Automatic Parallelism Management

SAM WESTRICK, Carnegie Mellon University, USA
MATTHEW FLUET, Rochester Institute of Technology, USA
MIKE RAINEY, Carnegie Mellon University, USA
UMUT A. ACAR, Carnegie Mellon University, USA

On any modern computer architecture today, parallelism comes with a modest cost, born from the creation and management of threads or tasks. Today, programmers battle this cost by manually optimizing/tuning their codes to minimize the cost of parallelism without harming its benefit, performance. This is a difficult battle: programmers must reason about architectural constant factors hidden behind layers of software abstractions, including thread schedulers and memory managers, and their impact on performance, also at scale. In languages that support higher-order functions, the battle hardens: higher order functions can make it difficult, if not impossible, to reason about the cost and benefits of parallelism.

Motivated by these challenges and the numerous advantages of high-level languages, we believe that it has become essential to manage parallelism automatically so as to minimize its cost and maximize its benefit. This is a challenging problem, even when considered on a case-by-case, application-specific basis. But if a solution were possible, then it could combine the many correctness benefits of high-level languages with performance by managing parallelism without the programmer effort needed to ensure performance. This paper proposes techniques for such automatic management of parallelism by combining static (compilation) and run-time techniques. Specifically, we consider the Parallel ML language with task parallelism, and describe a compiler pipeline that embeds "potential parallelism" directly into the call-stack and avoids the cost of task creation by default. We then pair this compilation pipeline with a run-time system that dynamically converts potential parallelism into actual parallel tasks. Together, the compiler and run-time system guarantee that the cost of parallelism remains low without losing its benefit. We prove that our techniques have no asymptotic impact on the work and span of parallel programs and thus preserve their asymptotic properties. We implement the proposed techniques by extending the MPL compiler for Parallel ML and show that it can eliminate the burden of manual optimization while delivering good practical performance.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; *Functional languages*; *Procedures, functions and subroutines*; **Compilers**.

Additional Key Words and Phrases: parallel programming languages, granularity control, compilers

ACM Reference Format:
Sam Westrick, Matthew Fluet, Mike Rainey, and Umut A. Acar. 2024. Automatic Parallelism Management. *Proc. ACM Program. Lang.* 8, POPL, Article 38 (January 2024), 32 pages. https://doi.org/10.1145/3632880

1 INTRODUCTION

Parallelism is here to stay. The numbers of cores in multicore chips have increased from a handful to many dozens in the last decade and brought parallelism to the mainstream. Yet, developing software that takes advantage of this parallelism continues to be a major challenge. There are two

Authors' addresses: Sam Westrick, Carnegie Mellon University, Pittsburgh, PA, USA, swestric@cs.cmu.edu; Matthew Fluet, Rochester Institute of Technology, Rochester, NY, USA, mtf@cs.rit.edu; Mike Rainey, Carnegie Mellon University, Pittsburgh, PA, USA, me@mike-rainey.site; Umut A. Acar, Carnegie Mellon University, Pittsburgh, PA, USA, umut@cs.cmu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/1-ART38
https://doi.org/10.1145/3632880

sides to the parallelism challenge. On the correctness side, the culprit is pernicious concurrency bugs, which are easy to write but hard to find and fix. On the performance side, the culprit is the (run-time) overhead of parallelism, which requires the programmer to optimize or tune their code to avoid creating too many parallel threads or tasks, lest it annul the expected benefits. Such optimization is difficult: it requires a deep understanding of the hardware, the many levels of abstractions implemented by modern programming languages, and the input data.

Much research on parallelism has focused on the correctness challenge. Researchers have considered a variety of programming languages, including both functional and procedural, and devoloped techniques to simplify the development of parallel programs. A long line of research focused on taking advantage of the absence of data races in functional programming [Arora et al. 2021; Fluet et al. 2011; Guatto et al. 2018; Li et al. 2007; Marlow and Peyton Jones 2011; Peyton Jones et al. 2008; Raghunathan et al. 2016; Sivaramakrishnan et al. 2014; Spoonhower 2009; Westrick et al. 2020]. Another long line of research focused on detecting races so as to avoid concurrency bugs [Bender et al. 2004; Cheng et al. 1998; Feng and Leiserson 1997; Fineman 2005; Flanagan and Freund 2009; Kini et al. 2017; Mellor-Crummey 1991; O'Callahan and Choi 2003; Raman et al. 2010, 2012; Savage et al. 1997; Smaragdakis et al. 2012; Utterback et al. 2016; Xu et al. 2020; Yu et al. 2005].

There has also been some research on the performance challenge. Much of this work has focused on scheduling algorithms, e.g., work stealing, that can guarantee scalability and small overheads relative to the numbers of threads. Researchers have considered scheduling from a variety of angles, including time cost [Arora et al. 2001; Eager et al. 1989], data locality [Acar et al. 2002; Blelloch et al. 2011], and and responsiveness [Muller and Acar 2016; Muller et al. 2023]. Scheduling alone, however, is not sufficient to control parallelism overhead, because it does not help with the problem of determining the "degree" of parallelism—concretely, the number of threads/tasks to create at any point in time. The work on heartbeat scheduling used an amortization technique to control the degree of parallelism [Acar et al. 2018]. Even though the approach can limit parallelism overhead in theory, it is not known if it can be realized in a high-level language with powerful features, including, for example, higher-order functions and automatic memory management.

Automatic Parallelism Management. In this paper, we propose to manage the cost and benefit of parallelism automatically by a combination of static (language based) and dynamic (run-time system) techniques. Our aim is to take parallel programs written in a high-level programming language and compile them to performant executables, without requiring the programmer to manually tune and prune the amount of parallelism. Specifically, in this paper, we consider fork-join parallel programs written in a functional programming language with effects, such as the Parallel ML language, which extends Standard ML. This language offers a primitive, **par**, which is used to express parallelism in the fork-join style.

As a simple example of the difficulty of hand-optimizing parallel programs for performance, consider a parallel, higher-order `map` function that applies a given function over a sequence. To control the cost of parallelism, the programmer may attempt to optimize `map` by rewriting it, so that "small" pieces of work do not run in parallel, but instead run sequentially. By "small", we mean computations whose parallelization costs outweigh the benefit from parallelism. The problem is, for a higher-order function such as `map`, there is no way to determine what "small" means.

To see this, consider the following three different calls: `map(fLight,s)`, `map(fHeavy,s)`, and `map(fPar,s)`. These respectively apply a light/cheap function (`fLight`), a heavy/expensive function (`fHeavy`), and a parallel function (`fPar`) to the same sequence (`s`). In the light case, "small" could be, say, ten applications of `fLight`. In the heavy case, "small" could be a just a single application of `fHeavy`; In the parallel case, it is not clear what "small" is, because it depends on precisely how `fPar` is parallelized. The `fPar` function could even contain another application of `map` itself.

Clearly, we have no uniform notion of “small”, and it would appear that no single implementation of `map` could perform well in all cases. We can complicate the problem further by throwing polymorphism into the mix, such that the sequence arguments (`s`) in the example were all of different types (e.g., `int seq`, `string seq`, and `(int seq) seq`). Now, the programmer can no longer assume a fixed type for the sequence, further complicating the reasoning.

The approach proposed in this paper leaves it to the language to manage all this complexity. We allow the programmer to write just one version of `map`, which maximally expresses all opportunities for parallelism. Then, the compiler and the run-time work together to figure out how to parallelize the computation, guaranteeing that the cost of parallelism remains low without losing its benefit.

The idea behind our approach is to compile each **par** into a ***potentially parallel*** function call—or **pcall**, for short—and represent **pcall**s explicitly as ***promotable frames*** in the call-stack. Each promotable frame can either (1) behave as a standard frame, which executes sequentially, with nearly zero additional cost, or (2) be ***promoted*** into an actual parallel task. In this way, we shift the overhead of task creation away from **par**, and instead rely on promotions (which we schedule separately) to create tasks. The programmer may therefore use **par** liberally to express all opportunities for parallelism, without harming performance.

Next, we develop a ***parallelism management algorithm*** to schedule promotions dynamically, during execution. Each promotion creates a task for an outstanding **pcall**, and thereby exposes parallelism, but incurs a cost to do so. Our algorithm uses a global amortization technique to amortize the cost of promotions against the actual work of the computation, while ensuring that a sufficient amount of parallelism is exposed for performance and scalability. Specifically, to expose sufficient parallelism, our algorithm uses an insight from heartbeat scheduling [Acar et al. 2018]: the idea is to promote the *oldest* outstanding **pcall** as soon as possible, i.e., as soon as the amortization policy allows. Each thread performs promotions independently on its own call-stack, thereby avoiding complications with concurrency and allowing all promotions to proceed in parallel.

To formalize the language-based aspect of our approach, we give a semantics for an SSA intermediate representation (IR), endowed with our **pcall** calling convention, and show how to compile source-level parallelism into this IR. Next, we present our parallelism management algorithm and prove that it asymptotically is both *work-efficient* and *span-efficient*, i.e., our algorithm bounds the overhead of parallelism without asymptotically impacting the span (length of the critical path) of the computation. Finally, we describe in detail how to implement our approach in a state-of-the-art parallel functional language implementation and evaluate its performance. We show that our implementation, called MPL^s (“Sugar MaPLe”), effectively amortizes the cost of parallelism in programs that maximally expose all parallelism at the source level, and achieves good performance and scalability in practice. Our specific contributions include the following.

- A compilation technique for *potentially parallel* function calls (**pcall**), which execute sequentially by default but can be dynamically promoted into parallel tasks (Section 3). We embed potential parallelism directly into the call-stack, guaranteeing that **pcall** has nearly zero cost.
- A semantics for an SSA intermediate language endowed with **pcall** (Section 3.3), and a technique for compiling high-level parallelism into **pcall** (Section 3.2).
- A *parallelism management* algorithm (Section 4) that dynamically performs promotions and provably guarantees both low overhead and sufficient parallelism during execution.
- A practical implementation, called MPL^s (“Sugar MaPLe”), which extends the MPL [Acar et al. 2020] compiler and run-time system with **pcall**s and automatic parallelism management (Section 5).
- An experimental evaluation, demonstrating that MPL^s successfully amortizes the cost of parallelism while retaining its benefits (Section 6).

```
fun map(f, A) = let
  val B = Array.allocate |A|
  fun go(i, j) =
    if i + 1 = j then B[i] := f(A[i])
    else
    let val m = ⌊(i+j)/2⌋
    in par(fn () ⇒ go(i, m),
           fn () ⇒ go(m, j)); () end
  in go(0, |A|); B end
```

```
// total work for par is negligible
// w.r.t. total work of many calls to expensiveFunction
fun expensiveFunction(x) = ...
val X = [1, 2, ..., 1000]
map(expensiveFunction, X)

// total work of par dominates, unless par has nearly zero cost
fun veryCheapFunction(y) = ...
val Y = [1, 2, ..., 10000000000]
map(veryCheapFunction, Y)
```

Fig. 1. The challenge of automatic parallelism management.

2 OVERVIEW AND KEY IDEAS

The source language we consider is an ML-like language with standard features (higher-order functions, parametric polymorphism, etc.), and with parallelism via the polymorphic function **par**.

$$\mathbf{par} : (\mathbf{unit} \to \alpha) \times (\mathbf{unit} \to \beta) \to \alpha \times \beta$$

Using **par**, a programmer can evaluate two functions in parallel and receive their results as a tuple. Often, **par** is used recursively to split a problem into many small pieces that can be processed in parallel. For example, a "parallel for-loop" can be implemented by splitting the index range in half and then recursively processing the two halves in parallel. This style of parallelism—known as *fork-join* parallelism—is well-suited for expressing balanced divide-and-conquer style algorithms.

