Cache Locality Optimization for Recursive Programs

Jonathan Lifflander
Sandia National Laboratory
Livermore, CA
jliffla@sandia.gov

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
Richland, WA
sriram@pnnl.gov

Abstract

We present an approach to optimize the cache locality for recursive programs by dynamically splicing—recursively interleaving—the execution of distinct function invocations. By utilizing data effect annotations, we identify concurrency and data reuse opportunities across function invocations and interleave them to reduce reuse distance. We present algorithms that efficiently track effects in recursive programs, detect interference and dependencies, and interleave execution of function invocations using user-level (non-kernel) lightweight threads. To enable multi-core execution, a program is parallelized using a nested fork/join programming model. Our cache optimization strategy is designed to work in the context of a random work-stealing scheduler. We present an implementation using the MIT Cilk framework that demonstrates significant improvements in sequential and parallel performance, competitive with a state-of-the-art compile-time optimizer for loop programs and a domain-specific optimizer for stencil programs.

CCS Concepts • **Software and its engineering** → **Recursion**; *Coroutines*

Keywords recursive programs; locality optimization

1. Introduction

Maximizing application performance requires careful orchestration of the execution to best match the given architecture. Manually performing this task is expensive and prone to error because of the non-obvious performance implications of source code transformations. The two most common automated approaches consist of compiler analysis and optimization, possibly coupled with automated exploration of the optimization space, or application composition from calls to carefully optimized libraries. Automated approaches can be applied to large programs, but they are limited in the class of programs handled, types of transformations considered, and architectural features that can be optimized. Equally important, such optimizers often generate complicated code that adversely impacts other optimization phases (e.g., register allocation or vectorization). Tuned libraries provide a set of commonly used functions that have been carefully optimized for a given architecture. However, an application composed of library calls might not exploit optimization opportunities across function boundaries.

In general, applications are composed of multiple functions embedded in different libraries, motivating the need for interprocedural locality analysis and optimization. We exploit data access (effect) annotations to identify data reuse opportunities. Effect annotations specify the regions of data read and written by a function invocation. Prior efforts have employed such annotations to determine concurrency between sequentially ordered function invocations at the coarsest possible granularity to minimize runtime overheads. Cache locality optimization requires going one step further: scheduling invocations so the data remain in cache between consecutive uses of a data region across function boundaries.

To increase the chances of cache reuse between access to the same data by the leading and trailing functions, we dynamically interleave (*splice*) their execution. The effect information is tracked in the call stack as the functions are executed. When the trailing function attempts an operation that would violate dependences inferred from the effect annotations, it is delayed and enqueued as waiting on the frame in the leading function it depends on. Depending on its effects, some sub-computations in the trailing function's invocation might be delayed, while others continue to be interleaved. We attempt to maximally interleave the executions while satisfying the dependences.

Often, recursive programs are parallelized using nested fork/join programming systems. Fork/join programming systems divide a given work into subtasks that can be executed concurrently. These programming models underpin several popular approaches to multicore parallelization (e.g., the Cilk family [3, 13, 14], Thread Building Blocks [34], Task Parallel Library [43], OpenMP [28], and X10 [48]). Many common idioms, such as sequential and parallel loops

and data iterators, can be represented using nested fork/join parallelism. Fork/join programs are typically scheduled using work stealing to balance the load across idle processes. We present an algorithm that further exploits data effects to generate a fork/join program automatically, combining depth-first execution of a nested fork/join program and a dependence-driven task-graph scheduler to execute the delayed sub-computations. The entire scheduling strategy coexists with a work-stealing scheduler.

We implement our approach in the MIT Cilk framework [26] and demonstrate that the effect system, concurrency and locality checks, and interleaved execution can be managed efficiently to accrue cache locality benefits. We also show that the space overheads of delayed execution are low. We evaluate the benefits of dynamic splicing compared to state-of-the-art approaches for optimizing loop programs. These approaches were chosen because of the compiler optimizations' ability to achieve the best performance. Specifically, we demonstrate that our approach can dynamically splice multiple function invocations to achieve performance comparable to optimized programs generated by Pluto, a polyhedral optimizer for affine loop programs, and Pochoir, a domain-specific language for stencil programs.

The primary contributions of this paper include:

- Dynamic splicing as an approach to optimizing cache locality
- Techniques to efficiently track effects and detect interference
- An efficient scheduler that combines depth-first execution of function invocations, dependence-driven task scheduling for delayed sub-computations, and work-stealing-based load balancing
- Experimental evaluation that demonstrates dynamic splicing can match performance achieved, in serial and in parallel, by complex compile-time transformations, such as diamond tiling.

2. Problem Statement

Consider the example program in Figure 1a[1]. It involves two operations, each copying array A to a different array. This program can be implemented as two calls to the optimized memcpy() routine in the C library. This implementation involves four array transfers across the memory hierarchy—once for B and C and twice for A. A compile-time optimizer can fuse the two operations to avoid moving array A twice across the memory hierarchy.

While compile-time techniques improve performance, applying them in the context of recursive programs involves overcoming several challenges. The data accesses in a recursive function invocation might only be known at runtime. Even if it can be approximated at compile time, effecting such a compiler transformation across function boundaries can be a non-trivial task. Finally, compiler transformation assumes that the source code is available for all functions of interest and in a specific form amenable to analysis transformation (e.g., non-linearized indices [25]). In this paper, we focus on a runtime approach to achieving the benefits of compile-time fusion for recursive programs (such as the one in Figure 1b).

Specifically, given a sequential recursive program, we try to answer the following questions:

- How can we efficiently capture dependence and reuse information across function invocations at a fine enough granularity?
- How can we adapt the runtime schedule to exploit the identified data reuse across function invocations?

3. Solution Approach

In this section, we present our runtime approach that dynamically *splices*, or interleaves, recursive function invocations in a sequential program to improve memory hierarchy data reuse. Our approach is based on the following observations:

- Inclusive effect annotations can help compactly capture and track data access and dependences in recursive programs.
- The execution order of work within distinct function invocations can be controlled by interleaving user-level (non-kernel) lightweight threads that execute the function invocations on the *same* hardware thread, similar to coroutines [12]. Thus, each distinct function invocation can maintain its own stack using a lightweight thread.

Effect annotations. One key to splicing involves tracking execution in the consecutive function invocations to ensure that no dependences are violated. To track these dependences, we consider functions written in a recursive form with effect annotations. Figure 2 describes the language for effect-annotated recursive programs. The effect annotations associated with a function definition specify the data regions read or written by an invocation in terms of its formal parameters. Effect annotations are also associated with each *step*—a serial block with no spliceable function invocations present. When no step effect annotation is provided, an empty effect annotation is assumed. In order to effectively splice, we obtain the effects of a function's *continutation* through static compiler analysis that determines the union of all reachable effects following a function invocation. Figure 3 shows an effect-annotated copy function.[2]

Spliced execution using lightweight threads. To enable inter-invocation splicing, we must explore the enclosing scope to discover future function invocations that may be spliced. When normal execution encounters a function invocation with an effect and this location is marked to attempt to splice, we begin spliced execution by creating a new user-level thread, referred to as the *trailing thread*, and execute

[1] While the implementation was done in C, we use Python syntax to present the key routines and examples.

[2] Detailed specification of effects is provided in Appendix A.

```
1  char A[M], B[M], C[M]
2
3  copy(A,B,M)
4  copy(A,C,M)
5
6  def copy(X, Y, n):
7    for (i = 0; i < n; i++):
8      Y[i] = X[i]
```

(a) A sequence of copy operations

```
1  char A[M], B[M], C[M]
2
3  copy(A, B, M)
4  copy(A, C, M)
5
6  def copy(X, Y, n):
7    if n < threshold:
8      for (i = 0; i < n; i++): Y[i] = X[i]
9    else:
10     mid = n/2
11     copy(X, Y, mid)
12     copy(X+mid, Y+mid, n-mid)
```

(b) Recursive implementation of the copy operation

```
1  char A[M], B[M], C[M]
2
3  spawn copy(A, B, M)
4  spawn copy(A, C, M)
5
6  def copy(X, Y, n):
7    if n < threshold:
8      for (i = 0; i < n; i++):
9        Y[i] = X[i]
10   else:
11     mid = n/2
12     spawn copy(X, Y, mid)
13     spawn copy(X+mid, Y+mid, n-mid)
```

(c) Fork/join implementation of the copy operation

```
1  def copy(X, Y, n):
2    if n < threshold:
3      if /*dependences satisfied*/:
4        for (i = 0; i < n; i++)
5          Y[i] = X[i]
6      else:
7        /*delay step*/
8    else:
9      mid = n/2
10     /*yield to next thread*/
11     copy(X, Y, mid)
12     /*yield to next thread*/
13     copy(X+mid, Y+mid, n-mid)
```

(d) Illustration of splicing for cache locality

Figure 1. Example copy program implemented sequentially (with for-loops), recursively, concurrently with fork/join, and transformed for splicing.

$$
\begin{aligned}
v &\in \mathbb{Z} && [Values]\\
b &\in \{\textbf{true}, \textbf{false}\} && [Booleans]\\
p &\in \{p_1, p_2, \ldots, p_k\} && [Parameters]\\
l &\in \{l_1, l_2, \ldots\} && [Locals]\\
g &\in \{g_1, g_2, \ldots\} && [Globals]
\end{aligned}
$$

$$
\begin{aligned}
e_b \in BExprs &::= f_b(e_1, e_2, \ldots)\\
pr_f \in FPragmas &::= \textbf{\#pragma splice func } \textbf{Reads}(e)\ \textbf{Writes}(e)\\
pr_s \in SPragmas &::= \textbf{\#pragma splice step } \textbf{Reads}(e)\ \textbf{Writes}(e)\\
&\quad \mid \epsilon\\
e \in Exprs &::= v \mid l \mid g \mid p \mid e_b \mid f_v(e_1, e_2, \ldots)\\
s \in Stmts &::= \textbf{return} \mid s; s \mid l := e \mid g := e\\
&\quad \mid \textbf{if } e_b \textbf{ then } s \textbf{ else } s \mid \textbf{while } (e_b)\ s\\
&\quad \mid f(e_1, e_2, \ldots, e_k)\ \ pr_s\\
ss \in StepStmts &::= pr_s\ \ s\\
m \in Method &::= pr_f\ \ f(p_1, \ldots, p_k)\ \ ss\\
pgm \in Program &::= m\ \ pgm \mid ss
\end{aligned}
$$

Figure 2. Language for effect-annotated recursive programs.

the function invocation in one thread and its continuation in another, the *leading* and *trailing* threads, respectively. The continuation is explored to reach the next function invocation. With two function invocations, each in its own stack, we execute them in an interleaved fashion.

