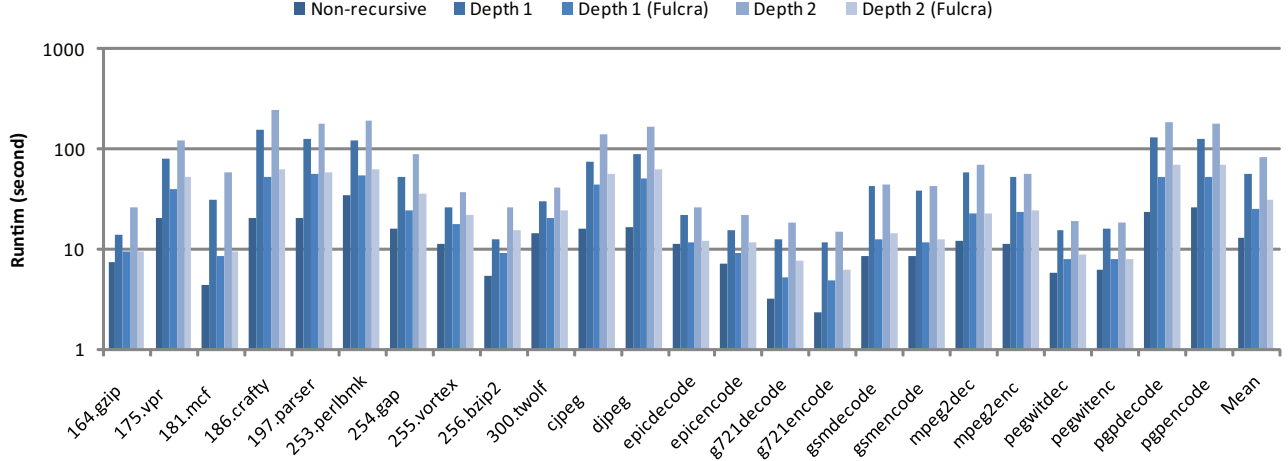
Figure 8. Runtime for the proposed commutativity analysis for all functions in each benchmark.

and can be used for equivalence checking in real applications, as opposed to symbolic execution, which can be too slow for these situations.

4.3 Number of Commutative Functions Identified

Figure 9 shows the number of commutative functions identified by our analysis as the percentage of the total functions in each program. The "Non-recursive" bars indicate the commutative functions that were identified without performing the `recursive_test` portion of the proposed algorithm. This result is equivalent to the commutativity test proposed by previous work [16]. The "Depth 1" and "Depth 2" bars show the fraction of the functions that were identified as commutative using the `recursive_test`, but bounding the recursive search to the respective depths of recursion. The "semantic" bars will be explained momentarily.

As mentioned in Section 4.1, vision of the entire application is necessary to find all commutative functions within an application. Calls to common library functions would prevent the identification of many commutative functions unless the compiler had special knowledge of their side-effects. In order to provide the compiler with this knowledge, we utilize a manually created database of common functions known to not affect commutativity, and the results reflect this.

One surprising result from Figure 9 is that in almost all benchmarks, a large fraction of functions are commutative. Note that not all of these functions can be parallelized for performance improvements, because they are not called consecutively. In general, MediaBench benchmarks had a larger fraction of commutative functions identified than that for the SPECint2000 benchmarks using the non-recursive approach. These programs were typically found to have a lot of small functions that were mostly used for computation on the arguments and did not touch as many global variables. One example of a useful commutative function identified

by this method is the `NN_Div` function in the `pgpdecode` and `pgpencode` benchmarks. This function is called within loops and has the potential for parallelization, providing good speedup.

The most important result to take from Figures 9 is that the recursive commutativity test identified many more commutative functions than the non-recursive version for nearly all benchmarks.