In this paper, we provide an implementation of **par** that has *nearly zero cost.* Specifically, we show how to compile the expression $\mathbf{par}(f, g)$ so that, dynamically, a scheduler may choose between two behaviors for **par**: either it creates tasks to execute $f()$ and $g()$ in parallel, or it executes $f()$ and $g()$ sequentially. Our compiler and dynamic scheduler then work together to *automatically manage parallelism*, offering the following performance guarantees.

(1) **Work-efficiency**: the total overhead of all calls to **par** is nearly zero. In particular, the total cost of creating tasks should be small, e.g., 1% of total processor time. Additionally, if the scheduler chooses not to execute a call $\mathbf{par}(f, g)$ in parallel, then the cost is no more expensive than the cost of executing $f()$ and $g()$ sequentially.
(2) **Span-efficiency**: the theoretical parallelism of the execution is preserved.

Example. To illustrate the challenge of automatic parallelism management, consider the example shown in Figure 1. On the left is a divide-and-conquer parallel implementation of the function `map`. This implementation is "fully parallel" in the sense that it uses **par** to expose all theoretical parallelism, with no manual granularity control. Ideally, such an implementation would be sufficient for all uses of `map`. On the right are two example calls to `map`, with two different functions passed as argument that have different costs. When an expensive function is passed to `map`, the total cost of **par** is negligible relative to the overall computation. However, when an inexpensive function is passed to `map`, the total cost of **par** dominates, unless **par** has nearly zero cost.

A typical workaround is to use manual granularity control to stop calling **par** when the size of the range $j - i$ is small, e.g., less than 1000. Such a workaround introduces hardcoded, constant thresholds, which need to be individually tuned, often with a different threshold per callsite. For example, in Figure 1, we might choose a threshold of 1000 for `map(veryCheapFunction,`Y`)`, but the same threshold is a bad choice for `map(expensiveFunction,`X`)`, because it would eliminate all parallelism at that callsite (note that the length of X is small).

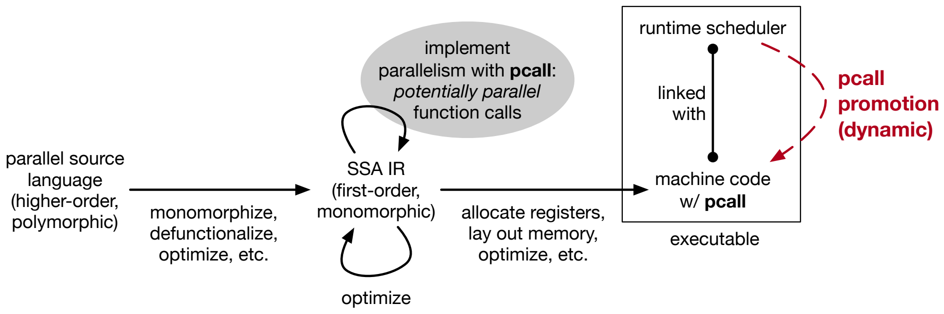
implement parallelism with **pcall**: *potentially parallel* function calls
parallel source language (higher-order, polymorphic)
monomorphize, defunctionalize, optimize, etc.
SSA IR (first-order, monomorphic)
optimize
allocate registers, lay out memory, optimize, etc.
runtime scheduler
linked with
machine code w/ **pcall**
executable
pcall promotion (dynamic)

Fig. 2. High-level summary of our approach. We leverage a standard compilation pipeline with a first-order IR. In this IR, we implement a new calling convention: a *potentially parallel* function call.

2.1 Our Approach

To meet the work- and span-efficiency guarantees described above, our approach consists of a combination of static and dynamic techniques, as summarized in Figure 2.

Statically, we use a standard compilation pipeline to lower **par** into a first-order, monomorphic, intermediate language (IR). In this IR, we make parallel control flow explicit by introducing a new calling convention: the ***potentially parallel*** call, or **pcall** for short. Each **pcall** is “potentially” parallel in the sense that it is sequential by default, but can later be dynamically *promoted* into a fully parallel call. After **pcall** is introduced in the compilation pipeline, the program can be optimized and compiled to machine code, which is linked with a runtime system and thread scheduler.

Dynamically, our scheduler judiciously selects a subset of outstanding **pcall**s to be ***promoted***. To promote a **pcall**, the scheduler (1) spawns a thread, and (2) adjusts the continuation of the **pcall** to synchronize with the spawned thread. If left unpromoted, no thread is spawned, and the synchronization code is skipped. Notably, our approach requires no dynamic check to distinguish between promoted and unpromoted **pcall**s. The only overhead of **pcall**, then, is the cost of communicating with the scheduler, to inform it of potential parallelism. We refer to this process as *marking* potential parallelism.

2.1.1 Marking Potential Parallelism. The **pcall** calling convention communicates with the scheduler using the call-stack, specifically by taking advantage of return addresses stored in call-stack frames. We use two kinds of frames: standard frames, which have a single return address, and *promotable* frames, which have three return addresses. Promotable frames have a distinguished “default” return address which aligns with the layout of standard frames, allowing promotable frames to behave as though they were standard call frames. Otherwise, the only difference between a standard and promotable frame is that a promotable frame uses two additional stack slots (to store two additional return addresses).

When a **pcall** occurs, a new promotable frame is pushed onto the callstack. The additional return addresses within this frame are used by the scheduler to dynamically adjust the behavior of the **pcall** upon return. By default, the **pcall** will return to a code path which continues execution sequentially, with no additional overhead. However, if the scheduler chooses to promote a **pcall**, it can do so by overwriting the default return address with one of the additional return addresses stored in the promotable frame. This causes the **pcall** to return to a different code path, specifically to execute code which synchronizes with another thread (spawned by the scheduler). The details of these different code paths are presented in Section 3.

Function Name	f, g		
Block Label	b		
Temporary	x, y		
Value	v	$::=$	$() \mid \textbf{true} \mid \textbf{false} \mid \ldots$
Primitive Op	o	$::=$	$\ldots$
Expression	e	$::=$	$v \mid o(\overline{x}) \mid \ldots$
Statement	S	$::=$	$x \leftarrow e$
Transfer	T	$::=$	$\textbf{goto}\ b(\overline{x}) \mid \textbf{if}\ x\ \textbf{then goto}\ b(\overline{x})\ \textbf{else goto}\ b(\overline{x}) \mid \textbf{call}\ f(\overline{x}) \rhd b \mid \textbf{return}\ (\overline{x})$
			$\mid\ \textbf{pcall}\ f(\overline{x}) \rhd (b, b, b) \ \mid\ \textbf{getjoin} \rhd b \ \mid\ \textbf{setjoin}\ (x)$
Basic Block	B	$::=$	$\textbf{block}\ b(\overline{x}) = \{\overline{S}; T\}$
Function	F	$::=$	$\textbf{fun}\ f(\overline{x}) = \textbf{let}\ \overline{B}\ \textbf{in}\ \{\overline{S}; T\}$
Program	P	$::=$	$\textbf{let}\ \overline{F}\ \textbf{in}\ f()$

Fig. 3. SSA IR syntax. The language is standard except for the highlighted constructs.

2.1.2 Dynamic Promotions. With all potential parallelism marked in call-stacks, the scheduler is free to choose to promote an outstanding **pcall** at any moment. Selecting **pcall**s to promote requires care, as there is a delicate balance between work-efficiency and span-efficiency: each promotion exposes more parallelism, but requires more work.

In Section 4, we present a dynamic algorithm that is provably work- and span-efficient. At a high level, our algorithm tracks the amount of "true" work performed by each thread (which excludes the cost of promotions) by accumulating tokens periodically during execution. These tokens may then be spent to perform promotions, guaranteeing that the number of promotions is small relative to the work of the computation.

To guarantee that sufficient parallelism is exposed, it is important to carefully select which **pcall**s are promoted. Here, we use an insight from prior work. In particular, in their work on heartbeat scheduling, Acar et al. [2018] proved formally that prioritizing the *oldest* tasks for promotions guarantees at most a constant factor inflation of the *span* (i.e., the length of the critical path). Intuitively, the oldest tasks need to be promoted as soon as possible because these tasks have already suffered the longest delay (and therefore are most likely to be on the critical path). Our promotion algorithm is inspired by this prior work, and in particular our proof of span-efficiency (Section 4, Theorem 4.2) similarly requires that promotions prioritize the oldest tasks. Therefore, in Section 3 (and especially in the operational semantics of Figure 8), we specialize for this behavior. This is important for performance, but otherwise not essential to the semantics of **pcall**.

3 PCALL: POTENTIALLY PARALLEL CALLS

The primary workhorse IR we consider is a first-order SSA (static single assignment) language. Our **pcall** primitive is not directly exposed to the user; rather, we show how to compile the source level **par** primitive into **pcall**, leveraging a standard compilation pipeline for functional languages.

In this section, we define the **pcall** primitive and give it a semantics. At a high level, **pcall** is similar to a standard function call, except that it has multiple possible continuations instead of just one. These multiple continuations are used to endow **pcall** with two behaviors, one of which is sequential and another which is parallel. In this way, **pcall** is *potentially parallel.*

3.1 Syntax

Figure 3 defines the syntax of the SSA IR we consider. A program in this IR is a collection of top-level first-order functions where each function is a collection of basic blocks.

A basic block consists of a *label*, a (potentially empty) list of *parameters*, a list of statements, and a *transfer* which specifies the continuation of the block. Note that we use basic blocks augmented with parameters, rather than phi nodes.

In keeping with the SSA tradition, we refer to the variables of the language as *temporaries*, denoted x, y, etc. Each temporary is assigned exactly once, and may be used in any block dominated by the block in which the temporary was assigned. For simplicity, we assume temporaries are uniquely named (i.e., no shadowing). Parameters (of basic blocks and functions) are also temporaries, and are considered assigned on entry (to the block or function).

Statements perform assignments to temporaries by evaluating simple expressions. These expressions could be operations such as arithmetic, memory allocation, reads and writes, etc. We omit the details of expressions, as these details are not pertinent to the control flow of the language. We instead focus on control flow, defined by the transfers of the language.

Standard transfers include unconditional jumps, conditional jumps, function calls, and function returns. Our extensions to the language are three additional transfers: **pcall**, **getjoin**, and **setjoin**. The **pcall** primitive is our main extension, discussed below.

The **getjoin** and **setjoin** primitives are used synchronize spawned threads, to implement the parallel behavior of **pcall**. Each **getjoin** statically has a **setjoin** that it is paired with; the semantics of the **getjoin** are to block until the corresponding **setjoin** occurs, which passes a value to the **getjoin**. In other words, these primitives encode a synchronization variable. The syntax of **getjoin** has a continuation b, which has arity 1, and receives the value of the corresponding **setjoin** as argument. In contrast, **setjoin** has no continuation, and instead it terminates the current thread.

We include **getjoin** and **setjoin** in the language to make it possible to give a complete operational semantics (Section 3.3) in terms of threads and thread synchronizations. Note that, in practice, these transfers are defined by a scheduler which is not "baked into" the compiler proper. This nuance is discussed in Section 5.1.2.