Maintaining multiple call stacks. During interleaved execution, we maintain the call trees for the two invocations in a symmetric fashion. That is, every function invocation (and return) in one stack is matched with a corresponding invocation (and return) in the other. The sequential yet interleaved execution of user-level threads in the same hardware thread dynamically produces an effect similar to compile-time fusion of the functions.

Dependence management. Two consecutive function invocations with only shared read effects can be fully spliced. Opportunities for cache locality optimization also arise between dependent invocations. For example, the last two copy operations (lines 6 and 7) in Figure 3 can benefit from splicing, even though the array C produced by the second statement is used in the third. Naively splicing such invocations can violate dependences and lead to incorrect execution. We check the effects of statements in the trailing thread for interference with the pending continuations in the leading thread. Subtrees of the trailing thread's call tree that might interfere are delayed (placed in the heap) for execution at a later point in time. This allows the runtime to continue spliced execution. When a delayed sub-computation's dependences are satisfied, it is immediately executed to exploit reuse.

```
1  RangeE:
2    void *ptr, *ptr2;
3  RangeE RE(void* ptr,void* ptr2); /*constructor*/
4
5  copy(A, B, M)
6  copy(A, C, M)
7  copy(C+1, D, M-1)
8
9  #pragma splice func Reads(RE(X,X+n)) Writes(RE(Y,Y+n))
10 def copy(X, Y, n):
11   #pragma splice step Reads(RE(X,X+n)) Writes(RE(Y,Y+n))
12   if n < threshold:
13     for (i = 0; i < n; i++)
14       Y[i] = X[i]
15   else:
16     mid = n/2
17     copy(X, Y, mid)
18     copy(X+mid, Y+mid, n-mid)
```

Figure 3. Illustration of effect annotations. The RE function call returns an effect object.

Figure 1d illustrates the interleaved execution of the copy function. Upon encountering a step (line 4), we check execute it if its dependences are satisfied.

The remainder of this paper addresses several challenges in making this approach work in practice. Frequent switching between thread contexts can be expensive. The dependence tracking requires additional information on data accesses. Dependence management must be optimized to avoid checking for dependences between every step in one task with every step in another. In addition, dependences need to

be tracked for any postponed steps. Significantly improving cache locality may require spliced execution of several functions, further increasing the runtime cost.

4. Data Effect Annotations for Recursive Programs

To correctly splice function invocations, the runtime system must be cognizant of the data accesses. The runtime obtains this information in the form of effect annotations on function invocations and steps (statement blocks between function invocations). An effect annotation specifies the data regions being read and written. Figure 3 depicts an example effect-annotated program.

A step's effect annotation specifies the data regions it reads and writes. The effect annotation associated with a function call includes all transitive effects encountered during its execution. In other words, the read/write effects of a function invocation are a subset of the read/write effects of the invoking function, as well as a subset of the effects of the continuations that enclose the call site. A step's effect is specified immediately preceding the statement block. The effect annotation associated with a function definition specifies the effects of a function invocation in terms of the function's formal parameters. The compiler determines the continuation effects following a function invocation as the union of all effects reachable after the function invocation.

The inclusive, hierarchical structure of the effect specification enables the runtime to quickly determine whether locality benefits can be accrued and if a step's execution will lead to a conflict. The compiler derivation of continuation effects is a crucial requirement that enables efficient runtime checks for interference. After each function invocation, the effects for the continuation in the current enclosing function scope must be determined to contain the combined effects remaining in that scope. This enables the runtime to determine if a step in a trailing sub-computation will conflict with future accesses encapsulated by the continuation.

While our approach is applicable to a diverse range of recursive programs, specifying and managing arbitrary effects can be expensive. Many recursive applications employ coarse base cases to minimize the cost of recursion. This also helps reduce the cost of managing the effects at runtime. The shared state modified by various function calls often can be compactly described—array sections, sub-trees, spatial regions, etc. In general, the most compact description of read/write effects depends greatly on the computation being performed. Therefore, we allow the user to define the effect types and associated operators. For example, in Figure 3, RangeE is a user-defined effect type, constructed using side-effect-free functions (RE() in the example).

Any effect type must support an interference operator that is used at runtime to ensure dependences are met while accruing locality benefits. Figure 2 includes the language specification for expressing an effect-annotated program.

$$\frac{\langle e,\sigma,\rho\rangle \to v}{\langle g := e,\sigma,\rho\rangle \to \langle \sigma[g \mapsto v],\rho\rangle}$$

$$\frac{\langle e,\sigma,\rho\rangle \to v, \rho = (\rho^{\text{top}}|\rho^{\text{rest}})}{\langle l := e,\sigma,\rho\rangle \to \langle \sigma,\rho^{\text{top}}[l \mapsto v]|\rho^{\text{rest}}\rangle}$$

$$\frac{\langle s_1,\sigma,\rho\rangle \to \langle s_1',\sigma_1,\rho_1\rangle}{\langle s_1; s_2,\sigma,\rho\rangle \to \langle s_1'; s_2,\sigma_1,\rho_1\rangle}$$

$$\frac{\langle s_1,\sigma,\rho\rangle \to \langle \sigma_1,\rho_1\rangle}{\langle s_1; s_2,\sigma,\rho\rangle \to \langle s_2,\sigma_1,\rho_1\rangle}$$

$$[\text{if}_t]\ \frac{\langle e_b,\sigma,\rho\rangle \to \textbf{true}}{\langle \textbf{if } e_b \textbf{ then } s_1 \textbf{ else } s_2,\sigma,\rho\rangle \to \langle s_1,\sigma,\rho\rangle}$$

$$[\text{if}_f]\ \frac{\langle e_b,\sigma,\rho\rangle \to \textbf{false}}{\langle \textbf{if } e_b \textbf{ then } s_1 \textbf{ else } s_2,\sigma,\rho\rangle \to \langle s_2,\sigma,\rho\rangle}$$

$$[\text{while}_t]\ \frac{\langle e_b,\sigma,\rho\rangle \to \textbf{true}}{\langle \textbf{while } (e_b)\ s,\sigma,\rho\rangle \to \langle s; \textbf{while } (e_b)\ s,\sigma,\rho\rangle}$$

$$[\text{while}_f]\ \frac{\langle e_b,\sigma,\rho\rangle \to \textbf{false}}{\langle \textbf{while } (e_b)\ s,\sigma,\rho\rangle \to \langle \sigma,\rho\rangle}$$

$$[\text{fn def}]\ \frac{}{\langle f(p_1,\ldots,p_k)s,\sigma,[]\rangle \to \langle \sigma[f \mapsto \lambda(p_1,\ldots,p_k).s],[]\rangle}$$

$$[\text{fn call}]\ \frac{\langle e_1,\sigma,\rho\rangle \to v_1 .. \langle e_k,\sigma,\rho\rangle \to v_k, \langle f,\sigma,\rho\rangle \to \lambda(p_1..p_k).s}{\langle f(e_1,\ldots,e_k),\sigma,\rho\rangle \to \langle s,\sigma,[p_1 \mapsto v_1,\ldots,p_k \mapsto v_k, \textbf{cont} \mapsto \textbf{this}]|\rho\rangle}$$

$$\frac{x(\textbf{cont}) \to s}{\langle \textbf{return},\sigma,x|\rho\rangle \to \langle s,\sigma,\rho\rangle}$$

Figure 4. Semantics of the language defined in Figure 2.

5. Splicing Scheduler

Figure 4 depicts the semantics of unspliced execution for the language shown in Figure 2. We use this to denote the executing statement's continuation, captured as the statement to be executed upon return from a function invocation.

The splicing scheduler interleaves the execution of two functions, e.g., f1 and f2, denoted by their statement bodies, by maintaining and context switching between concurrent stacks, one per spliced function. Let the invocation of f2 be a statement in f1's continuation. The thread executing f1 is denoted as the leading thread. The scheduler must ensure that spliced execution of the trailing thread, executing f2, does not violate any dependences. We first define the spliced execution by extending the semantics of unspliced execution without accounting for dependences.

The actions of the splicing scheduler in splicing two threads can be described as follows:

- Execution of the leading thread continues until a function call is encountered.
- Upon entering a function call, the leading thread enters the new function but context switches to the trailing thread.
- Execution of the trailing thread continues until a function call or return is encountered. If a function call is encountered, it enters the new function and context switches back to the leading thread. Otherwise, before executing the return statement, it context switches back to the leading thread until the leading thread executes a return at the same stack depth. Similar logic is applied when the lead-

ing thread attempts to return and the trailing thread has a deeper stack.

Figure 5 illustrates and specifies the semantics for the five transitions when splicing two threads. Intuitively, the scheduler attempts to keep the stacks in line with each other by context switching at appropriate points based on the actions taken (push or pop) by the spliced program. In the figure, upward arrows denote function invocations, while downward arrows designate return from functions. We denote spliced execution of statements s_1 and s_2 as $s_1 \S s_2$. Spliced execution maintains a pair of statements to be executed and a pair of stacks $\rho_1 \S \rho_2$. Both statements operate on the same global store σ. A stack ρ is organized as listed of frames labeled $\rho^0 \ldots \rho^{top}$, where ρ^0 is the oldest frame in ρ and ρ^{top} is the newest. The semantics describe exactly when the scheduler context switches based on the depths of each stack in the leading thread (ρ_1) and trailing thread (ρ_2). skip is a statement used to handle the return path.

Dependence checks. A step, function invocation, or a continuation is said to interfere with another if their effects share a data region (non-null intersection) and one of the shared effects is a write. Interfering operations must be executed in the specified program order to preserve dependences. To enable tracking of dependences, we associate each frame in this stack with its read and write effects as shown in Figure 6. Each stack frame tracks the read and write effects of its execution (reff and weff)[3].