4.4 Semantic Changes for Commutativity

When examining the benchmarks for commutative functions discovered by the algorithm, we noticed several examples of functions that would be commutative, except for a few simple dependencies. For example, the function `simpleSort()` in 256.bzip2 has one line that prints to `stderr` if a debugging flag is set. A human can recognize that the order the debugging output is printed may not matter, but the compiler cannot possibly automatically parallelize this function because doing so would *semantically change* the application.

Even though the commutativity analysis algorithm could not automatically parallelize `simpleSort()`, it proved very useful, providing a guide for what semantic changes needed to be made in order to enable automatic parallelization of the application. We used this technique, having the compiler identify statements and manually deciding whether the semantic changes would create an incorrect program output, to identify several more commutative functions. The results of this are shown in the "semantic" bars in Figures 9.

Our manual semantic changes consisted only of delineating what debugging output, such as the bzip2 example described above, and certain library functions, such as `calloc()`, were reorderable. Manually changing the semantics of these functions significantly increased the number of commutative functions identified for some applica-
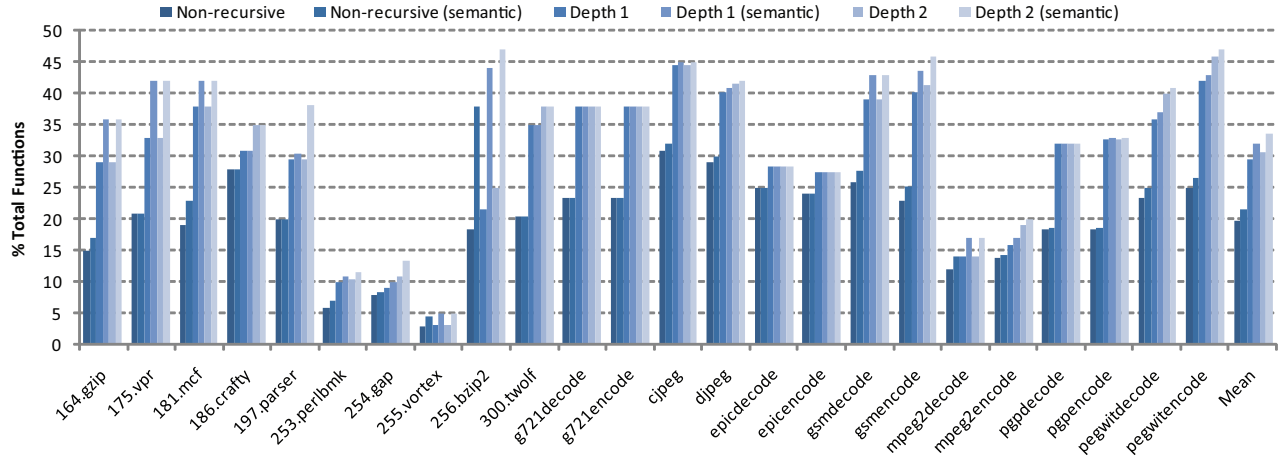
Figure 9. Percentage of functions identified commutative in SPECint2000 and MediaBench benchmarks.

tions. SPECint applications generally benefitted more than MediaBench applications because SPECint tended to have more debugging output.

4.5 Performance Evaluation

In order to evaluate the performance benefit achieved from parallelizing commutative functions, we modified Trimaran to generate code overlapping as many commutative functions as possible. That is, Trimaran inlined functions that appeared consecutively in the code (or unrolled loops and then inlined functions), and scheduled the results on an infinite-issue machine, while still respecting dependencies. The scheduler leveraged manual semantic changes to the code (e.g., identification of debugging output) as well as the functions that were identified automatically by the commutativity detection algorithm. The functions were scheduled on a hypothetical machine with infinite resources and a perfect memory system. The reasoning behind this evaluation technique is to get an idea of how much parallelism can be uncovered using this type of analysis without complicating matters due to microarchitectural issues. These results provide a first-order approximation of the parallelism that can be achieved, and if nothing else, can be used to compare the different commutativity detection algorithms between each other. We are actively working on getting results using these algorithms on real hardware, as well, but time restrictions prevent us from including them here.