Call and PCall. The defining feature of our language is that it has two types of calls: standard function calls, and *potentially parallel* calls. Standard calls, written **call** $f(x_1, \ldots, x_n) \rhd b$, consist of: a function name f, a (possibly empty) list of arguments $x_1, \ldots, x_n$, and a return label b, indicating where control should return to when the function call returns.

The **pcall** transfer is similar, except with three return labels instead of one. It is written **pcall** $f(x_1, \ldots, x_n) \rhd (b_{\text{seq}}, b_{\text{sync}}, b_{\text{spwn}})$. The first return continuation, b_{seq}, is used for the sequential behavior. By default, **pcall** will use this continuation, causing it behave identically to **call**, i.e., it behaves as a standard function call. The latter two continuations, b_{sync} and b_{spwn}, are used for the parallel behavior: b_{spwn} points to the code that will be executed in parallel, by a spawned thread, and b_{sync} points to the code that will synchronize with the spawned thread.

The intent is that the code path at b_{sync} will eventually call **getjoin** to wait for the spawned thread to complete; similarly, the code path at b_{spwn} will eventually call **setjoin** to pass a value back to the original thread. Importantly, on the sequential code path b_{seq}, no thread is spawned, and no synchronization code is necessary.

3.2 Lowering High-Level Parallelism into SSA with PCall

The **par** function from Section 2 is lowered into SSA in two steps, illustrated in Figures 4 and 5.

First, using standard compilation techniques (in particular, defunctionalization), a source-level **par**(f, g) expression is compiled into SSA by encoding it as a transfer of the form **par** $f_{\text{top}}(\ldots), g_{\text{top}}(\ldots) \rhd b$. This transfer resembles a normal function call, but it calls two functions instead of one. (Note the similarity with **call** transfer of Figure 3.) Figures 4a and 4b respectively show (a) an example expression **par**(f, g), and (b) the resulting code after compiling into SSA. The

```
let $x$ = ... in let $y$ = ... in let $z$ = ... in
let $f$ = fn () ⇒ ... $x$ ... in
let $g$ = fn () ⇒ ... $y$ ... $z$ ... in
let $r$ = par($f, g$) in
... $r$ ...
```

(a) Example source

```
fun $f_{top}(x)$ = ... $x$ ...
fun $g_{top}(y, z)$ = ... $y$ ... $z$ ...
fun $main()$ =
  let
    ...
    block $finish\_par(r_1, r_2)$ =
      {$r \leftarrow (r_1, r_2)$; goto $after\_par()$}
    block $after\_par()$ = ... $r$ ...
  in { $x \leftarrow$ ...; $y \leftarrow$ ...; $z \leftarrow$ ...;
       par $f_{top}(x), g_{top}(y, z) \triangleright finish\_par$ }
```

(b) Translation into SSA with **par** transfer, to be eliminated

```
fun $f_{top}(x)$ = ... $x$ ...
fun $g_{top}(y, z)$ = ... $y$ ... $z$ ...
fun $main()$ =
  let
    ...
    block $seq\_enter(r_1')$ = {call $g_{top}(y, z) \triangleright seq\_exit$}
    block $seq\_exit(r_2')$ = {goto $finish\_par(r_1', r_2')$}

    block $sync\_enter(r_1'')$ = {getjoin $\triangleright sync\_exit$}
    block $sync\_exit(r_2'')$ = {goto $finish\_par(r_1'', r_2'')$}

    block $spwn\_enter()$ = {call $g_{top}(y, z) \triangleright spwn\_exit$}
    block $spwn\_exit(r_2''')$ = {setjoin($r_2'''$)}

    block $finish\_par(r_1, r_2)$ = {$r \leftarrow (r_1, r_2)$; goto $after\_par()$}
    block $after\_par()$ = ... $r$ ...
  in { $x \leftarrow$ ...; $y \leftarrow$ ...; $z \leftarrow$ ...;
       pcall $f_{top}(x) \triangleright (seq\_enter, sync\_enter, spwn\_enter)$ }
```

(c) After eliminating **par**, replacing with **pcall**, **getjoin**, and **setjoin**.

Fig. 4. Implementing parallelism using **pcall** in SSA.

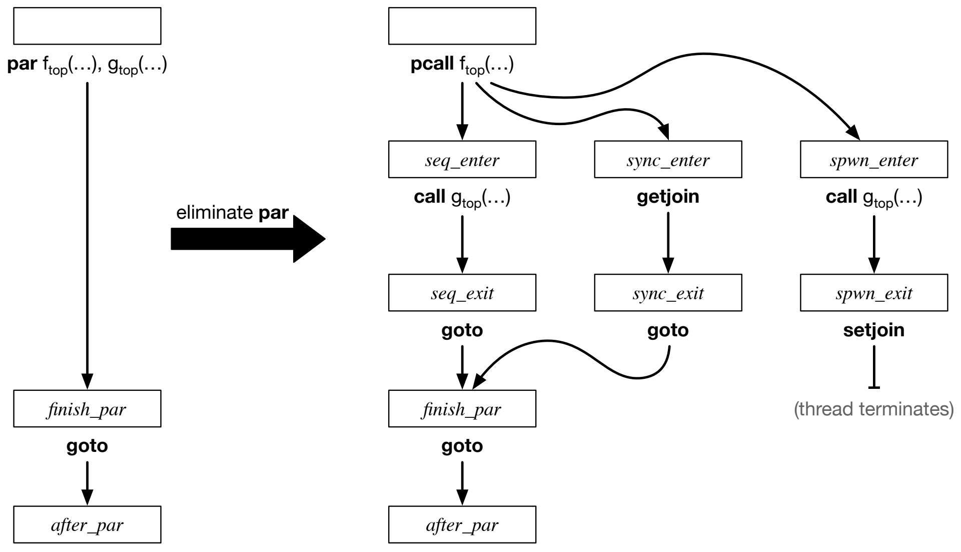

Fig. 5. Control flow graph for our approach, illustrating Figures 4b and 4c. The left (corresponding to Figure 4b) shows the control flow before eliminating **par**. The right (corresponding to Figure 4c) shows the control flow with **pcall**. Rectangles are basic blocks. Each block ends with a transfer, which has one or more successor blocks.

Thread Identifier	t		
Value Mapping	$\mathcal{X}, \mathcal{Y}$	$\in$	*(Temporary)* $\rightharpoonup$ *(Value)*
Promotion Mark	ρ	::=	**none** | **promotable**(b, b)
Join Obligation	σ	::=	**none** | **child**(t)
Stack Frame	k	::=	**frame**$(\rho, \sigma, \mathcal{X}, f, b)$
Stack	$\mathcal{K}$	::=	$\bullet$ | $\mathcal{K}; k$
Thread State	$\mathcal{T}$	::=	**thread**$(\mathcal{K}, \sigma, \mathcal{X}, f, \{\overline{S}; T\})$
Thread Pool	$\mathcal{P}$	$\in$	*(Thread Identifier)* $\rightharpoonup$ *(Thread State)*

Fig. 6. SSA IR: stacks and threads for operational semantics.

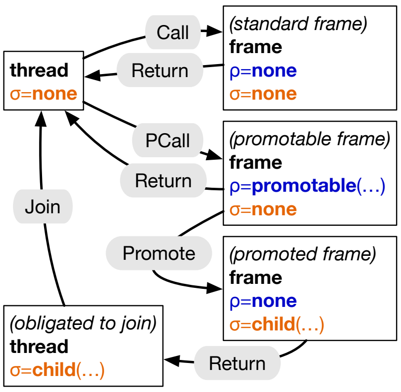

Fig. 7. Overview of calling convention, from caller's perspective. Arrows correspond to rules of Figure 8.

SSA-level functions f_{top} and g_{top} are the result of defunctionalizing the first-class functions f and g. The continuation block b has arity 2, expecting the results of f_{top} and g_{top} to be passed as argument.

Second, shown in Figures 4b, 4c, and 5, we eliminate the **par** transfer via an SSA-to-SSA translation. This translation introduces **pcall**, as well as **getjoin** and **setjoin**. In the resulting code, we begin by calling $f_{\text{top}}(\ldots)$ using a **pcall** instead of a standard function call. There are then three possible continuations of **pcall**: *seq_enter*, *sync_enter*, and *spwn_enter*, which are used to implement two different behaviors.

- **If the pcall is never promoted**, only the *seq_enter* path is used. This path simply calls $g_{\text{top}}(\ldots)$ sequentially after completing $f_{\text{top}}(\ldots)$.
- **If the pcall is promoted**, only the *sync_enter* and *spwn_enter* paths are used. In this case, the runtime system creates a new thread. The new thread executes the path starting at *spwn_enter*. From the perspective of the new thread, it is as though the function call $f_{\text{top}}(\ldots)$ completed but returned no result. In the meantime, the original thread is still working on the call $f_{\text{top}}(\ldots)$. When $f_{\text{top}}(\ldots)$ returns, the original thread will execute the path starting at *sync_enter*. On this path, the original thread simply performs a **getjoin**, which blocks until the corresponding **setjoin** occurs. Finally, the original thread can continue with *finish_par*.

3.3 Operational Semantics

We specify the meaning of **pcall** by presenting an operational semantics for the language. The semantics is a small-step semantics of the form $\overline{F} \vdash \mathcal{P} \longmapsto \mathcal{P}$, where $\overline{F}$ is a collection of first-order function definitions, and $\mathcal{P}$ is a pool of active threads. The syntax for threads and thread pools is given in Figure 6, and an overview of the calling convention is shown in Figure 7. The rules of the operational semantics are shown in Figure 8.

3.3.1 *Overview of Operational Semantics.*

The high-level idea of the operational semantics is to augment the frames of a standard call-stack with two additional components: a promotion mark ρ, and a join obligation σ. These components are managed by the calling convention of the language, and are used to keep track of both promoted and unpromoted **pcall**s, as well as to allow concurrent threads to synchronize with each other.