Before executing a step in the trailing step (transitions II and IV in Figure 5), the step's effects are checked to be non-interfering. This is computed as:

$$\wedge_{i=0}^{top} \neg \text{interfere}\left(\left(\rho_1^i(\text{reff}), \rho_1^i(\text{weff})\right), SE[\![s_2]\!](\sigma, \rho_2)\right)$$

where:

$$\text{interfere}\left((r1, w1), (r2, w2)\right) = (r1 \cap w2) \text{ or } (r2 \cap w1) \text{ or } (w1 \cap w2).$$

Locality checks. In addition to checking for dependences, a step execution in the trailing thread is spliced with the execution of the leading thread (transition II in Figure 5) only when there is potential for improved cache locality from data reuse. This is determined as the intersection of the total effects of the computation at the top of the trailing thread's stack and effects of the leading thread's computation it can be spliced with. Unless the trailing thread is in its return path, the leading thread's stack is one level deeper than the trailing thread's when this check is performed (rule II), enabling us to perform the check as follows:

$$\left(\rho_1^{top}(\text{reff}) \cup \rho_1^{top}(\text{weff}) \cup \rho_1^{top-1}(\text{reff}) \cup \rho_1^{top-1}(\text{weff})\right) \cap \left(\rho_2^{top}(\text{reff}) \cup \rho_2^{top}(\text{weff})\right) \neq \emptyset.$$

When there is no locality benefit from splicing, execution follows the transitions as if the trailing thread is on the return path. When the leading thread is ready to return from the level at which the execution was unspliced, transition IV is used to complete execution of the trailing thread (without splicing) until both threads are ready to return from the same level (transition V).

Delaying a step. When a step interferes with a preceding thread's effect, it cannot be immediately executed. In this scenario, the frame in the trailing thread is removed from the stack and placed on the heap. The delayed frame is "registered" with all of the frames it depends on. Execution continues with the continuation of the delayed step at the top of the trailing thread's stack. The delayed steps in the heap are structured as a task graph with each step tracking the number of incoming dependences and the steps that depend on it (outgoing dependences).

Releasing dependences. When a stack frame completes execution and is ready to be destroyed, all steps tracked as being dependent on it are notified. If any of these steps are ready (do not have any pending dependences), they are immediately executed. This can cause more frames to be destroyed, enabling more steps. All such transitively enabled steps are executed immediately.

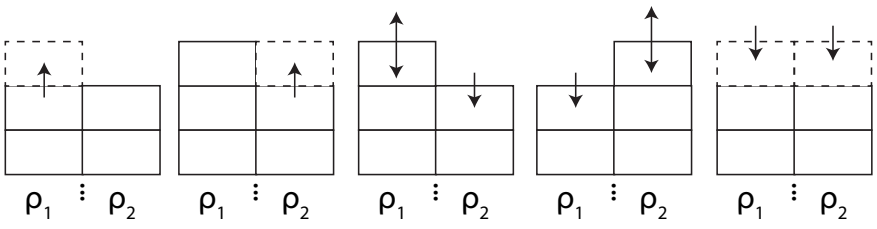
6. Splicing in Parallel

This section describes how the data effect system and splicing approach can be used to generate nested, parallel programs scheduled with work stealing.

Recursive programs can be translated into nested fork/join programs using a few simple keywords. Figure 1c shows the example copy program parallelized using Cilk primitives. The spawn keyword signals that the function invocation can be executed concurrently with the subsequent code. The sync keyword signifies a dependence between one or more of the spawned functions that precede the sync and any statements that follow it. In other words, the sync statement acts as an ordering constraint, stating that the computation past the sync cannot begin until all computations preceding the sync in the task are complete. The two keywords, spawn and sync (or their variants in other languages), constitute the key components of a large class of nested fork/join programs [14, 22, 35]. This spawn-sync model, together with the work-stealing scheduler, is used in the Cilk Plus C/C++ language extensions, now widely available in several mainstream compilers, including Intel ICC, GNU GCC, and Clang. The associated scheduling policies provide provably good space and time bounds [3] within a constant factor.

In the parallel context, a *step* is a sequence of instructions with no interleaving spawn or sync. Each step is executed by exactly one worker thread and cannot be migrated once it begins execution. Each spawn has an associated level, also referred to as its *stack depth*. The initial task has a level of 0. A task's level is defined as one greater than the level of the task that spawned it. The program is executed using a work-stealing scheduler. One worker thread begins execution with

[3] Semantics of step (SE), continuation (CE), and inclusive (IE) effects are presented in Appendix A.



$$\text{I}\ \frac{\langle s_1,\sigma,\rho_1\rangle \to \langle s_1',\sigma',\rho_1'\rangle, \mathrm{len}(\rho_1') \geq \mathrm{len}(\rho_1) = \mathrm{len}(\rho_2),\ \rho_1^{\mathrm{top}}(\mathrm{term}) = \textbf{false}}{\langle (s_1 \S s_2),\sigma,(\rho_1 \S \rho_2)\rangle \to \langle (s_1' \S s_2),\sigma',(\rho_1' \S \rho_2)\rangle}$$

$$\text{II}\ \frac{\langle s_2,\sigma,\rho_2\rangle \to \langle s_2',\sigma',\rho_2'\rangle, \mathrm{len}(\rho_2) < \mathrm{len}(\rho_1),\ \mathrm{len}(\rho_2') \geq \mathrm{len}(\rho_2), \rho_2^{\mathrm{top}}(\mathrm{term}) = \textbf{false}}{\langle (s_1 \S s_2),\sigma,(\rho_1 \S \rho_2)\rangle \to \langle (s_1 \S s_2'),\sigma',(\rho_1 \S \rho_2')\rangle}$$

$$\text{III}\ \frac{\langle s_1,\sigma,\rho_1\rangle \to \langle s_1',\sigma',\rho_1'\rangle, \mathrm{len}(\rho_1) > \mathrm{len}(\rho_2), \rho_2^{\mathrm{top}}(\mathrm{term}) = \textbf{true}}{\langle (s_1 \S \textbf{skip}),\sigma,(\rho_1 \S \rho_2)\rangle \to \langle (s_1' \S \textbf{skip}),\sigma',(\rho_1' \S \rho_2)\rangle}$$

$$\text{IV}\ \frac{\langle s_2,\sigma,\rho_2\rangle \to \langle s_2',\sigma',\rho_2'\rangle, \mathrm{len}(\rho_2) > \mathrm{len}(\rho_1), \rho_1^{\mathrm{top}}(\mathrm{term}) = \textbf{true}}{\langle (\textbf{skip} \S s_2),\sigma,(\rho_1 \S \rho_2)\rangle \to \langle (\textbf{skip} \S s_2'),\sigma',(\rho_1 \S \rho_2')\rangle}$$

$$\text{V}\ \frac{\mathrm{len}(\rho_1) = \mathrm{len}(\rho_2), \rho_1^{\mathrm{top}}(\mathrm{term}) = \textbf{true},\ \rho_2^{\mathrm{top}}(\mathrm{term}) = \textbf{true}, x_1(\textbf{cont}) = s_1, x_2(\textbf{cont}) = s_2}{\langle (\textbf{skip} \S \textbf{skip}),\sigma,((x_1|\rho_1) \S (x_2|\rho_2))\rangle \to \langle (s_1 \S s_2),\sigma,(\rho_1 \S \rho_2)\rangle}$$

Figure 5. Semantics of spliced execution context switching, depending on stack action and depth. Variable term is used to handle the return path. It is set to **false** in each stack frame during function invocation and set to **true** by the return statement.

$$\frac{\langle e_1,\sigma,\rho\rangle \to v_1,\ldots,\langle e_k,\sigma,\rho\rangle \to v_k,\quad \langle f,\sigma,\rho\rangle \to \lambda(p_1\ldots p_k).s,\quad \rho = (\rho^{\mathrm{top}}|\rho^{\mathrm{rest}}),\ \mathsf{IE}[\![f(e_1,\ldots,e_k)]\!](\sigma,\rho) = (r_1,w_1),\quad \mathsf{CE}[\![f(e_1,\ldots,e_k)]\!](\sigma,\rho) = (r_2,w_2)}{\langle f(e_1,\ldots,e_k),\sigma,\rho\rangle \to \langle s,\sigma,[p_1 \mapsto v_1,\ldots,p_k \mapsto v_k,\textbf{cont} \mapsto \textbf{this},\textbf{reff} \mapsto r_1,\textbf{weff} \mapsto w_1]|\rho^{\mathrm{top}}[\textbf{reff} \mapsto r_2,\textbf{weff} \mapsto w_2]|\rho^{\mathrm{rest}}\rangle}$$

Figure 6. Tracking effects in the stack.

Algorithm 1: Actions taken on the global tier when a steal occurs.

Input: v : the victim of a successful steal
t : the thief stealing from v
@steal(v, t) **begin**
  foreach *spliced function continuation stolen following f* **do**
    t.global_tier $= t$.global_tier $\cup$ IE$[\![f]\!](p_1,\ldots,p_k)$;
    v.global_tier $= v$.global_tier $\cup$ CE$[\![f]\!]$;

one task, and other worker threads begin execution in an idle state. An idle worker enters the *stealing phase* and attempts to steal work from a randomly chosen victim, repeating this process until it finds work. Upon finding work, a worker begins a *working phase*, which ends once its local double-ended queue (deque) of tasks is empty. Each worker threads is typically mapped to a distinct hardware thread.

Transforming the recursive program to Cilk. The inclusive effect system described in the context of a recursive program lends itself naturally to building a nested fork/join program automatically. Because inclusive effects for a function and continuation are available at compile time, we can determine at runtime if a sync is required between the function and following continuation by testing for interference between them. Thus, we optimistically insert a spawn for every function/continuation pair that has a data effect present and conditionally insert a sync.