The parallelism numbers for our applications are shown in Figure 10. This chart demonstrates that amount of parallelism achieved varies significantly from application to application. For example, `gsmencode`, `gsmdecode` and `164.gzip` showed little to no benefit from the identification of commutative functions. The `164.gzip` result agrees with prior work [2], which found commutativity was not useful for that application. Commutative functions are

identified in these programs, however, their call-sites are not dependence free, and thus cannot be overlapped.

Several benchmarks show a significant amount of parallelism, however. For example, `pegwitdec` achieves significant parallelism by overlapping many of its functions which compute Galois field arithmetic, such as `gfMultiply()` and `gfSquare()`. Many of these functions required recursively testing readers in order to correctly identify that the functions were in fact commutative. `255.vortex` also showed significant parallelism from identifying the commutative function `Query_AssertOnObject()`, which also required testing the readers. Manually providing semantic hints about commutativity *did not improve the parallelism of these applications* significantly, except in two cases: identifying `simpleSort()` in bzip2 improved the parallelism from 1.51 to 2.06, and `xalloc()` in 197.parser improved that function from 1.51 to 1.6.

Generally speaking the new commutativity analysis technique was able to uncover significant amounts of parallelism automatically, and performs significantly better than previous commutativity detection strategies.

5. Related Work

Detecting commutative functions for automatic parallelization has not been the subject of extensive prior research, although the utility of detecting commutative functions has been well described. For example, Bridges et al. [2] recently noted the importance of leveraging commutativity in automatically parallelizing programs by showing improvements on the SPECint2000 benchmark suite. Prior to that work, Rinard and Lam created the Jade programming language, which had semantics allowing the programmer to manually specify if functions were commutative [10, 17]. Wu et al. [23] recently showed that commutativity can be leveraged to significantly speed up database applications, and went further, examining why commutativity exists in applications.
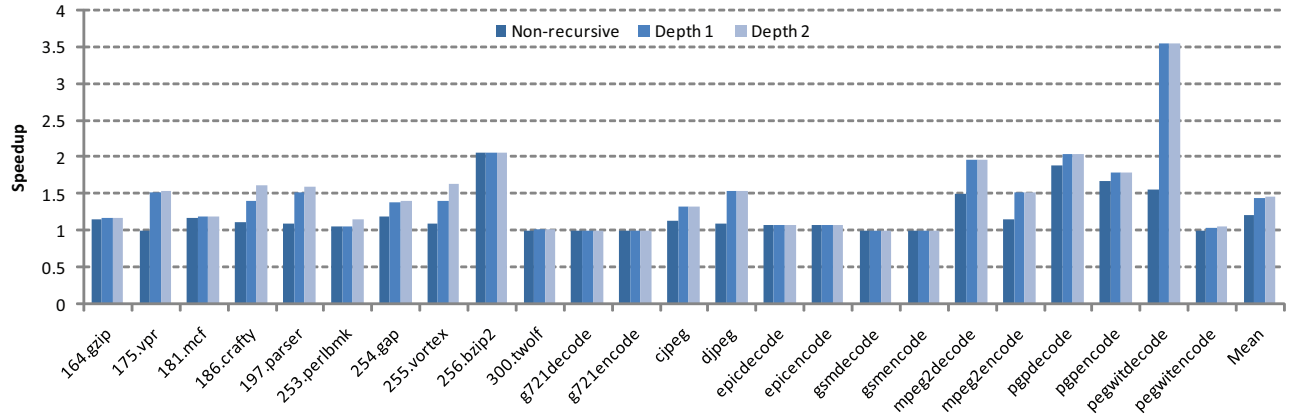
Figure 10. Parallelism uncovered by various commutativity analyses.