This aspect of the design is intricate, so we give a diagram in Figure 7. In the diagram, the arrows correspond to rules in the operational semantics, and the white boxes represent different states

Auxiliary Definitions: Function and Block Lookup

$$\frac{\overline{F} \ni \textbf{fun}\ f(\overline{x}) = \textbf{let}\ \_\ \textbf{in}\ \{\overline{S}; T\}}{\mathsf{FuncLookup}(\overline{F}, f) = ((\overline{x}), \{\overline{S}; T\})} \qquad \frac{\overline{F} \ni \textbf{fun}\ f(\_) = \textbf{let}\ \overline{B}\ \textbf{in}\ \_ \qquad \overline{B} \ni \textbf{block}\ b(\overline{x}) = \{\overline{S}; T\}}{\mathsf{BlockLookup}(\overline{F}, f, b) = ((\overline{x}), \{\overline{S}; T\})}$$

Auxiliary Definitions: Oldest Promotable Frame

$$\frac{\mathsf{NoPromotableFrames}(\mathcal{K}) \qquad k_{\mathrm{p}} = \textbf{frame}(\textbf{promotable}(\_), \_, \_, \_, \_)}{\mathsf{SplitOldestPromotable}(\mathcal{K}; k_{\mathrm{p}}) = \mathcal{K}; k_{\mathrm{p}}; \bullet}$$

$$\frac{\mathsf{SplitOldestPromotable}(\mathcal{K}) = \mathcal{K}_1; k_{\mathrm{p}}; \mathcal{K}_2 \qquad \mathcal{K}_2' = \mathcal{K}_2; k}{\mathsf{SplitOldestPromotable}(\mathcal{K}; k) = \mathcal{K}_1; k_{\mathrm{p}}; \mathcal{K}_2'}$$

$$\frac{}{\mathsf{NoPromotableFrames}(\bullet)} \qquad \frac{\mathsf{NoPromotableFrames}(\mathcal{K})}{\mathsf{NoPromotableFrames}(\mathcal{K}; \textbf{frame}(\textbf{none}, \_, \_, \_, \_))}$$

Execution $\boxed{\overline{F} \vdash \mathcal{P} \longmapsto \mathcal{P}}$

$$\frac{\begin{array}{c}\mathcal{T} = \textbf{thread}(\mathcal{K}, \sigma, \mathcal{X}, f, \{S_1; S_2; \ldots; T\}) \\ \mathcal{X}' = \ldots \mathit{apply}\ S_1\ \mathit{to}\ \mathcal{X} \ldots \qquad \mathcal{T}' = \textbf{thread}(\mathcal{K}, \sigma, \mathcal{X}', f, \{S_2; \ldots; T\})\end{array}}{\overline{F} \vdash \mathcal{P}[t \hookrightarrow \mathcal{T}] \longmapsto \mathcal{P}[t \hookrightarrow \mathcal{T}']}\ \textsc{Stmt}$$

$$\frac{\begin{array}{c}\mathcal{T} = \textbf{thread}(\mathcal{K}, \sigma, \mathcal{X}, f, \{\textbf{call}\ g(x_1, \ldots, x_n) \rhd b\}) \\ \mathsf{FuncLookup}(\overline{F}, g) = ((y_1, \ldots, y_n), \{\overline{S}; T\}) \qquad \mathcal{Y} = [y_1 \hookrightarrow \mathcal{X}(x_1), \ldots, y_n \hookrightarrow \mathcal{X}(x_n)] \\ \mathcal{K}' = \mathcal{K}; \textbf{frame}(\textbf{none}, \sigma, \mathcal{X}, f, b) \qquad \mathcal{T}' = \textbf{thread}(\mathcal{K}', \textbf{none}, \mathcal{Y}, g, \{\overline{S}; T\})\end{array}}{\overline{F} \vdash \mathcal{P}[t \hookrightarrow \mathcal{T}] \longmapsto \mathcal{P}[t \hookrightarrow \mathcal{T}']}\ \textsc{Call}$$

$$\frac{\begin{array}{c}\mathcal{T} = \textbf{thread}(\mathcal{K}, \textbf{none}, \mathcal{Y}, \_, \{\textbf{return}\ (y_1, \ldots, y_n)\}) \\ \mathcal{K} = \mathcal{K}'; \textbf{frame}(\_, \sigma, \mathcal{X}, f, b) \qquad \mathsf{BlockLookup}(\overline{F}, f, b) = ((x_1, \ldots, x_n), \{\overline{S}; T\}) \\ \mathcal{X}' = \mathcal{X}[x_1 \hookrightarrow \mathcal{Y}(y_1), \ldots, x_n \hookrightarrow \mathcal{Y}(y_n)] \qquad \mathcal{T}' = \textbf{thread}(\mathcal{K}', \sigma, \mathcal{X}', f, \{\overline{S}; T\})\end{array}}{\overline{F} \vdash \mathcal{P}[t \hookrightarrow \mathcal{T}] \longmapsto \mathcal{P}[t \hookrightarrow \mathcal{T}']}\ \textsc{Return}$$

$$\frac{\begin{array}{c}\mathcal{T} = \textbf{thread}(\mathcal{K}, \textbf{none}, \mathcal{X}, f, \{\textbf{pcall}\ g(x_1, \ldots, x_n) \rhd (b_{\mathrm{seq}}, b_{\mathrm{sync}}, b_{\mathrm{spwn}})\}) \\ \mathsf{FuncLookup}(\overline{F}, g) = ((y_1, \ldots, y_n), \{\overline{S}; T\}) \qquad \mathcal{Y} = [y_1 \hookrightarrow \mathcal{X}(x_1), \ldots, y_n \hookrightarrow \mathcal{X}(x_n)] \\ \mathcal{K}' = \mathcal{K}; \textbf{frame}(\textbf{promotable}(b_{\mathrm{sync}}, b_{\mathrm{spwn}}), \textbf{none}, \mathcal{X}, f, b_{\mathrm{seq}}) \\ \mathcal{T}' = \textbf{thread}(\mathcal{K}', \textbf{none}, \mathcal{Y}, g, \{\overline{S}; T\})\end{array}}{\overline{F} \vdash \mathcal{P}[t \hookrightarrow \mathcal{T}] \longmapsto \mathcal{P}[t \hookrightarrow \mathcal{T}']}\ \textsc{PCall}$$

$$\frac{\begin{array}{c}\mathcal{T}_1 = \textbf{thread}(\mathcal{K}, \sigma, \mathcal{X}, f, \{\overline{S_1}; T_1\}) \\ \mathsf{SplitOldestPromotable}(\mathcal{K}) = \mathcal{K}_1; \textbf{frame}(\textbf{promotable}(b_{\mathrm{sync}}, b_{\mathrm{spwn}}), \textbf{none}, \mathcal{Y}, g, \_); \mathcal{K}_2 \\ \mathcal{K}' = \mathcal{K}_1; \textbf{frame}(\textbf{none}, \textbf{child}(t_2), \mathcal{Y}, g, b_{\mathrm{sync}}); \mathcal{K}_2 \qquad \mathcal{T}_1' = \textbf{thread}(\mathcal{K}', \sigma, \mathcal{X}, f, \{\overline{S_1}; T_1\}) \\ \mathsf{BlockLookup}(\overline{F}, g, b_{\mathrm{spwn}}) = ((), \{\overline{S_2}; T_2\}) \qquad \mathcal{T}_2 = \textbf{thread}(\bullet, \textbf{none}, \mathcal{Y}, g, \{\overline{S_2}; T_2\}) \qquad t_2\ \text{fresh}\end{array}}{\overline{F} \vdash \mathcal{P}[t_1 \hookrightarrow \mathcal{T}_1] \longmapsto \mathcal{P}[t_1 \hookrightarrow \mathcal{T}_1'][t_2 \hookrightarrow \mathcal{T}_2]}\ \textsc{Promote}$$

$$\frac{\begin{array}{c}\mathcal{T}_1 = \textbf{thread}(\mathcal{K}, \textbf{child}(t2), \mathcal{X}, f, \{\textbf{getjoin} \rhd b\}) \qquad \mathcal{T}_2 = \textbf{thread}(\_, \_, \mathcal{Y}, \_, \{\textbf{setjoin}(y)\}) \\ \mathsf{BlockLookup}(\overline{F}, f, b) = ((x), \{\overline{S}; T\}) \qquad \mathcal{X}' = \mathcal{X}[x \hookrightarrow \mathcal{Y}(y)] \qquad \mathcal{T}_1' = \textbf{thread}(\mathcal{K}, \textbf{none}, \mathcal{X}', f, \{\overline{S}; T\})\end{array}}{\overline{F} \vdash \mathcal{P}[t_1 \hookrightarrow \mathcal{T}_1][t_2 \hookrightarrow \mathcal{T}_2] \longmapsto \mathcal{P}[t_1 \hookrightarrow \mathcal{T}_1']}\ \textsc{Join}$$

Fig. 8. SSA IR: operational semantics (selected rules only). Rules Promote and Join respectively spawn and synchronize threads. It is correct for Promote to happen any time. For performance, Promote must happen *enough* (for parallelism) and also *not too much* (for work-efficiency). In Section 4, we describe an algorithm for controlling when Promote occurs.

that an executing function can be in, forming a small state machine. When a function is actively being executed by a thread (i.e., when the function currently has no outstanding callees), we label it simply as a **thread**. Otherwise, when a function is suspended waiting for an outstanding callee, it is pushed onto the call-stack as a **frame**.

The following three types of frames are used within the call-stack.

(1) A ***standard frame*** corresponds to a standard function call, and is identified by the condition $\rho = \textbf{none} \land \sigma = \textbf{none}$.

(2) A ***promotable frame*** corresponds to a **pcall**ed function which has not been promoted, and is identified by the condition $\rho = \textbf{promotable}(\ldots) \land \sigma = \textbf{none}$.

(3) A ***promoted frame*** corresponds to a **pcall**ed function which has already been promoted, and is identified by the condition $\rho = \textbf{none} \land \sigma = \textbf{child}(\ldots)$.

When a function frame is promoted, the semantics sets $\sigma = \textbf{child}(t)$; at this point, the function is then obligated to join with the spawned child thread, t, before the function itself performs a **pcall** or returns. Note that this obligation is met by the transformation that introduces **pcall**, as shown in Figure 5. By requiring a join with the spawned child, the semantics ensures that the lifetime of the child thread aligns with the lifetime of the **pcall**ed function, which is necessary to match the intended semantics of the source-level **par**. In our operational semantics, the obligation to join is encoded in rules PCall and Return, both of which get stuck if the current thread has $\sigma \neq \textbf{none}$.

It is important to emphasize that these nuances in the semantics are intentional: we designed the semantics with the goal of having a single, uniform mechanism for function returns, regardless of whether the callee was **call**ed or **pcall**ed. In particular, on return to a caller, the promotion mark of the caller is simply thrown away. The performance implications of this design are significant: if not promoted, a **pcall** is nearly identical to a standard **call**; the only difference is a small amount of additional data (the promotion mark, ρ) stored in the frame.

3.3.2 Detailed Description of Operational Semantics.

Threads, frames, and call-stacks. We denote call-stacks by $\mathcal{K}$, which are stacks of frames. When a thread performs a functional call (Figure 8, either rule Call or rule PCall), a new frame is pushed onto the stack. Similarly, when a function call returns (rule Return), the most recent frame is popped. Each frame k stores the promotion mark ρ and join obligation σ as described above, as well as a collection of stack slots $\mathcal{X}$ (mapping temporaries to values), a function name f, and a return block label, b.

Threads, denoted $\mathcal{T}$, consist of a call-stack $\mathcal{K}$ together with (what essentially amounts to) the data of a "current" frame. Specifically, a $\textbf{thread}(\mathcal{K}, \sigma, \mathcal{X}, f, \{S_1; \ldots S_n; T\})$ is currently executing the statements $\{S_1; \ldots S_n; T\}$ inside function f, with current temporaries $\mathcal{X}$, in the call stack $\mathcal{K}$.

Calls and returns. Standard function calls are defined by rule Call. In this rule, when a thread of the form $\textbf{thread}(\mathcal{K}, \sigma, \mathcal{X}, f, \ldots)$ wishes to call a function, it does so by pushing $\textbf{frame}(\textbf{none}, \sigma, \mathcal{X}, f, b)$ onto its call-stack, where b is the return block of the call.

Similarly, potentially parallel calls (**pcall**s) are defined by rule PCall. In this rule, when a thread of the form $\textbf{thread}(\mathcal{K}, \textbf{none}, \mathcal{X}, f, \ldots)$ wishes to **pcall** a function, it does so by pushing $\textbf{frame}(\textbf{promotable}(b_{\text{sync}}, b_{\text{spwn}}), \textbf{none}, \mathcal{X}, f, b_{\text{seq}})$ onto its call-stack, where b_{seq}, b_{sync}, and b_{spwn} are the three continuations of the **pcall**.

Note that the semantics has only one Return rule. That is, to return from a function call, the language blindly returns to the label in the most recent frame, and ignores the promotion mark ρ.