At runtime, we branch on the result of this interference check, executing a sync when needed to inhibit parallel execution past this point. In this way, we ensure the transformed concurrent program executes correctly in parallel. The following describes the transformation being applied:

X$[\![f(p_1,\ldots,p_k)$ **Reads**(e_1) **Writes**(e_2) $ss]\!]\sigma =$
  cilk $f(p_1,\ldots,p_k)$ /*switch to next thread, if available*/ X$[\![ss]\!]\sigma$
X$[\![f(e_1,\ldots,e_k)]\!]\sigma =$
  typeof(p_1) $p_1 = e_1;\ldots;$ typeof(p_k) $p_k = e_k$
  f_eff $=$ IE$[\![f]\!](p_1,\ldots,p_k)$ /*evaluated in this context*/
  c_eff $=$ CE$[\![f]\!]$/*evaluated in calling context*/
  spawn $f(p_1,\ldots,p_k)$
  if effects_interfere(f_eff,c_eff) **then sync else skip**
X$[\![pr_s s]\!]\sigma =$
  s_eff $=$ SE$[\![s]\!]\sigma$ /*evaluated in this context*/
  c_eff $=$ CE$[\![f]\!]\sigma$ /*evaluated in this context*/
  if /*step effect conflicts*/ **then**
    /*suspend this stack frame*/
  else
    X$[\![s]\!]\sigma$

Extending Cilk's runtime to manage lightweight threads. Each Cilk worker thread (a hardware thread) manages multiple lightweight threads. Within a Cilk worker thread, ex-

ecution follows the semantics in Figure 5. Steal operations take a frame from each lightweight thread in the victim's Cilk worker thread. Suspending execution suspends frames of all lightweight threads in the Cilk worker thread. Resuming execution initializes the stacks of the lightweight threads in a Cilk worker thread with the suspended frames and resumes execution of all lightweight threads. Using the stack semantics and ordering of push/pop operations in Figure 5, we ensure that before it is stolen from, all lightweight threads at a victim worker thread have a non-executing bottom frame. The Cilk worker deque is mostly unchanged, except for modifying the visibility of the frames pushed into the deque by a leading thread (to incoming thieves) until all spliced stacks reach that depth. Thus, stack management during spliced execution does not impose extra synchronization beyond traditional work stealing.

Tracking global dependences. The dependence structure discussed thus far only considers local dependences across spliced invocations, which is sufficient when the program is interleaved sequentially. The same local dependence checks are sufficient in a working phase. However, in the presence of steals, global dependences must be maintained between working phases to ensure correct execution.

To ensure non-interference across steps in distinct working phases, a thief must determine which global conflicts may exist when a steal occurs. The effects that may interfere are the set of effects at each spawn preceding the set of stolen continuations for each spliced thread. Therefore, the thief copies the effect set at each spawn for every user-level thread and checks for interference with the spawn's effect before executing any step.

This leaves the victim's effect object in a read-only state. During the execution of a working phase, the victim accesses the stolen continuation's effects (through its read-only copy) and checks for interference without incurring extra synchronization between the victim and the thief. This check is conservative: the execution since the steal operation might have satisfied an interfering effect from a stolen frame. However, this will not be visible to the victim, which only checks its (possibly out-of-date) read-only copy of the frame's effect.

When future thieves successfully steal from the victim, they add to their global conflict list the set of previously stolen continuations from the victim's working phase to ensure non-interference between the previous thieves. Using this protocol, the global conflict set is propagated between distinct working phases as steals occur.

7. Optimizations and Discussions

Thus far, we have presented the key aspects of effect-directed runtime scheduling and parallel execution to optimize locality. In this section, we discuss further optimizations employed and constraints imposed in our implementation to lower overheads and make the approach useful in practice. Also, we discuss the overheads involved and other limitations.

Lightweight user-level threads. The splicing scheduler splices function invocations using lightweight user-level threads that are mapped to the same hardware thread. The threads are not visible to the operating system, always execute on the same core, and share all levels of the cache. This ensures that the data reuse between the spliced threads leads to improved cache misses.

Splicing multiple threads. At the top-level when splicing is initiated by the runtime, multiple function invocations often will be spliced to maximize cache locality. When multiple function invocations are spliced using distinct user-level threads, the threads are ordered based on the function invocation order. To preserve dependences, each thread's execution must check for interference with all preceding threads. This can lead to quadratic dependence checking costs. Where possible, we reduce this overhead by exploiting transitive dependences between preceding threads. For example, if an effect is equivalent or a subset of a preceding thread's effect, it is sufficient to wait until that preceding thread's effect is ready to execute it instead of searching and waiting on all dependences in every preceding thread. We discover transitive dependences by searching for equivalent or subset effects in preceding threads at the same level of the stack. If we find such an effect, we wait for it to be ready to execute instead of searching all preceding threads. For benchmarks that iteratively execute and often have a symmetric effect pattern, we find this optimization significantly decreases the overhead of tracking dependences. For this optimization to be available to the runtime, the user must supply an equivalence-subset operator for effects.

Checking dependences with preceding threads. The execution semantics shows a step in a trailing thread step checking the effect of every frame in all preceding threads for potential conflicts. While transitive effects may reduce this cost to effectively checking a single preceding thread, this cost can still be substantial.

In the implementation, we reduce this cost by *refining* the dependences as execution progresses. Each stack frame tracks the oldest frame f_p, in each preceding thread p, that it conflicts with (based on the effect in the frame). Subsequent dependence checks can safely ignore frames older than f_p.

Instead of waiting until we encounter a step, we reduce the cost of this check by incrementally checking for interference each time we encounter a function invocation during execution. On encountering a function invocation, we update this dependence information to reflect the modified effects corresponding to the continuation.

Dependence evaluation for delayed steps. The second dependence optimization is for locating delayed work in preceding threads that may interfere. Instead of searching through a list of delayed steps for a given thread, we re-

cursively walk the tree of live stack frames that lead to the steps. Any live delayed step must be reachable from the root where splicing started. Exploiting the inclusive nature of the effect system, we search all paths from the root recursively, eliminating paths where the effect at that subtree does not interfere with the step being checked. This optimization increases performance substantially compared to the naive approach of searching through lists of delayed steps.

Bootstrapping spliced execution. Thus far, we have focused on the execution under spliced mode. However, program execution begins in unspliced mode. Splicing is beneficial only when it can exploit cache reuse between the spliced threads. In this work, we exploit user annotations to identify such opportunities. The user annotation TrySplice(n) directs the runtime that the following n function invocations need to be spliced. When these functions return, execution returns to unspliced mode.

Step pipelining. An executable step in the computation may encompass a large amount of sequential work (and cache footprint) that could benefit from being interleaved with other steps in different threads. Because we are not modifying or analyzing the sequential code in a step, we can not interleave it with other steps without help from the user. To enable step interleaving, we allow the user to write a slice operator for an effect, which constructs a pair of partial effects that can be executed distinctly. For this optimization to be applied, the user must also write a parametric variant of the step that takes an effect as input and performs the corresponding computation. When the runtime encounters a parametrically defined step with a sliceable effect, it invokes the slice operator on the effect and applies the first partial effect. Then, it context switches to the next user-level thread. If there is a matching parametric step, the runtime slices it and only executes the partial effect immediately when there is no interference. This pattern continues for all threads, enabling a tight interleaving of matching steps across user-level threads. The system iteratively calls the slice operator until slicing is not possible, or the effect is empty. By defining the slice operator, the user controls the granularity of the resulting partial step executed by the system.

Deadlock freedom. When dependences prevent a trailing thread step from being executed, it is delayed and placed on the heap. Even under spliced execution, the leading thread is never stalled from making progress. In general, a thread is never stalled due to actions relating to other trailing threads. This ensures deadlock freedom.

Implications on Cilk scheduler's provable optimality. Dynamic splicing changes execution order compared to the explicitly annotated Cilk program. In particular, splicing trades off some space and time compared to an explicit Cilk program to improve data reuse. Simultaneously maintaining multiple thread stacks increases space bound by a factor proportional to the number of phases interleaved. In the worst case, a significant number of steps in the trailing threads might be delayed and placed on the heap, increasing the space overheads. However, the space overheads stemming from delayed steps can be tracked at runtime. When it exceeds a specified bound, the interleaved phases can be "unspliced" to execute in a non-interleaved fashion, preserving the required space bound. This exposes a trade-off between interleaved execution and associated space overheads.

Algorithm 2: Pipelined execution of a parametric step with slicing.

```
1  if s is parametric ∧ s_step is sliceable then
2      s_next ← s_step;
3      repeat
4          (s_cur, s_next) ← slice(s_next);
5          if s_cur does not interfere then
6              parametric_fn(s_step)(s_cur);
7              context switch to next ult;
8          else
9              delay s_cur ∧ s_next to the heap;
10             context switch to next ult;
11             break;
12     until s_next = ∅;
```

In addition, the spliced execution increases cache footprint in terms of additional stacks for the lightweight threads and potentially increased footprint when, despite reuse, the total data accessed by the spliced threads is greater than the that accessed by any individual thread. This can impact optimal tile size choices and potentially increase total cache misses incurred.

There are two aspects to splicing's impact on Cilk's time bound:

- The instructions to perform the runtime checks increase total work to be performed (T_1) and critical path length (T_∞)
- Splicing never stalls the leading thread or precludes frames from being available for a steal. Therefore, splicing dependent phases (separated by a sync), does not affect T_∞. Splicing concurrent phases interleaves concurrent work in a serial fashion, potentially increasing T_∞ by a multiplicative factor equal to the number of phases interleaved.

Limitations of the effect system. Accurately computing the continuation effect can be difficult in the presence of function pointers, etc. Effects of continuations with function invocations can also be computed by evaluating the function arguments, which can require executing the intervening steps. In these cases, splicing can benefit from explicit user specification of continuation effects. Steps and functions without effect annotations (and cannot be easily inferred) will be treated as affecting all state. Our approach relies on deterministic execution within (to construct effects

in advance) and across phases (to splice them for locality). E.g.,a sequence of very different graph traversals (e.g., starting from a different vertex) or graph update traversals using atomics/locks cannot be optimized through splicing. On the other hand, that effects are constructed at runtime using application state enables splicing beyond the scope of compile-time techniques.

8. Experimental Evaluation

All experiments are executed on an eight-socket system with 1 TB DRAM, comprised of eight 10-core 2.27 GHz Intel Xeon E7-8860 processors. Our splicing scheduler is implemented in MIT Cilk 5.4.6, and all generated codes are compiled with GCC 5.2.0. The lightweight threads are implemented using the Argobots lightweight threading framework [39]. To comparatively evaluate our splicing scheduler, we use MIT Cilk and two compiler frameworks, Pochoir [41] and Pluto [31]. Pluto and Pochoir have been shown to generate highly optimized implementations based on extensive prior research [4, 41]. Thus, we have employed benchmarks that are statically analyzable, regular, and suitable for optimization by these compiler suites.

Benchmarks evaluated. Table 1 includes the evaluated benchmarks and configurations, which were taken from the Polybench [32] and Pochoir benchmark suites. Each benchmark has multiple phases that can benefit from data reuse. Six of the benchmarks are stencils (JACOBI-1D, JACOBI-2D, JACOBI-3D, FDTD-2D, SEIDEL-2D, and APOP) that have significant data reuse potential between multiple iterations (often referred to as *time tiling*). The other benchmarks, MVT and BICG, have two distinct phases that can benefit from interleaved execution. We evaluated the six stencil benchmarks with both the Pochoir and Pluto compilers. The MVT and BICG benchmarks cannot be written in Pochoir because it exclusively expresses stencil patterns. For the stencil benchmarks, we selected time tile sizes that resulted in the best performance for each scheduler evaluated.