Many people have also investigated restructuring applications by leveraging the commutativity of individual operations [22]. That work restructured computation, such as iterative additions into a tree of additions, in order to take advantage of addition's commutativity. Rinard et al.'s later work generalizes this result to detect entire commutative functions [16]. However, the approach taken in that work only checks that the output of the reordered functions is identical (i.e., produces the same output). The `a()` function in Figure 4 is an example of a function that is handled by our proposed approach, but is not handled by Rinard et al.'s method. Another significant difference between Rinard's work and our approach is that they used symbolic interpretation to detect the equivalence of execution orders, where we use random interpretation. Random interpretation does not suffer from the scalability issues typically associated symbolic interpretation. Although our proposed commutativity detection algorithm leverages on random interpretation, other methods, such as verified abstract states [24] or fractal symbolic analysis [12], could be used as well.

6. Summary

Commutative functions are a significant source of parallelism in many applications. This paper introduces a novel compiler algorithm that can detect these functions automatically. Whereas previous work required that the output of the commutative functions must match after reordering, this paper leverages the observation that only the *program* outputs matter; if the reordered function outputs do not affect the program output, then the function *is* commutative.

To detect these situations, our algorithm uses advanced pointer analysis to identify the functions which read the results produced by a potentially commutative function. These reader are evaluated using *random interpretation*, a scalable method for proving the equivalence of two program representations. This algorithm is sufficiently fast, taking less than 5 minutes in the worst case, and the algorithm can detect

13% more commutative functions than prior work, which results in an average of 28% more parallelism uncovered.

Even when the commutativity detection cannot find significant amounts of parallelism automatically, the algorithm can present a software developer with a list of small semantic changes that would enable automatic parallelization. This method of commutativity analysis can be a significant productivity aid for software developers creating applications for manycore architectures.

Acknowledgments

References

[1] W. Blume et al. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[2] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multi-core. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 69–84, 2007.

[3] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, 2003.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[6] Z.-H. Du et al. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proc. of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 71–81, 2004.

[7] S. Gulwani. *Program Analysis using Random Interpretation*. PhD thesis, University of California Berkeley, 2005.

[8] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. of the 2001 ACM Workshop on Program Analysis For Software Tools and Engineering*, pages 54–61, June 2001.

[9] W. Hwu et al. Implicitly parallel programming models for thousand-core microprocessors. In *Proc. of the 44th Design Automation Conference*, pages 754–759, June 2007.

[10] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, 1991.

[11] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[12] V. Menon, K. Pengali, and N. Mateev. Fractal symbolic analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):776–813, Nov. 2003.

[13] S. Moon, B. So, and M. W. Hall. Evaluating automatic parallelization in SUIF. *Journal of Parallel and Distributed Computing*, 11(1):36–49, 2000.

[14] E. Nystrom, H.-S. Kim, and W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proc. of the 11th Static Analysis Symposium*, pages 165–180, Aug. 2004.

[15] M. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *Proc. of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 142–152, June 2005.

[16] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):1–47, Nov. 1997.

[17] M. C. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford University, 1994.

[18] S. Ryoo et al. Automatic discovery of coarse-grained parallelism in media applications. *Transactions on High Performance Embedded Architectures and Compilers*, 1(1):194–213, Jan. 2007.

[19] J. Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Journal on Discrete and Computational Geometry*, 18(3):305–363, 1997.

[20] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[21] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[22] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. of the 1990 European Symposium on Programming*, pages 344–358, 1990.

[23] P. Wu and A. Fekete. An empirical study of commutativity in application code. In *Proc. of the 2003 International Database Engineering and Applications Symposium*, page 358, 2003.

[24] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *Proc. of the SIGPLAN '08 Conference on Programming Language Design and Implementation*, pages 349–361, 2008.

[25] H. Zhong et al. Uncovering hidden loop level parallelism in sequential applications. In *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, pages 290–301, 2008.