Promotion. In the semantics, the rule Promote causes a thread to promote one promotable frame in the thread's call-stack. In principle, any frame could be promoted, but we specialize the semantics to select the oldest promotable frame, as this is what our parallelism management algorithm requires (Section 4). We use an auxiliary definition, SplitOldestPromotable, to find an appropriate promotable frame within the call-stack.

The frame selected for promotion has the form $\mathbf{frame}(\mathbf{promotable}(b_{\mathsf{sync}}, b_{\mathsf{spwn}}), \mathbf{none}, \mathcal{Y}, g, \_)$. The rule Promote spawns a new thread with identifier t_2, and then updates the selected frame to $\mathbf{frame}(\mathbf{none}, \mathbf{child}(t_2), \mathcal{Y}, g, b_{\mathsf{sync}})$. Three components of the frame are updated: (1) the promotion mark is set to **none**, indicating that this frame is no longer promotable; (2) the join obligation is set to $\mathbf{child}(t_2)$, where t_2 is the identifier of the newly spawned thread of this promotion; (3) the return label is set to b_{sync}, which will cause the thread to execute the appropriate synchronization code when it returns to this frame. Finally, the newly spawned thread is initialized with a copy of the value mapping $\mathcal{Y}$ (copied from the promoted frame), and immediately begins execution at the b_{spwn} codepath. In the rule, an auxiliary function BlockLookup is used to look up the body of this block, which has arity 0.

Synchronization. By storing the join obligation σ explicitly, we allow for the **getjoin** and **setjoin** primitives to synchronize with each other. In particular, in rule Join, we see that a thread labeled t_1 is waiting on a **getjoin**, and the join obligation of that thread is set to $\mathbf{child}(t_2)$. Meanwhile, thread t_2 has reached a **setjoin**. The two threads then synchronize by passing the argument of the **setjoin** to the continuation of the **getjoin**, i.e., thread t_2 passes a value to thread t_1. Finally, the thread t_2 terminates and is removed from the thread pool, and thread t_1 continues with its join obligation updated to **none**.

4 PARALLELISM MANAGEMENT ALGORITHM

The semantics of Section 3 allows for promotions (via rule Promote of Figure 8) at any time. Although correct for execution, this is inefficient. Here, we present a parallelism management algorithm that restricts promotions to guarantee low overhead (work efficiency) and while preserving parallelism (span efficiency). The algorithm tracks the amount of work performed during execution and uses this information to promote a subset of outstanding potentially parallel calls.

4.1 Token Accounting Algorithm

Our parallelism management algorithm uses an amortization technique. During execution, threads explicitly track the amount of work that they perform, excluding promotions. Promotions are then performed only when their cost can be charged against non-promotion work.

Ideally, to track work, each thread would count exactly the number of steps it performs. However, this approach is too fine-grained to be implemented efficiently. We therefore coarsen the approach: rather than count every step, threads instead periodically collect some number of ***work tokens***, and then spend tokens to perform promotions.

Algorithm description. After every N steps of work, each thread receives C work tokens. (The variables C and N are tunable parameters; see Section 4.3 for a discussion, and Section 5.2 for implementation details.) These tokens are then spent to perform promotions:

- Every time a thread receives tokens, it checks if there is a promotable frame in its own call-stack; if so, it spends a token and performs one promotion. This is repeated until the thread either runs out of tokens, or no longer has promotable frames in its call-stack. Any unspent tokens are saved.
- Every time a thread performs a **pcall**, it checks how many saved tokens it has. If the thread has at least one saved token, it immediately performs one promotion and spends a token.

Accumulating and distributing tokens. Note that a thread may accumulate many tokens, for example, by performing a long sequential task without any **pcall**s. Our technique allows accumulated tokens to be distributed among threads without violating work-efficiency in essentially any manner. For example, one strategy would be to have a global pool of work tokens where excess tokens are accumulated, allowing threads to share tokens freely with other threads as needed. Another strategy would be to give each processor a pool of tokens, and to integrate with scheduling to distribute tokens (e.g., by stealing half of another processor's tokens at each successful steal in a work-stealing scheduler).

In our implementation, we use a simple heuristic which distributes tokens locally, at spawns and joins. When a thread spawns a child, the parent passes half of its (unused) tokens to the child. When a child joins with its parent, it passes its (unused) tokens to the parent. This heuristic has the tendency to distribute tokens evenly across divide-and-conquer computations.

4.2 Work- and Span-efficiency

By explicitly accounting for work tokens, our algorithm enforces both work-efficiency and span-efficiency, as stated in the bounds shown below. In particular, the additional cost of promotions increases the work and span of a computation each by at most a constant multiplicative factor. Note that these bound are precise, not asymptotic.

THEOREM 4.1 (WORK-EFFICIENCY). *Consider a program with work W, excluding the work of promotions. The number of promotions performed will be at most $\frac{C}{N}W$. Therefore, if promotion costs τ work, the total amount of work (including the cost of promotions) is $\left(1 + \frac{C \cdot \tau}{N}\right) W$.*

THEOREM 4.2 (SPAN-EFFICIENCY). *Consider a program with span S, excluding the cost of promotions. The overall span, including the cost of promotions, is at most $(\tau + N)S$, where τ is the cost of promotion.*

4.3 Tunable Parameters: Set and Forget

Our algorithm has two tunable parameters: C and N, where C is the number of tokens generated by each thread after every N steps of work. These parameters only need to be set once, to control the cost of promotions. In particular, our work-efficiency bound (Theorem 4.1) clarifies that the ratio $\frac{C \cdot \tau}{N}$ should be set close to 0, where τ is the cost of a promotion. For example, setting this ratio to .01 would guarantee that the cost of promotions consumes at most 1% of total processor time. This bound holds for all programs.

Our span-efficiency bound (Theorem 4.2) additionally suggests that N should be set as small as possible, as increasing N decreases parallelism. In practice, N cannot be made arbitrarily small. As we discuss further in Section 5, the parameter N is determined by a signal handling mechanism which polls to check if a periodic signal has arrived. To limit the cost of signal handling, N must be set to a reasonably large interval. Therefore, rather than tuning N, we can instead control the cost of promotions by selecting an appropriate value for C.

5 IMPLEMENTATION

We implemented our PCall SSA IR (Section 3) and our parallelism management algorithm (Section 4) in the context of a compiler and runtime system dubbed MPL^s ("Sugar MaPLe"). MPL^s is a modification of MPL ("MaPLe") [Acar et al. 2020], which has been exploring efficient and scalable parallel functional programming by coupling thread scheduling and memory management for nested fork-join parallelism [Acar et al. 2015] through disentanglement [Arora et al. 2021; Westrick et al. 2022, 2020] and hierarchical heaps [Guatto et al. 2018; Raghunathan et al. 2016]. MPL is itself a modification of MLton [MLton nd; Weeks 2006], a whole-program optimizing compiler for Standard ML. MPL and MPL^s inherit many features from MLton, especially in terms of the compiler proper; the most substantial changes are localized to the runtime system to support thread scheduling and memory management and to the implementation of the (extended) standard library, where a significant portion of thread scheduling and memory management is implemented in source SML code with calls to MPL/MPL^s runtime-system functions as necessary.

In this section, we present an overview of several important aspects of the MPL^s implementation, including compiler changes to support the PCall SSA IR and the integration of parallelism management with the thread-scheduling and memory-management components of MPL. We note that, although MPL^s and MLton are whole-program optimizing compilers, whole-program compilation is not required to implement the special behavior of **pcall**.

5.1 Implementing the PCall SSA IR in MLton/MPL^s

Implementing a novel control-flow construct such as **pcall** in an industrial-strength, optimizing compiler would appear to be a significant and invasive undertaking. At a very-high level, extending a compiler with support for **pcall** requires three major categories of changes:

(1) Front-end changes, which introduce a mechanism (e.g., special syntax or primitive function) by which source code can indicate function calls that should be executed as **pcall**s along with the three return continuations.
(2) Middle-end changes, which explicitly represent **pcall**s in IRs and update analysis and optimization passes to handle **pcall**s.
(3) Back-end changes, which implement the **pcall** calling convention and emit **pcall** frames.

Our claim is that the implementation effort required to support **pcall** in a production compiler can be surprisingly low and our evidence is the implementation of MPL^s as a set of changes to MPL. Although *these particular changes* to *this particular compiler* may not be directly transferable to other systems, we give a moderate level of detail in order to support our claim with an accounting of the effort required to implement **pcall** in MPL^s.

Through a fortunate combination of compiler-construction and functional-programming techniques and building on our expertise with the compiler infrastructure, we were able to implement **pcall** in MLton/MPL^s by making mostly local changes to the codebase. In total, changes were limited to less than 2000 lines (1.2%) of the 162K LOC that comprise the compiler proper. This does not include the runtime system or standard library, which include MPL^s's thread scheduling and parallelism management algorithm, as these are implemented outside of the compiler proper.

To implement the PCall SSA IR, we added just three new compiler primitives, modified the middle-end IRs to support the **pcall** transfer, and introduced new forms of call-stack frames.

5.1.1 MLton Compilation Preliminaries. MLton's approach to compilation can be summarized as whole-program optimization using a simply-typed first-order intermediate language. MLton is a batch compiler and does not support separate compilation; rather, all SML source code contributing to the program is compiled as a single compilation unit. Complete knowledge of the program is necessary for the critical steps used to convert SML's modules, parametric polymorphism, and higher-order functions into a simply-typed first-order intermediate representation; it also allows for aggressive control-flow and data-representation optimizations. Briefly, SML's modules are eliminated by *defunctorization* [Elsman 1999]; parametric polymorphism is eliminated by *monomorphisation* [Tolmach and Oliva 1998], yielding a simply-typed program; and higher-order functions are eliminated by *defunctionalization* [Reynolds 1972] using a monovariant whole-program control-flow analysis [Cejtin et al. 2000], yielding a first-order program.

Most of MLton's optimizations are performed on simply-typed, first-order SSA-based intermediate representations. MLton's SSA IRs are very similar to the one presented in Section 3 (minus the

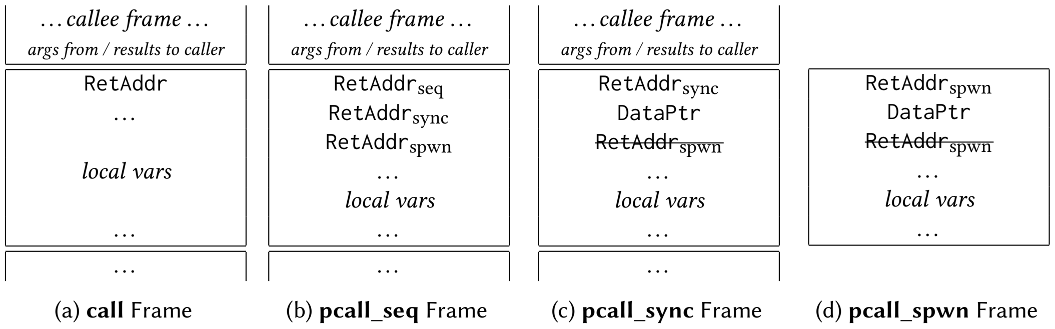

Fig. 9. Frames (stack grows upward)

constructs expressing parallelism). To efficiently compile SML, MLton's SSA IRs also include transfers for tail calls, non-tail calls that install exception handlers, raising exceptions, and case dispatch on algebraic datatypes. MLton's three SSA IRs use successively lower-level data representations, but are otherwise nearly identical in terms of control-flow transfers.