Evaluation with Pluto. Pluto is a polyhedral source-to-source compiler transformation framework for affine loop nests that optimizes locality and automatically parallelizes loop nests. Pluto uses deep, static dependence analysis to time-tile and parallelize affine loop nests by generating C code annotated with OpenMP pragmas. For applicable loops, it can generate highly optimized parallel implementations that rival other locality optimizing frameworks and the best production compilers. To evaluate our runtime approach, we compared it against the latest version of Pluto [31] that uses *diamond* tiling (on the pet branch). For the codes tested, the Pluto-generated code naturally achieves good non-uniform memory access (NUMA) locality because of the generated "for-loop" matched structure. We compiled an analyzable for-loop version of all the codes in the Pluto transformation framework with `--parallelize`, `--partlbtile`, and `--tile`. These options are mentioned as obtaining the best performance in a recent work by Pluto's authors [5]. For all of the benchmarks, we tested several tile sizes (for L1 and L2 cache) to find the best-performing configurations. Table 1 shows the tile sizes we selected for Pluto.

Evaluation with Pochoir. Pochoir [41] is a compiler and runtime framework that transforms a stencil-specific domain-specific language (DSL) embedded in C++ code to a concurrent multicore execution using Cilk. The underlying Pochoir algorithm creates a cache-oblivious parallelization of time-stepped multidimensional grids using "hyperspace cuts," an asymptotic improvement over trapezoidal decompositions. Thus, Pochoir does not require any user input on tiling or NUMA placement. Because of the limitations in compiling Pochoir-generated code using GCC, we evaluate a version of each benchmark written in Pochoir, backed with the Intel ICPC compiler, version 14.0.3.4[4]. We use the following flags for compiling the Pochoir codes: `-O3 -funroll-loops -fno-alias -fno-fnalias -fp-model precise`.

Evaluation with splicing scheduler: **SP.** We implement each benchmark in recursive form with the data effect annotations. For the stencils, we provide a hint to the splicing scheduler at the top-level time-stepping loop to attempt splicing the subsequent T_s time steps (a tunable parameter). For each stencil, we evaluate several block sizes and values for T_s to find the best-performing block and time tile sizes (presented in Table 1). To obtain the best performance for the stencil benchmarks, we implement a parametric variant of the kernel that executes a partial effect, allowing the runtime to pipeline the kernel's execution. We have found that without this optimization, the very small block sizes required to obtain cache reuse at high levels of the cache (L1/L2) are prohibitively expensive because of runtime effect management and Cilk overheads. For the non-stencil benchmarks (MVT and BICG), we offer a similar hint to the splicing scheduler immediately preceding the two phases of each benchmark.

Evaluation with Cilk: **CILK.** For each benchmark, we evaluate the version equivalent to the Cilk code generated from the effect system. All effect-related actions are removed, and the code is executed as a conventional Cilk program.

Evaluation with NUMA-optimized Cilk: **CILKN.** In past work, Cilk and the underlying random work-stealing scheduling algorithm have been shown to cause performance degradations in the NUMA contexts when the data accessed in a task are not executed where the page is mapped by the system. For instance, in the first-touch policy, if the data initialization (thus, the page allocation) occurs in a different

[4] We compiled Pochoir programs on a system with Intel compilers (but fewer cores) and executed the binary on this system.

Benchmark	Problem Size	Serial Time Per Iter. (s)	Cilk, CilkN, Sp, SpN T_s	Block Size	PL	Pluto Timesteps	L1 Block Size	L2 Block Size	Pochoir Timesteps
JACOBI-1D	268435456	1.1 ± 0.01	16	1048576	✓	16384	512×512		16384
JACOBI-2D	32768^2	4.7 ± 0.03	16	1024×4096	✓	128	$128 \times 128 \times 128$	$8 \times 8 \times 8$	256
JACOBI-3D	1024^3	7.9 ± 0.05	8	$64 \times 64 \times 1024$	✓	128		$16^3 \times 256$	64
SEIDEL-2D	32768^2	8.0 ± 0.20	8	512×8192	✓	128	$128 \times 128 \times 128$	$8 \times 8 \times 8$	256
FDTD-2D	16384^2	3.8 ± 0.05	8	512×4096	✓	128	$128 \times 128 \times 128$	$8 \times 8 \times 8$	256
APOP	268435456	2.6 ± 0.03	16	1048576	✓	16384	512×512		16384
MVT	16384^2	1.7 ± 0.02	—	32×1024		—	$8 \times 128 \times 8$	$8 \times 2 \times 8$	—
BICG	32768^2	7.0 ± 0.07	—	32×1024		—	$8 \times 128 \times 8$	$8 \times 2 \times 8$	—

Table 1. Benchmark configurations with best tile/block size, T_s (spliced time steps), and PL (pipelining optimization) selected.

Domain	Arithmetic Mean SpN	CilkN	Pluto	Pochoir	Cilk	Geometric Mean SpN	CilkN	Pluto	Pochoir	Cilk
Cores 1-8 (within NUMA)	1.01	1.96	0.85	1.23	1.94	1.01	2.11	0.87	1.31	2.10
Cores 16-64 (across NUMA)	0.68	1.57	0.49	0.62	2.76	0.71	1.82	0.68	0.68	3.08

Table 2. Overview of trends across all benchmarks. Each number is the arithmetic or geometric mean of execution time ratios for all benchmarks between x, the column header, and the splicing (Sp) scheduler. For example, the first column is SpN/Sp.

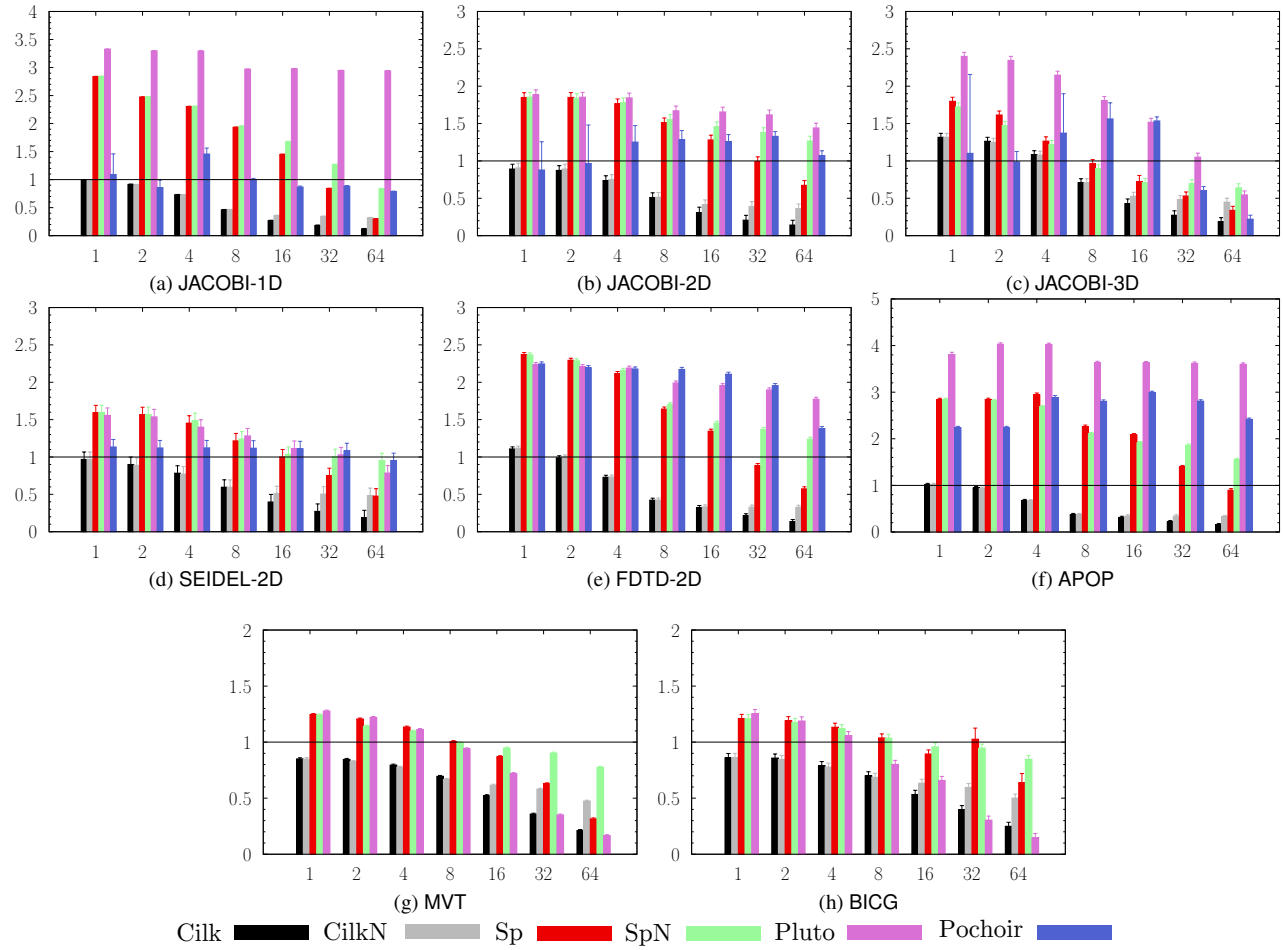


Figure 7. Mean speedup of five runs achieved over a hypothetical perfectly scaled serial implementation run on a single core ($\frac{\text{mean execution time}}{\text{serialtime}/c}$, where c is the number of cores). The Pluto compiler time tiles and parallelizes the codes with OpenMP. The Cilk and CilkN bars correspond to using Cilk without and with NUMA optimizations. The Sp and SpN bars correspond to using the splicing scheduler without and with NUMA optimizations. x-axis—number of cores; y-axis—speedup; error bars—standard deviation.

memory domain than use, the task time may be inflated. By using a random work-stealing scheduler, the locale of page allocation and subsequent use is often mismatched. Hence, when used in conjunction with random work stealing, our splicing algorithm suffers from these effects. In work by Lifflander et al. [23], the authors demonstrate a variant of Cilk augmented with NUMA optimizations that performs comparable to OpenMP for stencil-like benchmarks. We also consider this version for evaluation.

***NUMA-optimized splicing scheduler:* SPN.** To reap the NUMA benefits from CILKN, we have built a version of our splicing scheduler on top of their scheduling algorithm: *relaxed work stealing*.

Experimental speedup comparison. In Figure 7, we compare the speedup achieved by the various scheduling and locality optimizers for the benchmarks evaluated. The speedup is calculated relative to serial execution time scaled by the number of cores. Each benchmark is written with standard for-loops and compiled with GCC 5.2.0 using `-O3`. Table 1 lists the serial timings. Except for Pochoir, all generated codes are compiled with the same version of GCC. Each data point on the graphs is the arithmetic mean of five executions, and the error bars show the standard deviation.

On the speedup graphs, the CILK bars represent the speedup achieved with the work-stealing Cilk scheduler. For these benchmarks, the scaling is limited by memory bandwidth and NUMA locality. The CILKN bars show the speedup achieved by incorporating NUMA optimizations [23]. At 16 cores, beyond a single memory domain, the scaling significantly improves with CILKN.

The SPN and SP bars show the speedup obtained using the splicing scheduler with and without NUMA optimization, respectively, by interleaving T_s time steps (shown in Table 1). Across all benchmarks and scales, the splicing scheduler significantly outperforms native Cilk code. Compared to CILK, SP performs around $2\times$–$3\times$ better overall. To summarize the relative improvement, Table 2 presents the mean speedup over all the benchmarks, delineated by the NUMA regime. By incorporating NUMA optimizations, performance is further improved beyond eight cores. Compared to CILK beyond eight cores, SPN executes $4.1\times$ faster (arithmetic mean over the benchmarks as shown in Table 2). Compared to the NUMA-optimized Cilk, SPN executes $2.4\times$ faster beyond eight cores.

Across the benchmarks, our splicing scheduler is competitive with Pluto and Pochoir. Pluto and Pochoir extract dependences at compile time through static polyhedral analysis and DSL representation, respectively. Although we pay the runtime cost of managing dependences, context switching between user-level threads, and postponing tasks, we still significantly improve the native Cilk program's performance. In general, Pluto and Pochoir extract reuse at all levels of cache, including the L1 cache. Because of the overheads in executing a user-defined kernel, our scheduling mechanism cannot optimize as well for L1 cache reuse. Without compile-time transformations, the cost of a function call for each kernel limits the reuse that is possible without incurring prohibitive overheads. For the one-dimensional APOP benchmark in particular, Pluto performs better due to tight fusion of the kernel that is not limited by complex boundary conditions.

The MVT and BICG benchmarks involve a reduction-like operation from a matrix to a vector. Thus, Pluto cannot generate an efficient parallel implementation in OpenMP because it does not detect and optimize for the reduction-like pattern. The effects of this limitation can be observed in Figure 7g and 7h: the performance decreases with scale. For our recursive versions, we use an accumulator to combine results into the vector. Thus, the two phases in MVT and BICG can be spliced without delaying work and decreasing concurrency. This implementation accrues locality benefits and scales much further than Pluto.

Hardware performance counters. To further explain the the splicing scheduler's behavior, Table 3 presents the cache and translation lookaside buffer (TLB) miss rates and instruction counts using hardware counters accessed through the Linux kernel API provided with `perf`. Compared to the serial implementation, the instruction counts are higher for all locality-optimized codes. Pluto increases the instruction count because of the conditionals and min/max operations it performs for tiling and parallelization. Pochoir's runtime performs complex hyperspace cuts, which significantly increases the instructions executed. Compared to Cilk and serial miss rates, the splicing scheduler reduces L2 and L3 misses, which correlate strongly with the performance improvement. The spliced version significantly reduces the L2 and L3 misses compared to Cilk. These factors demonstrate that splicing performance improvements are correlated with better memory hierarchy reuse.

Overheads. Table 4 presents statistics on the splicing scheduler's operations to concurrently interleave while ensuring correctness on 64 cores. For JACOBI-2D, JACOBI-3D, SEIDEL-2D, and FDTD-2D, the system performs tens of millions interference checks to splice the program and execute it concurrently. Because JACOBI-1D and APOP are one-dimensional with simple boundary conditions, they require fewer interference checks. The two phases spliced for MVT and BICG are effectively concurrent and do not pause steps or have any dependences. Overall, a large number of runtime operations are required to extract cache locality and parallelism. As such, efficient effect description and runtime design are required to obtain high performance.

Table 4 also includes an upper bound on the amount of extra space required to store delayed steps during spliced execution. This is computed as the product of the size of the largest Cilk frame and the maximum number of live Cilk stack frames at any point. In general, the number of extra stack frames allocated will be proportional to the number

Config	JACOBI-1D							JACOBI-2D						
	L1		L3		TLB		Ins.	L1		L3		TLB		Ins.
	Acc.	Miss	Acc.	Miss	Acc.	Miss	Count	Acc.	Miss	Acc.	Miss	Acc.	Miss	Count
Splice	0.2B	38.8M	18.2M	9.1M	134M	0.35M	0.5B	1.6B	270M	18M	4M	100M	2M	2.9B
Cilk	0.2B	41.4M	3.9M	0.58M	138M	0.10M	0.5B	1.4B	260M	68M	33M	910M	1.6M	2.7B
Serial	0.2B	38.8M	18.5M	9.2M	134M	0.35M	0.4B	1.2B	260M	75M	36M	780M	1.4M	2.3B
Pluto	0.2B	0.11M	19K	10K	146M	1.3K	0.6B	1.4B	180M	58M	0.51M	870M	6.7M	3.4B
Pochoir	0.4B	0.11M	3K	0.4K	271M	0.09K	0.9B	2.1B	210M	92M	0.24M	1.3B	18M	4.4B
Config	JACOBI-3D							SEIDEL-2D						
	L1		L3		TLB		Ins.	L1		L3		TLB		Ins.
	Acc.	Miss	Acc.	Miss	Acc.	Miss	Count	Acc.	Miss	Acc.	Miss	Acc.	Miss	Count
Splice	2.6B	372M	30M	8.4M	1.59B	3.6M	4.7B	2.1B	330M	24M	3.1M	1360M	1.8M	11B
Cilk	2.5B	349M	68M	32.5M	1.53B	1.8M	4.6B	2.1B	320M	52M	24M	1320M	1.4M	9.1B
Serial	1.7B	348M	112M	56.2M	1.05B	2.7M	3.4B	1.7B	310M	82M	37M	1100M	1.3M	4.54B
Pluto	1.8B	292M	43M	3.2M	1.13B	4.8M	4.5B	2.8B	84M	50M	1.7M	1830M	15M	6.1B
Pochoir	3.2B	312M	18M	1.9M	1.95B	5.4B	5.9B	2.1B	420M	180M	0.48M	1400M	34M	14B
Config	FDTD-2D							APOP						
	L1		L3		TLB		Ins.	L1		L3		TLB		Ins.
	Acc.	Miss	Acc.	Miss	Acc.	Miss	Count	Acc.	Miss	Acc.	Miss	Acc.	Miss	Count
Splice	1.0B	110M	10M	2.9M	630M	1.3M	1.92B	0.38B	83M	10M	2.0M	240M	0.98M	1.1B
Cilk	0.93B	110M	46M	23M	590M	1.1M	1.8B	0.36B	80M	32M	16M	230M	0.88M	1.08B
Serial	0.72B	120M	63M	31M	460M	1.3M	1.55B	0.36B	74M	32M	16M	230M	3.4M	1.03B
Pluto	1.4B	54M	29M	0.35M	910M	4.5M	3.5B	0.47B	2.1M	0.11M	55K	300M	3.9M	1.5B
Pochoir	1.5B	230M	120M	0.60M	950M	21M	3.0B	0.70B	0.39M	0.01M	1K	430M	2K	1.8B
Config	MVT							BICG						
	L1		L3		TLB		Ins.	L1		L3		TLB		Ins.
	Acc.	Miss	Acc.	Miss	Acc.	Miss	Count	Acc.	Miss	Acc.	Miss	Acc.	Misss	Count
Splice	0.63B	27.5M	9.7M	3.0M	480M	0.5M	1.8B	2.9B	110M	33M	13M	1920M	1.9M	7.1B
Cilk	0.73B	27.5M	13M	6.3M	480M	0.4M	1.8B	2.9B	110M	52M	25M	1920M	1.7M	7.1B
Serial	0.36B	52M	20M	9.1M	230M	0.3M	1.0B	1.45B	210M	82M	38M	910M	1.4M	4.13B
Pluto	0.42B	184M	100M	9.0M	250M	84M	1.5B	1.7B	920M	410M	38M	1000M	340M	5.8B

Table 3. Cache, TLB, and instruction count instrumentation on one core.

of iterations spliced together and the number of conflicts. Beyond this, the size of each delayed step corresponds to the amount of state (live variables) on the stack that must be saved on the heap to be executed when ready. For the benchmark evaluated, we find that the space overheads stemming from delayed steps are low: the maximum extra space required is 16 MB on 64 cores for the JACOBI-2D benchmark. MVT and BICG do not require extra space because the spliced phases are concurrent and, thus, never delayed.

9. Related Work

Compile-time analysis and transformation for fork/join programs. Burstall and Darlington present a system of rules for transforming recursive programs [8]. Nandivada et al. [27] introduce a transformation framework to optimize exposed concurrency in task-parallel programs. Neither work takes data locality into account. The work by Rugina et al. [36–38] on static analysis of pointer and array index references in recursive programs can aid the construction of the effects used in the work described in this paper.

DSLs and library composition. DSLs allow information related to data locality and dependences to be expressed at a higher level of abstraction, enabling aggressive optimizations [40]. Pochoir [41] is an example stencil DSL that exploits language-level information to generate optimized stencil programs in Cilk. Chandra et al. [10] present the Cool language, an extension of C++ for task-parallel programs, that allows specification of affinity and migration hints to the runtime scheduler, effectively an approach to user-controlled NUMA-like locality. Active libraries exploit domain information and associated optimizations without the need for distinct language syntax [45]. Just as in DSLs, active libraries generate optimized code based on transformation of the source code being analyzed. Using library annotations to optimize across libraries without requiring their source code also has been extensively studied [17, 21, 33]. Unlike our runtime approach, these methods typically exploit the library annotations to generate optimized implementations at compile time.