After the SSA IR optimizations, MLton gathers garbage-collection information and translates to a low-level untyped IR that makes the call stack explicit. Finally, MLton generates either C, LLVM, native x86, or native amd64 code that is compiled and linked with the runtime-system library; MPL and MPL^s are restricted to using the C code generator.

Stacks, Frames, and Calling Conventions. To support precise garbage collection, the runtime system must walk call stacks in order to identify roots. A call stack is a single (contiguous) heap-allocated object. The lower-address portion is filled with a linear sequence of frames and the remaining higher-address portion is reserved to accommodate pushing additional frames at function calls. A stack-top pointer points to the top of the frame of the active function and determines the boundary between the used and reserved portion of the call stack. A call stack can be relocated and resized (increasing or decreasing its reserved space) during garbage collection.

A (normal) **call** frame collects function-local variables that are live at the call and stores a return address at the top (highest-address) of the frame (see Figure 9a); it implements a **frame**(**none**, **none**, X, f, b) from the operational semantics of Section 3. The calling convention is simple: the caller writes arguments just above its frame (in what will become the bottom of the frame of the callee). The returning convention is equally simple: the callee writes results at the bottom of its frame and jumps to the return address found just below its frame.

Each return address can be mapped, via statically allocated data structures emitted by the compiler, to *frame information* that includes a kind (normally `CALL_RET_FRAME`), a frame size, and a pointer to an array that records the offsets of pointers within the frame that are live at the return address. To walk a call stack, a runtime-system function simply iterates over each frame by reading the return address pointed to by the stack-top pointer and decrementing the stack-top pointer by the size recorded in the corresponding frame info until the stack-top pointer is equal to the beginning of the stack object.

Threads and Signal-Handling. A thread is a heap-allocated object that pairs a call stack with additional data. A thread is an explicit object that can be manipulated by the source program.

One such manipulation is through a signal handler. The signal handler is an arbitrary SML function of type `Thread.t -> unit`; the signal handler receives the thread object of the thread that was interrupted by the arrival of the signal. Through the thread object, the signal handler has (indirect) access to the call stack of the interrupted thread.

5.1.2 Implementing **par** *in Source SML Code with Compiler Primitives.* To expose parallelism, MPL^s provides the user with a function par of type $(\texttt{unit} \to \alpha) \times (\texttt{unit} \to \beta) \to \alpha \times \beta$. One strategy would be to make par a compiler primitive and translate it to uses of **pcall**, **getjoin**, and **setjoin** primitives and/or IR constructs, as described in Section 3.2. However, **getjoin** and **setjoin** are really synchronization operations[1] that are implemented, along with the thread scheduler, in source SML code. It would be infeasible and impractical to bake these operations into the compiler.

Therefore, we take a different approach. We introduce a polymorphic higher-order pcall primitive and implement par using pcall in source SML code. The pcall primitive has the type $(\alpha \to \beta) \times \alpha \times (\beta \to \gamma) \times (\beta \to \gamma) \times (\texttt{unit} \to \delta) \to \gamma$, where pcall(f,x,seq,sync,spwn) roughly corresponds to the **pcall** $f(x) \triangleright (b_{\text{seq}}, b_{\text{sync}}, b_{\text{spwn}})$ transfer from Section 3, except that the pcall primitive returns the result of either seq or sync (implicitly joining their control-flow), while the **pcall** transfer allows the control-flow of b_{seq} and b_{sync} to be arbitrary. The only assumption that the compiler makes about the pcall primitive is that the spwn argument function does not terminate with a value and instead exits the executing thread (as is the semantics of **setjoin** in Section 3).

Using pcall, we can give a simplified implementation of par.

```
fun par (f: unit -> 'a, g: unit -> 'b): 'a * 'b =
    let fun seq  rf = (rf, g ())
        fun sync rf = let val j: 'b join = getPCallData () in (rf, getJoin j) end
        fun spwn () = let val j: 'b join = getPCallData ()
                      in  setJoin (j, g ()) ; Thread.exit ()  end
    in  pcall (f, (), seq, sync, spwn)  end
fun handler t = (... spawnOldestPCallAndSetData (t, newJoin ()) ... ; ())
val _ = Signal.setHandler (Signal.SIGALRM, handler)
```

The SML functions getJoin and setJoin use a (scheduler-defined) data structure of type β join to synchronize and pass a value of type β. Note that the join value is only required if the parallelism exposed by pcall is promoted and sync and spwn are executed. Rather than always executing newJoin when **par** is called, which would be unnecessary overhead when the parallelism is not promoted, newJoin is only executed when a promotion occurs and the join value is stored in both the new thread's copied frame and the promoted frame (similar to **child(_)** in the operational semantics of Section 3). The join value can be accessed within sync and spwn via a (polymorphic) primitive getPCallData used at the type $\texttt{unit} \to \beta$ join.

We use an interval timer to regularly deliver a SIGALRM signal to the program and install a signal handler that grants work tokens and attempts promotions, implementing our parallelism management algorithm (Section 4). The critical operation is a (polymorphic) primitive spawnOldestPCallAndSetData (t, d), which walks the call stack of an interrupted thread t, promoting the oldest promotable frame and storing (a pointer to) an arbitrary heap-allocated data object d in both the new thread's copied frame and the promoted frame; in practice, the data object is always a join value.

Note that while getPCallData and spawnOldestPCallAndSetData could be implemented by runtime-system functions (indeed, the latter is translated to a foreign C call to a runtime-system function late in compilation), it is nonetheless important that they are exposed as a compiler primitives. This is because various flow analyses in the compiler must "see" that values that flow in to spawnOldestPCallAndSetData flow out of getPCallData. Moreover, getPCallData can be implemented efficiently in the compiler when frames and the call stack are made explicit.

While the actual implementation of par in source SML code is somewhat more sophisticated, reifying and propagating exceptions raised by f and g and making use of additional (opaque) foreign

[1]Essentially, Id-style I-structures [Arvind et al. 1989], with additional support to integrate with MPL memory management.

C runtime-system functions, no additional primitives beyond `pcall`, `spawnOldestPCallAndSetData`, and `getPCallData` are required. The rest of this subsection therefore focuses on the compiler changes required to support these primitives.

5.1.3 Front-End Changes. None! No changes to the syntax or type checking of the language were made. Compiler primitives are exposed as SML functions in a generic manner and the `pcall`, `spawnOldestPCallAndSetData`, and `getPCallData` primitives required no special handling. Because Standard ML is a higher-order language, it is easy to expose the non-trivial control-flow of **pcall** as a higher-order primitive; if the host language were first-order, it would almost certainly require syntax changes in order to support **pcall**. The earliest phase of the compiler that required changes was the closure-conversion phase.

5.1.4 Closure-Conversion Changes. The closure-conversion phase of the compiler is responsible for transforming a higher-order IR into a first-order SSA-based IR, using *defunctionalization* [Reynolds 1972] guided by a monovariant whole-program control-flow analysis [Cejtin et al. 2000].

Although `pcall` is the first higher-order primitive to be added to the compiler, it posed little difficulty for the control-flow analysis or defunctionalization transformation. To analyze `pcall(f,x,seq,sync,spwn)`, we arrange for `x` to flow as an argument to `f`, the result of `f` to flow as an argument to both `seq` and `sync`, and the results of `seq` and `sync` to flow to the result of the `pcall`; additionally, we analyze `spwn` as though it were called. Defunctionalization of `pcall(f,x,seq,sync,spwn)` is only slightly more involved. The `pcall` primitive is translated to a `PCall` transfer with a new top-level function as the target, `f` and `x` as the arguments, and three new blocks b_{seq}, b_{sync}, and b_{spwn} as the continuations. The new top-level function takes `f` and `x` and performs the defunctionalized call of `f x`. Both b_{seq} and b_{sync} are of arity 1 (receiving the result of `f x` as a variable `rf`) and proceed to control-flow graphs that, respectively, perform the defunctionalized calls of `seq rf` and `spwn rf` and then join at the block receiving the result of the `pcall` primitive. b_{spwn} is of arity 0 and proceeds to a control-flow graph that performs the defunctionalized call of `spwn ()`.

5.1.5 SSA IR Middle-End Changes. Most of MLton's optimizations are performed on three simply-typed, first-order SSA-based IRs; these optimizations and supporting infrastructure make up over 20% of the compiler. Each SSA IR was extended with a `PCall` transfer similar to the **pcall** transfer of Section 3: taking a function name, a list of actual arguments, and three continuation blocks.

Although there are over 30 individual optimization passes on these SSA IRs, extending them to support `PCall` was greatly simplified by good compiler-construction techniques. For example, many optimization analyses require visiting the blocks of a function's control-flow graph in depth-first or dominator order. Such traversals are captured by infrastructure functions; it sufficed to update these infrastructure functions with the control-flow graph edges implied by the `PCall` transfer. Similarly, many optimization analyses are formulated as whole-program flow analyses that, among other things, require matching `Return` transfers with receiving continuation blocks. A generic flow analysis is provided by the infrastructure and it sufficed to update this generic flow analysis with support for `PCall`. Elsewhere, we could simply share code with or copy and lightly edit the code for the implementation of a (normal) `Call` transfer; with additional refactoring, more code could be shared between `Call` and `PCall` transfers.

One optimization pass that does treat `Call` and `PCall` differently is the inlining pass. A function cannot be inlined at a `PCall`, because that would eliminate the mark of potential parallelism. This is not disasterous, since the fork-join parallel algorithms that we consider make use of (non-tail) recursive functions that cannot be inlined. Also, note that while inlining *at* a `PCall` is disallowed, inlining at `Calls` *within* a `PCalled` function is allowed; for example, while the recursive and **pcall**ed go function from Figure 1 cannot be inlined, it is likely that the `map` function will be specialized to the argument function `f` and `f` can be inlined within `go`.

5.1.6 Frame, Calling Convention, and Back-End Changes. To mark and realize the potential parallelism of a **pcall**, we introduced three new forms of call-stack frames (see Figures 9b, 9c, and 9d).

A **pcall_seq** frame (Figure 9b) implements a **frame**(**promotable**($b_{\text{sync}}, b_{\text{spwn}}$), **none**, $\mathcal{X}, f, b_{\text{seq}}$). The frame info corresponding to $\texttt{RetAddr}_{\text{seq}}$ is given a new `PCALL_SEQ_RET_FRAME` kind. Since a **pcall_seq** frame can continue executing at any of its return addresses, the frame info corresponding to $\texttt{RetAddr}_{\text{seq}}$ records the set of live pointers as the *union* of the variables that are live at the beginning of $\texttt{RetAddr}_{\text{seq}}$, $\texttt{RetAddr}_{\text{sync}}$, and $\texttt{RetAddr}_{\text{spwn}}$.

We can describe the essence of the implementation of `spawnOldestPCallAndSetData (t, d)`. First, the oldest **pcall_seq** frame of the thread `t` is identified by walking the call stack and looking at frames with the `PCALL_SEQ_RET_FRAME` kind. Next, the **pcall_seq** frame is copied (including all of the local variables, which include the live/free variables of $\texttt{RetAddr}_{\text{spwn}}/b_{\text{spwn}}$ and which may be pointers to heap-allocated objects, but no copying of heap-allocated objects is necessary) to the bottom of a new call stack and the $\texttt{RetAddr}_{\text{spwn}}$ is written to the top of the copied frame and the heap-object pointer `d` is written just below, converting it to a **pcall_spwn** frame (Figure 9d; note that a **pcall_spwn** frame is necessarily the only frame on its call stack). Finally, the $\texttt{RetAddr}_{\text{sync}}$ is written to the top of the identified **pcall_seq** frame and the heap-object pointer `d` is written just below, converting it to a **pcall_sync** frame (Figure 9c).