Benchmark (64 cores)	Interference Checks	Context Switches	Delayed Steps	Total Deps.	Space Overhead (1–64 cores, in MB) 1	2	4	8	16	32	64
JACOBI-1D	220,849	2,029,776	7,438	20,244	0.4	1	1	2	4	6	10
JACOBI-2D	69,753,410	3,960,272	15,102	66,414	0.8	2	4	7	12	14	16
JACOBI-3D	41,675,491	11,890,664	10,752	55,680	0.5	2	3	5	6	7	9
SEIDEL-2D	59,253,018	2,022,304	30,076	132,712	0.5	1	4	4	8	13	16
FDTD-2D	31,047,246	1,998,240	15,100	48,528	0.5	0.9	2	3	5	7	10
APOP	216,225	2,032,848	7,392	19,584	0.2	0.4	0.8	2	3	5	8
MVT	262,124	98,870	0	0	0	0	0	0	0	0	0
BICG	1,048,556	393,782	0	0	0	0	0	0	0	0	0

Table 4. Runtime statistics using our splicing scheduler.

Runtime locality-aware scheduling of fork/join programs. Our scheduling strategy—explore the continuation of an invoked function for splicing—is similar to the help-first scheduling strategy presented by Guo et al. [15]. Their work focuses on exposing additional parallelism for load balancing rather than optimizing data locality. Lifflander et al. optimize fork/join programs for NUMA locality, but not cache locality, by constraining the work-stealing scheduler [23]. Parallel depth-first schedulers are designed to enable constructive cache sharing among concurrent tasks in fork/join programs [11]. Meanwhile, Guo et al. present scalable locality-aware work stealing, or SLAW, to schedule tasks to places based on locality hints [16]. These runtime approaches optimize locality within a phase, represented by a single recursive function or task.

Scheduling based on effect information. TWEJava [18] exploits effect information to extract deterministic parallelism. Legion [2, 44] and Habanero Java (HJp) [46] exploit information on memory accesses by tasks to ensure deterministic parallelism. These approaches focus on automatic extraction of parallelism and determinism guarantees, but they do not tackle data locality. Specifically, Legion [2] does not attempt to refine the dependences to the finest possible granularity to minimize runtime costs, which is required for cache locality optimization. Pan and Pai [29] employ a hardware-software approach to managing last-level cache for input-annotated task-parallel programs using the OmpSs task-parallel programming model. Jo and Kulkarni [19] exploit access information to schedule concurrent threads operating on shared data to optimize inter-thread data locality for tree traversals. Deterministic Parallel Java (DPJ) exploits information on memory accesses by tasks to extract deterministic parallelism [20], but it does not use effects to increase memory locality. Philbin et al. [30] is most related to our work. They extract sub-computations in a sequential program into parallel threads, which are then executed in an order that minimizes cache misses. However, their strategy does not tackle dependences, does not consider interleaved execution of the threads to further reduce cache misses, cannot handle recursive programs, and does not interoperate with a dynamic parallel scheduler such as Cilk.

Thread/task management. The splicing optimization can be implemented using various user-level thread libraries that support lightweight context switching (e.g., Marsh et al. [24], Qthreads [47], or Boost [6]). Our management of dependences involving delayed steps is similar to various task graph schedulers (e.g., Nabbit [1], CnC [7], Supermatrix [9], and X10 [42]).

10. Conclusions

Compile-time optimization across function boundaries is inherently limited by interprocedural analysis and the availability of enough information—source code, effects, or higher-level semantics—to enable such transformations. We have demonstrated an approach to cache locality optimization across invocations of effect-annotated recursive functions. This includes efficiently tracking effects, preventing dependence violations, and using lightweight-thread-based scheduling to interleave execution of function invocations. We demonstrate significant performance improvements, competitive with the best alternative optimization strategies. In particular, we present splicing across time iterations in stencil computations to mimic the complex diamond-tiling transformation in Pluto and time tiling in Pochoir.

Achieving the best performance with any strategy—traditional compiler analysis, compilation from domain-specific languages, or runtime optimization—requires careful tuning of a large parameter space. Our results cannot be used to claim any strategy as being definitively superior across the benchmarks and configurations considered. The alternative strategies considered can reason in a richer dependence space and perform a greater number of transformations (permutation, reversal, tiling, duplicated execution, etc.) not considered in this work. While these strategies often can handle more general dependence patterns, our approach works best with localized dependences, where any given task in the trailing phase depends on a small number of tasks in the leading phase. This enables efficient splicing with a limited number of delayed steps. When the number of dependences to be tracked increases with problem size, dependence tracking can incur significant cost. We consider our approach a complement to compile-time optimizers, meant to be used when such optimizers cannot be applied.

Acknowledgments

We thank the PLDI'17 reviewers for their extensive feedback and suggestions, which helped improve the paper. This work was supported in part by U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under award number 64386. Pacific Northwest National Laboratory is operated by Battelle for DOE under Contract DE-AC05-76RL01830.

References

[1] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pages 1–12, 2010.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *SC Conference on High Performance Computing Networking, Storage and Analysis*, page 66, 2012.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 207–216, 1995.

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 101–113, 2008.

[5] U. Bondhugula, V. Bandishti, and I. Pananilath. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298, May 2017.

[6] Boost Context. Boost Context. `http://www.boost.org/doc/libs/1_56_0/libs/context/doc/html/index.html`.

[7] Z. Budimlic, M. G. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. M. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar. Concurrent collections. *Scientific Programming*, 18(3-4):203–217, 2010.

[8] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1): 44–67, 1977.

[9] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. A. van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, 2007.

[10] R. Chandra, A. Gupta, and J. L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 249–259, 1993.

[11] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *SPAA: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 105–115, 2007.

[12] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[13] J. S. Danaher, I. A. Lee, and C. E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, 2006.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[15] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2009.

[16] Y. Guo, Y. Zhao, V. Cavé, and V. Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multicore systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 341–342, 2010.

[17] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of the Second Conference on Domain-Specific Languages (DSL)*, pages 39–52, 1999.

[18] S. Heumann, V. S. Adve, and S. Wang. The tasks with effects model for safe concurrency. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 239–250, 2013.

[19] Y. Jo and M. Kulkarni. Enhancing locality for recursive traversals of recursive structures. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 463–482, 2011.

[20] R. L. B. Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 535–548, 2011.

[21] K. Kennedy, B. Broom, K. D. Cooper, J. Dongarra, R. J. Fowler, D. Gannon, S. L. Johnsson, J. M. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel Distribued Computing*, 61(12):1803–1826, 2001.

[22] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 36–43, 2000.

[23] J. Lifflander, S. Krishnamoorthy, and L. V. Kalé. Optimizing data locality for fork/join programs using constrained work stealing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 857–868, 2014.

[24] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level theads. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP)*, pages 110–121, 1991.

[25] V. Maslov. Delinearization: An efficient way to break multi-loop dependence equations. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 152–161, 1992.

[26] MIT Cilk 5.4.6. MIT Cilk 5.4.6. `http://supertech.lcs.mit.edu/cilk`.

[27] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. A transformation framework for optimizing task-parallel programs. *ACM Transactions on Programming Languages and Systems*, 35(1):3:1–3:48, 2013.

[28] OpenMP Architecture Review Board. OpenMP Specification and Features. `http://openmp.org/wp/`, May 2008.

[29] A. Pan and V. Pai. Runtime-driven shared last-level cache management for task-parallel programs. Technical Report 466, Department of Electrical and Computer Engineering, Purdue University, 2015.

[30] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. In *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 60–71, 1996.

[31] Pluto – an automatic parallelizer and locality optimizer for affine loop nests. Pluto – an automatic parallelizer and locality optimizer for affine loop nests. `http://pluto-compiler.sourceforge.net`.

[32] L.-N. Pouchet. Polybench: The polyhedral benchmark suite, 2012.

[33] D. J. Quinlan, M. Schordan, Q. Yi, and A. Sæbjørnsen. Classification and utilization of abstractions for optimization. In *Leveraging Applications of Formal Methods, First International Symposium (ISoLA)*, pages 57–73, 2004.

[34] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. 2007.

[35] A. D. Robison. Composable parallel patterns with Intel Cilk Plus. *Computing in Science and Engineering*, 15(2):66–71, 2013.

[36] R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 72–83, 1999.

[37] R. Rugina and M. C. Rinard. Pointer analysis for structured parallel programs. *ACM Transactions on Programming Languages and Systems*, 25(1):70–116, 2003.

[38] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Transactions on Programming Languages and Systems*, 27(2): 185–235, 2005.

[39] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, A. Castello, D. Genet, T. Herault, P. Jindal, L. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, and P. H. Beckman. Argobots: a lightweight threading/tasking framework. Technical Report ANL/MCS-P5515-0116, Argonne National Laboratory, 2016.

[40] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP - Object-Oriented Programming - 27th European Conference*, pages 52–78, 2013.

[41] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 117–128, 2011.

[42] O. Tardieu, H. Wang, and H. Lin. A work-stealing scheduler for X10's task parallelism with suspension. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 267–276, 2012.

[43] TPL. The Task Parallel Library. `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`, Oct. 2007.

[44] S. Treichler, M. Bauer, and A. Aiken. Language support for dynamic, hierarchical data partitioning. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 495–514, 2013.

[45] T. L. Veldhuizen. *Active libraries and universal languages*. PhD thesis, Indiana University, 2004.

[46] E. M. Westbrook, J. Zhao, Z. Budimlic, and V. Sarkar. Permission regions for race-free parallelism. In *Runtime Verification - Second International Conference (RV)*, pages 94–109, 2011.

[47] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8, 2008.

[48] X10. The X10 Programming Language. `www.research.ibm.com/x10/`, Mar. 2006.