The frame infos corresponding to $\texttt{RetAddr}_{\text{sync}}$ and $\texttt{RetAddr}_{\text{spwn}}$ can be given the `CALL_RET_FRAME` kind, since, these frames have no special behavior. Because the `DataPtr` slot of **pcall_sync** and **pcall_spwn** frames holds a heap-object pointer, it must be treated as a live pointer when tracing such frames, else the object could be garbage collected between the time of the promotion and the resumption of the frame. Therefore, the frame infos corresponding to $\texttt{RetAddr}_{\text{sync}}$ and $\texttt{RetAddr}_{\text{spwn}}$ include the `DataPtr` slot among the set of live pointers along with the variables that are live at the beginning of $\texttt{RetAddr}_{\text{sync}}$ and $\texttt{RetAddr}_{\text{spwn}}$, respectively.

Making a **pcall** is just as efficient as making a normal call: the caller writes arguments to the call stack just above its frame, writes the three appropriate return addresses to the top of its frame, and jumps to the callee; the callee, whether called or **pcall**ed, increments the stack-top pointer to the top of its frame. Moreover, returning from a **pcall** to either $\texttt{RetAddr}_{\text{seq}}$ or $\texttt{RetAddr}_{\text{sync}}$ is exactly the same as returning from a call: the callee writes results to the call stack at the bottom of its frame, decrements the stack-top pointer to just below the bottom of its frame, and jumps to the return address now pointed to by the stack-top pointer. Returning to $\texttt{RetAddr}_{\text{spwn}}$ is exactly the same as returning from a runtime-system function C call: jump to the return address pointed to by the stack-top pointer.

To minimize overhead, it is important that a callee's entry and exit operations are agnostic with respect to being called or **pcall**ed. This is enabled in our system by passing arguments and results *above* the return address(es), in the frame of the callee. It is also important that `spawnOldestPCallAndSetData` is able to locate the frame slots that are read and modified when promoting a **pcall_seq** frame. This is enabled in our system by using the slots *below* the $\texttt{RetAddr}_{\text{seq}}$, at fixed offsets relative to the top of the frame. MLton's existing calling convention made this a natural approach, but it is not the only one. Implementing **pcall** in another system may require thinking about the best way to adapt an existing calling convention to achieve these properties.

The changes to the back end of the compiler are minimal: associate the appropriate frame infos for the return addresses of a `PCall` transfer and implement the `PCall` transfer, which is identical to `Call` except for writing two additional return addresses to the top of the frame. Finally, the `getPCallData` primitive is implemented by fetching the `DataPtr` slot from the current frame, which is at a fixed offset relative to the top of the frame.

5.2 Parallelism Management

To implement our parallelism management algorithm (Section 4), we equip each thread in the system with a count of how many work tokens it has saved, and install a signal handler on a regularly delivered `SIGALRM` signal (specifically an interval timer) to generate tokens and perform promotions. Recall that our algorithm (Section 4) has two tunable parameters, C and N. In our implementation, N is the determined by the interval of the `SIGALRM` signal, and C is the number of tokens delivered to each active thread when the signal is received. We set N to 500μs, which is the smallest interval for which we could reliably receive signals at a regular rate, and then set C to 30, which comes out to at least $500/30 \approx 16\mu$s of work per promotion, on average.

5.3 Integration with Work-Stealing Scheduler

MPL^s schedules work onto processors with a work-stealing scheduler, inherited from MPL. This scheduler features a standard optimization known as the *clone optimization* [Frigo et al. 1998], which avoids the cost of fully instantiating a thread in the case where a spawned task is not stolen. Our approach integrates seamlessly with this optimization, resulting in three different representations for parallel tasks used throughout execution. (1) If **unpromoted**, a task is represented by a promotable frame. (2) If **promoted but not stolen**, a task is represented by a single heap-allocated frame, stored in a scheduler queue. If it stays unstolen, the only additional cost is removing it from the queue. (This is the "fast clone" of the clone optimization.) (3) If **promoted and stolen**, a task becomes a runnable thread, represented by a heap-allocated call stack and any associated scheduling metadata.

One remaining question is whether or not the token accounting algorithm (Section 4) can benefit from handling unstolen threads differently from stolen threads. We found that penalizing unstolen threads helped ensure that the number of unstolen threads remains low across different processor counts. Specifically, in the case where the child thread is unstolen, we throw away the child's tokens. Note that throwing away excess tokens is always safe, with respect to amortization of the cost of **par**.

5.4 Integration with MPL Memory Management

One final issue in our implementation is the integration with MPL's memory management system, which is largely specific to MPL and may not be an issue in other systems. The issue is that MPL allocates new hierarchical heaps at each **par**. This cost, although small, would nevertheless be prohibitively expensive if it were to happen at every **pcall**. Therefore, we only allocate new heaps when threads are spawned (i.e., at promotions). The impact is that some objects allocated by a child task instead appear to have been allocated by the parent, which can introduce additional down-pointers into the system, but is nevertheless safe for the disentanglement invariants [Westrick et al. 2020] that MPL's architecture relies on.

6 EVALUATION

We evaluate the performance of our implementation, called MPL^s ("Sugar MaPLe"). For our evaluation, we consider a benchmark suite consisting of both "fully parallel" benchmarks (Section 6.1) which use **par** liberally (without any manual granularity control), as well as manually tuned versions of the same benchmarks, which use manual granularity control to reduce the number of calls to **par** and optimize for performance. All our experiments in this section use exactly the same source code across different systems.

Our evaluation consists of four parts:

(1) In Section 6.3 we demonstrate the effectiveness of automatic parallelism management by comparing MPL^s against MPL [Acar et al. 2020], the compiler on which MPL^s is based. MPL features an eager implementation of **par**, which pays a modest cost for every call, as is typical in existing parallel systems. On the fully parallel benchmarks, we show that MPL^s can be as much as 14x faster than MPL on 64 cores, with an average of 4x faster. This performance advantage is due to MPL^s's guaranteed amortization of the cost of **par**.

(2) In Section 6.4, we evaluate the effectiveness of our token accounting algorithm presented in Section 4. For this evaluation, we compare against the original heartbeat algorithm (as proposed by Acar et al. [2018]), which is similar to our algorithm except that it does not attempt to save unused tokens. The results show an average of 24% improvement, with a max in one case of 3x.

(3) In Section 6.5, we measure low overheads (less than 2x on average) and good scalability (27x speedup on average) for the fully parallel benchmarks in comparison to MLton [MLton nd], which generates fast sequential code. These results show that MPL^s is able to simultaneously amortize the overhead of **par** without sacrificing parallelism.

(4) In Section 6.6, we investigate the impact of manual granularity control by comparing the fully parallel benchmarks against their manually tuned counterparts. That is, we compare the performance of *different benchmark codes* on the *same system* (as opposed to the other comparisons, which consider the *same benchmark codes* on *different systems*). By comparing the performance gap between untuned (fully-parallel) and manually tuned codes, we determine how much each system individually relies on manual tuning for performance. Our results show that MPL relies heavily on manual tuning, with an average performance gap of 7x on 64 cores, and a maximum of 46x in one case. In contrast, for MPL^s, manual tuning offers less than 2x performance improvement on average, with a max of 6x in one case. On 64 cores, the maximum is only 3.3x. This demonstrates that our approach significantly reduces the need for programmers to manually tune and prune parallelism by hand.

6.1 Benchmarks

We consider a number of parallel benchmarks taken from the Parallel ML Benchmark Suite [Westrick 2022], covering multiple problem domains, including graph processing, computational geometry, text analysis, numerical algorithms, etc. These benchmarks are mostly written in terms of library functions (`map`, `scan`, `reduce`, `filter`, etc.) with manual granularity control in the form of constant thresholds, hard-coded into the program source. Often, grain sizes had been tuned per call-site, by passing a grain size as an additional argument to a function. For example, it is common to see code such as $\texttt{map}(1000, f, A)$, where the first argument is the grain size.

As we discuss below in more detail, we modified these benchmarks to remove any manual pruning of parallelism (e.g., granularity control via hard-coded constant thresholds), producing a new version of the benchmarks which is "fully parallel". These modified benchmarks are what we use for our evaluation.

Fully parallel programs. One especially common form of manual pruning found in real-world code is manual granularity control in the form of hardcoded constant thresholds. We investigated the use of hardcoded thresholds throughout the benchmarks, and classified each threshold into one of two categories: (1) par-grains, whose purpose is to optimize performance by reducing the number of calls to **par**, thus amortizing its overhead, and (2) algorithmic grains, whose purpose is

Table 1. Times (seconds), performance improvements of MPL^s relative to MPL, and analysis of number of **par**s and promotions, for the fully parallel benchmarks.

	T_1			T_{64}						
	MPL	MPL^s (Ours)	$\frac{\text{MPL}}{\text{MPL}^s}$	MPL	MPL^s (Ours)	$\frac{\text{MPL}}{\text{MPL}^s}$	#**par**	#promote	$\frac{\#\textbf{par}}{\#\text{promote}}$	$\frac{\text{work}(\mu s)}{\text{promotion}}$
bfs	7.47	4.32	**1.73**	.281	.115	**2.44**	36 M	76 K	473	57
bignum-add	4.39	1.48	**2.97**	.218	.028	**7.79**	32 M	17 K	1833	85
delaunay	7.38	7.56	**0.98**	.275	.290	**0.95**	2.0 M	.18 M	11.2	42
grep	9.55	3.87	**2.47**	.406	.076	**5.34**	60 M	45 K	1320	86
linefit	10.7	2.58	**4.15**	.518	.066	**7.85**	67 M	36 K	1890	73
mandelbrot	4.52	2.87	**1.57**	.163	.049	**3.33**	17 M	29 K	603	100
map-heavy	3.63	3.69	**0.98**	.058	.060	**0.97**	3.9 K	3.0 K	1.26	1210
map-light	17.3	3.93	**4.40**	.773	.078	**9.91**	.13 B	45 K	2996	88
msort	7.14	4.99	**1.43**	.222	.093	**2.39**	19 M	58 K	333	86
nearest-nbrs	1.39	1.25	**1.11**	.031	.026	**1.19**	1.5 M	18 K	83.5	71
nqueens	8.23	3.07	**2.68**	.431	.053	**8.13**	55 M	31 K	1791	101
primes	21.1	5.75	**3.67**	.942	.124	**7.60**	.14 B	73 K	1970	79
sparse-mxv-csr	15.5	4.32	**3.59**	.547	.088	**6.22**	78 M	48 K	1633	90
suffix-array	6.58	3.79	**1.74**	.318	.084	**3.79**	29 M	36 K	792	105
triangle-count	15.7	9.20	**1.71**	.714	.168	**4.25**	54 M	95 K	566	97
wc	7.80	1.84	**4.24**	.464	.032	**14.50**	54 M	19 K	2901	99
min			**0.98**			**0.95**				
geomean			**2.17**			**4.09**				
max			**4.40**			**14.50**				

to optimize performance by switching between different algorithms. For example, a divide-and-conquer implementation of `reduce` uses a par-grain to decide whether or not the two recursive calls should be evaluated in parallel or sequentially. In contrast, as an example of an algorithmic grain, consider mergesort, where it is beneficial to switch to a different sorting algorithm (e.g., insertion sort) at small sizes due to improvements in constant factors.