A. Additional Semantics on Effects

Figures 8, 9, 10 describe the inclusive, step, and continuation effects for the language in Figure 2. Effect annotations provided in the input program should be a superset of the effects constructed using this specification.

$$\mathsf{IE}[\![l]\!](\sigma,\rho) = (\emptyset,\emptyset) \qquad \mathsf{IE}[\![p]\!](\sigma,\rho) = (\emptyset,\emptyset) \qquad \mathsf{IE}[\![g]\!](\sigma,\rho) = (\{g\},\emptyset)$$

$$\mathsf{IE}[\![\textbf{return}]\!](\sigma,\rho) = (\emptyset,\emptyset) \qquad \mathsf{IE}[\![g := e]\!](\sigma,\rho) = (\mathsf{IE}[\![e]\!](\sigma,\rho),\{g\})$$

$$\mathsf{IE}[\![f_v(e_1,\ldots,e_k)]\!](\sigma,\rho) = (\mathsf{IE}[\![e_1]\!](\sigma,\rho) \cup \ldots \cup \mathsf{IE}[\![e_k]\!](\sigma,\rho),\emptyset) \qquad \mathsf{IE}[\![f_b(e_1,\ldots,e_k)]\!](\sigma,\rho) = (\mathsf{IE}[\![e_1]\!](\sigma,\rho) \cup \ldots \cup \mathsf{IE}[\![e_k]\!](\sigma,\rho),\emptyset)$$

$$\frac{\langle s_1,\sigma,\rho\rangle \to \langle \sigma_1,\rho_1\rangle}{\mathsf{IE}[\![s_1;s_2]\!](\sigma,\rho) = \mathsf{IE}[\![s_1]\!](\sigma,\rho) \cup \mathsf{IE}[\![s_2]\!](\sigma_1,\rho_1)}$$

$$[\text{if}_t]\frac{[\![e_b]\!](\sigma,\rho) = \textbf{true}}{\mathsf{IE}[\![\textbf{if } e_b \textbf{ then } s_1 \textbf{ else } s_2]\!](\sigma,\rho) = \mathsf{IE}[\![e_b]\!](\sigma,\rho) \cup \mathsf{IE}[\![s_1]\!](\sigma,\rho)} \qquad [\text{if}_f]\frac{[\![e_b]\!](\sigma,\rho) = \textbf{false}}{\mathsf{IE}[\![\textbf{if } e_b \textbf{ then } s_1 \textbf{ else } s_2]\!](\sigma,\rho) = \mathsf{IE}[\![e_b]\!](\sigma,\rho) \cup \mathsf{IE}[\![s_2]\!](\sigma,\rho)}$$

$$[\text{while}_t]\frac{[\![e_b]\!](\sigma,\rho) = \textbf{true}}{\mathsf{IE}[\![\textbf{while } (e_b)\ s]\!](\sigma,\rho) = \mathsf{IE}[\![e_b]\!](\sigma,\rho) \cup \mathsf{IE}[\![s;\textbf{while } (e_b)\ s]\!](\sigma,\rho)} \qquad [\text{while}_f]\frac{[\![e_b]\!](\sigma,\rho) = \textbf{false}}{\mathsf{IE}[\![\textbf{while } (e_b)\ s]\!](\sigma,\rho) = \mathsf{IE}[\![e_b]\!](\sigma,\rho)}$$

$$[\text{fn def}]\mathsf{IE}[\![f(p_1,\ldots,p_k)\ ss]\!](\sigma,[]) = (\emptyset,\emptyset) \qquad [\text{fn call}]\ \frac{\langle f,\sigma,\rho\rangle \to \lambda(p_1\ldots p_k).ss}{\mathsf{IE}[\![f(e_1,\ldots,e_k)]\!](\sigma,\rho) = \mathsf{IE}[\![ss]\!](\sigma,[p_1 \mapsto e_1,\ldots,p_k \mapsto e_k]|\rho)}$$

Figure 8. Inclusive effects for effect-annotated recursive programs. These are determined while ignoring the effect pragmas.

$$\langle \textbf{return},\sigma\rangle \to \sigma[term \mapsto \textbf{true}] \qquad \langle f(e_1,\ldots,e_k),\sigma\rangle \to \sigma[term \mapsto \textbf{true}]$$

$$\frac{\mathsf{IE}[\![l := e]\!](\sigma,\rho) = (\varphi_1,\varphi_2), \mathsf{SE}[\![\textbf{skip}]\!](\sigma,\rho,\kappa) = (\varphi_3,\varphi_4)}{\mathsf{SE}[\![l := e]\!](\sigma,\rho,\kappa) = (\varphi_1 \cup \varphi_3, \varphi_2 \cup \varphi_4)} \qquad \frac{\mathsf{IE}[\![g := e]\!](\sigma,\rho) = (\varphi_1,\varphi_2), \mathsf{SE}[\![\textbf{skip}]\!](\sigma,\rho,\kappa) = (\varphi_3,\varphi_4)}{\mathsf{SE}[\![g := e]\!](\sigma,\rho,\kappa) = (\varphi_1 \cup \varphi_3, \varphi_2 \cup \varphi_4)}$$

$$\frac{\langle s_1,\sigma,\rho\rangle \to (\sigma_1,\rho_1), \sigma_1(term) = \textbf{true}}{\mathsf{SE}[\![s_1;s_2]\!](\sigma,\rho,\kappa) = \mathsf{SE}[\![s_1]\!](\sigma,\rho,\textbf{null})}$$

$$\frac{\langle s_1,\sigma,\rho\rangle \to (\sigma_1,\rho_1), \sigma_1(term) = \textbf{false}, \mathsf{IE}[\![s_1]\!](\sigma,\rho) = (\varphi_1,\varphi_2), \mathsf{SE}[\![s2]\!](\sigma_1,\rho_1,\kappa) = (\varphi_3,\varphi_4)}{\mathsf{SE}[\![s_1;s_2]\!](\sigma,\rho,\kappa) = (\varphi_1 \cup \varphi_3, \varphi_2 \cup \varphi_4)}$$

$$[\text{if}_t]\frac{\mathsf{IE}[\![e_b]\!](\sigma,\rho) = (\varphi_1,\varphi_2), [\![e_b]\!](\sigma,\rho) = \textbf{true}, \mathsf{SE}[\![s_1]\!](\sigma,\rho,\kappa) = (\varphi_3,\varphi_4)}{\mathsf{SE}[\![\textbf{if } e_b \textbf{ then } s_1 \textbf{ else } s_2]\!](\sigma,\rho,\kappa) = (\varphi_1 \cup \varphi_3, \varphi_2 \cup \varphi_4)}$$

$$[\text{if}_f]\frac{\mathsf{IE}[\![e_b]\!](\sigma,\rho) = (\varphi_1,\varphi_2), [\![e_b]\!](\sigma,\rho) = \textbf{false}, \mathsf{SE}[\![s_2]\!](\sigma,\rho,\kappa) = (\varphi_3,\varphi_4)}{\mathsf{SE}[\![\textbf{if } e_b \textbf{ then } s_1 \textbf{ else } s_2]\!](\sigma,\rho,\kappa) = (\varphi_1 \cup \varphi_3, \varphi_2 \cup \varphi_4)}$$

$$[\text{while}_t]\frac{\mathsf{IE}[\![e_b]\!](\sigma,\rho) = (\varphi_1,\varphi_2), [\![e_b]\!](\sigma,\rho) = \textbf{true}, \mathsf{SE}[\![s;\textbf{while } (e_b)\ s]\!](\sigma,\rho,\kappa) = (\varphi_3,\varphi_4)}{\mathsf{SE}[\![\textbf{while } (e_b)\ s]\!](\sigma,\rho,\kappa) = (\varphi_1 \cup \varphi_3, \varphi_2 \cup \varphi_4)}$$

$$[\text{while}_f]\frac{\mathsf{IE}[\![e_b]\!](\sigma,\rho) = (\varphi_1,\varphi_2), [\![e_b]\!](\sigma,\rho) = \textbf{false}, \mathsf{SE}[\![\textbf{skip}]\!](\sigma,\rho,\kappa) = (\varphi_3,\varphi_4)}{\mathsf{SE}[\![\textbf{while } (e_b)\ s]\!](\sigma,\rho,\kappa) = (\varphi_1 \cup \varphi_3, \varphi_2 \cup \varphi_4)}$$

Figure 9. Step effects for effect-annotated recursive programs. These are determined while ignoring the effect pragmas. κ captures a statement's context in the form of its continuation.

$$\mathsf{CE}[\![pgm]\!]\sigma = (\emptyset,\emptyset) \qquad \frac{\langle s_1,\sigma,\rho\rangle \to \langle \sigma_1,\rho_1\rangle, \mathsf{IE}[\![s_2]\!](\sigma_1,\rho_1) = (\varphi_1,\varphi_2), \mathsf{CE}[\![s_1;s_2]\!](\sigma,\rho) = (\varphi_3,\varphi_4)}{\mathsf{CE}[\![s_1]\!](\sigma,\rho) = (\varphi_1 \cup \varphi_3, \varphi_2 \cup \varphi_4)}$$

$$\frac{\langle s_1,\sigma,\rho\rangle \to \langle \sigma_1,\rho_1\rangle, \mathsf{CE}[\![s_1;s_2]\!](\sigma,\rho) = (\varphi_3,\varphi_4)}{\mathsf{CE}[\![s_2]\!](\sigma_1,\rho_1) = (\varphi_3,\varphi_4)}$$

$$[\text{if}_t]\frac{\mathsf{CE}[\![\textbf{if } e_b \textbf{ then } s_1 \textbf{ else } s_2]\!](\sigma,\rho) = (\varphi_1,\varphi_2)}{\mathsf{CE}[\![s_1]\!](\sigma,\rho) = (\varphi_1,\varphi_2)} \qquad [\text{if}_f]\frac{\mathsf{CE}[\![\textbf{if } e_b \textbf{ then } s_1 \textbf{ else } s_2]\!](\sigma,\rho) = (\varphi_1,\varphi_2)}{\mathsf{CE}[\![s_2]\!](\sigma,\rho) = (\varphi_1,\varphi_2)}$$

$$[\text{while}_t]\frac{[\![e_b]\!](\sigma,\rho) = \textbf{true}, \mathsf{CE}[\![\textbf{while } (e_b)\ s]\!](\sigma,\rho) = (\varphi_1,\varphi_2), \langle s,\sigma,\rho\rangle \to (\sigma_1,\rho_1), \mathsf{IE}[\![\textbf{while } (e_b)\ s]\!](\sigma_1,\rho_1) = (\varphi_3,\varphi_4)}{\mathsf{CE}[\![s]\!](\sigma,\rho) = (\varphi_1 \cup \varphi_3, \varphi_2 \cup \varphi_4)}$$

$$[\text{while}_f]\frac{[\![e_b]\!](\sigma,\rho) = \textbf{false}, \mathsf{CE}[\![\textbf{while } (e_b)\ s]\!](\sigma,\rho) = (\varphi_1,\varphi_2)}{\mathsf{CE}[\![s]\!](\sigma,\rho) = (\varphi_1,\varphi_2)}$$

Figure 10. Continuation effects for effect-annotated recursive programs. These are determined while ignoring the effect pragmas.