We found that par-grains were often exposed as additional arguments to library functions, and needed to be tuned per call-site; in contrast, algorithmic grains were often hard-coded into the implementation and therefore did not need to be exposed to the client of the library. As discussed in earlier sections, par-grains are typically at odds with high-level programming, especially the use of higher-order functions and polymorphism.

We consider a program to be *fully parallel* if it uses no par-grains. In the fully parallel versions of our benchmarks, we removed the par-grains, not the algorithmic grains.

6.2 Experimental Setup

We run all our experiments on a 64-core AWS `r6i.32xlarge` instance, which is a two-socket machine equipped with two 2.9GHz Intel Xeon (32-core) Platinum 8375C CPUs. The machine has 1TB of memory, and runs Amazon Linux version 2 with Linux kernel version `5.10.135-122.509.amzn2.x86_64`. In Section 6.5, we use MLton version `20210117`, and in Section 6.3, we use MPL version `0.3`.

To measure timings for a benchmark, we first run the benchmark back-to-back for at least 5 seconds as a warmup. We then run the benchmark 20 times back-to-back. All of this occurs in the same program instance. The time we report is the average of the 20 runs (after the warmup, which is not included in the measurements).

6.3 Result #1: MPL^s Consistently Outperforms Eager par

In this section, we measure how effectively MPL^s amortizes the cost of parallelism. We do so by comparing against MPL, which is identical to our MPL^s except for the implementation of **par**. In particular, MPL features an "eager" implementation of **par** which immediately allocates a closure for the spawned task and pushes this closure onto the scheduler queue. This cost, although small, accumulates over may calls to **par** and degrades performance. In contrast, we show that our implementation of **par** in MPL^s is significantly more efficient and scalable.

The results of this comparison are presented in Table 1, which shows the performance of the fully parallel benchmarks using both MPL and MPL^s on 1 and 64 processors. The columns labeled $\frac{\text{MPL}}{\text{MPL}^s}$ compute the improvement of MPL^s relative to MPL. This number summarizes the performance advantage of our approach (higher is better), which is due to MPL^s's guaranteed amortization of the cost of **par**.

On a single processor, we observe that MPL^s is as much as 4 times faster than MPL, with an average improvement of 2x. On 64 processors, the gap is larger, with an improvement of as much as 14x in comparison to MPL, and an average of 4x. This shows that not only does MPL^s amortize the work cost of **par**, but it also scales better. The improvement in scalability appears to be due to a significant reduction in the number of unnecessary thread spawns, which increase memory pressure in the memory management system, and contention in the scheduler.

To further investigate the advantage of our approach, we report additional statistics for each benchmark in Table 1, including the number of calls to **par**, the number of promotions performed by MPL^s, and the average amount of work (in microseconds) performed per promotion.

We observe that almost all benchmarks call **par** millions of times, with some as many as hundreds of millions. In contrast, the number of promotions is typically three orders of magnitude smaller. The column $\frac{\#\textbf{par}}{\#\text{promote}}$ measures the specific ratio for each benchmark. Except for two outliers (`delaunay` and `map-heavy`, discussed below), we observe that there are typically as many as 100-1000x more calls to **par** as there are promotions. MPL pays a modest cost per call, which adds up to a significant impact on performance. In contrast, our approach reduces the cost of **par** by multiple orders of magnitude.

There are two cases where MPL^s is not faster than MPL: `delaunay` and `map-heavy`, where MPL^s is approximately 5% slower than MPL. Both of these benchmarks naturally amortize the cost of **par**; this is evidenced by a low number of **par**s per promotion as reported in Table 1. We therefore do not expect MPL^s to be faster than MPL on these two benchmarks. Rather, our results show that even when our approach is not needed, MPL^s is still able to match existing techniques in performance.

Finally, in the final column, we report the average work performed by MPL^s per promotion (computed as T_1/#promote). Our parallelism management algorithm guarantees a lower bound on this quantity on average, controllable by tuning the parameters C and N discussed in Section 4 (which are set once for the system and do not need to be tuned per benchmark). Based on the parameters set for MPL^s, we expect a lower bound of approximately 16μs or higher (see Section 5.2), and we observe that the measured values here are consistent with that predicted lower bound. We also separately confirmed that by adjusting C and N, we control the average work performed per promotion, increasing and decreasing it as desired.

6.4 Result #2: Token Accounting Outperforms Classic Heartbeat Scheduling

We evaluate the effectiveness of our token accounting algorithm (Section 4) in comparison to the original heartbeat algorithm proposed by Acar et al. [2018]. Our token accounting algorithm refines the heartbeat scheduling algorithm to extract more parallelism while retaining provable

Table 2. Single-core overheads and 64-core speedups of MPL^s in comparison to MLton on the fully parallel benchmarks.

Benchmark	Baseline MLton	Overhead $\frac{T_1(\text{MPL}^s)}{\text{MLton}}$	Speedup $\frac{\text{MLton}}{T_{64}(\text{MPL}^s)}$
bfs	2.79	**1.55**	**24**
bignum-add	.802	**1.85**	**29**
delaunay	5.16	**1.47**	**18**
grep	1.76	**2.20**	**23**
linefit	1.67	**1.54**	**25**
mandelbrot	2.03	**1.41**	**41**
map-heavy	2.92	**1.26**	**49**
map-light	1.46	**2.69**	**19**
msort	3.46	**1.44**	**37**
nearest-nbrs	.913	**1.37**	**35**
nqueens	1.28	**2.40**	**24**
primes	2.42	**2.38**	**20**
sparse-mxv-csr	1.75	**2.47**	**20**
suffix-array	2.31	**1.64**	**28**
triangle-count	5.10	**1.80**	**30**
wc	.937	**1.96**	**29**
min		**1.26**	**18**
geomean		**1.79**	**27**
max		**2.69**	**49**

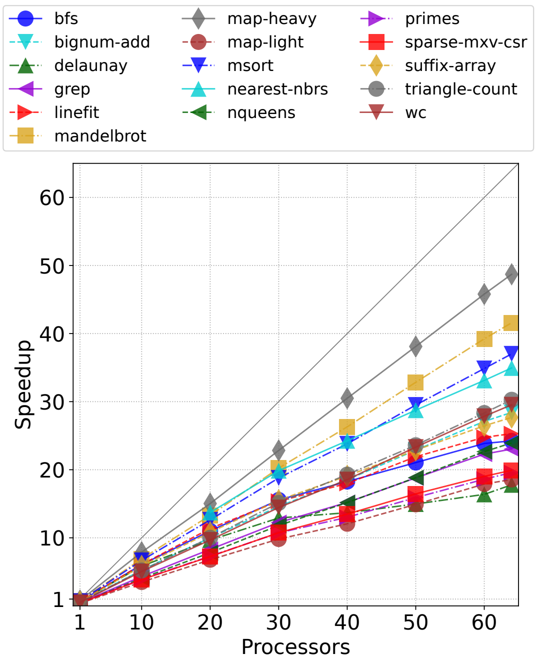

Fig. 10. Speedups of MPL^s over MLton on fully parallel benchmarks.

amortization. Roughly speaking, the difference is that our algorithm saves unused tokens, whereas the original heartbeat algorithm immediately throws away all unused tokens. We do not compare directly with either of the prior implementations of heartbeat scheduling [Acar et al. 2018; Rainey et al. 2021], because the underlying systems are too different. Instead, we implemented heartbeat scheduling in our system for this comparison, which was straightforward, as the promotions required naturally align with our **pcall** semantics.

The results of this comparison are shown in Table 3. The columns report the improvement (expressed as a ratio) in running time of our algorithm, in comparison to the original heartbeat algorithm. Because our algorithm extracts more parallelism but has the same work-efficiency guarantee, we do not expect to see a difference on a single processor; indeed, we observe on average no change in performance in this case. On 64 processors, however, the improvement is significant, with an average of 24% improvement and a max of 3x improvement in one case (`delaunay`). In the case of `delaunay`, the improvement appears to be due to heartbeat delays accumulating along the critical path; in contrast, our algorithm is able to eagerly spend excess tokens as soon as each **par** occurs, with no delay.

6.5 Result #3: Good Scalability and Low Single-Core Overheads

We compare against MLton, on which MPL^s is based. In this comparison, we compile the sequential elision of each fully parallel benchmark with MLton, and use this binary as the baseline. In the sequential elision, each **par**(f, g) is replaced with the sequential tuple $(f(), g())$. This produces an equivalent sequential program which is otherwise identical to the original benchmark, allowing us to determine the overhead of scalability of MPL^s as a “whole package” (i.e., including parallel memory management overheads, etc.)

Table 3. Improvement factors due to token accounting, in comparison to heartbeat scheduling on the fully parallel benchmarks.

	$P = 1$	$P = 64$
bfs	0.94	1.51
bignum-add	1.00	1.25
delaunay	1.01	3.23
grep	1.00	1.14
linefit	0.98	1.05
mandelbrot	0.99	1.02
map-heavy	1.00	1.03
map-light	0.97	1.03
msort	1.06	1.17
nearest-nbrs	1.00	1.77
nqueens	1.02	1.08
primes	0.97	1.12
sparse-mxv-csr	1.41	1.10
suffix-array	0.97	1.44
triangle-count	0.98	1.00
wc	0.96	1.03
min	0.94	1.00
geomean	1.01	1.24
max	1.41	3.23

Table 4. Times (in seconds) of manually-tuned benchmarks, and the performance overhead of fully-parallel programming relative to manually tuned, for each of MPL and MPL^s. Smaller overheads are preferable. (See Table 1 for the reported fully-parallel timings.)

	T_1(manual)		$\frac{T_1\,\text{(fully-par)}}{T_1\,\text{(manual)}}$		T_{64}(manual)		$\frac{T_{64}\,\text{(fully-par)}}{T_{64}\,\text{(manual)}}$	
	MPL	MPL^s (Ours)	MPL	MPL^s (Ours)	MPL	MPL^s (Ours)	MPL	MPL^s (Ours)
bfs	3.03	3.15	**2.47**	**1.37**	.079	.083	**3.56**	**1.39**
bignum-add	3.31	3.20	**1.33**	**0.46**	.054	.054	**4.04**	**0.52**
delaunay	7.07	7.33	**1.04**	**1.03**	.238	.261	**1.16**	**1.11**
grep	2.00	2.02	**4.78**	**1.92**	.035	.036	**11.60**	**2.11**
linefit	.352	.455	**30.40**	**5.67**	.020	.020	**25.90**	**3.30**
mandelbrot	1.62	1.63	**2.79**	**1.76**	.026	.026	**6.27**	**1.88**
map-heavy	3.62	3.66	**1.00**	**1.01**	.057	.059	**1.02**	**1.02**
map-light	.920	.996	**18.80**	**3.95**	.031	.032	**24.94**	**2.44**
msort	3.84	4.26	**1.86**	**1.17**	.068	.074	**3.26**	**1.26**
nearest-nbrs	1.15	1.17	**1.21**	**1.07**	.023	.024	**1.35**	**1.08**
nqueens	1.40	1.38	**5.88**	**2.22**	.024	.024	**17.96**	**2.21**
primes	1.82	1.86	**11.59**	**3.09**	.052	.052	**18.12**	**2.38**
sparse-mxv-csr	1.71	1.85	**9.06**	**2.34**	.040	.042	**13.68**	**2.10**
suffix-array	4.55	4.65	**1.45**	**0.82**	.093	.097	**3.42**	**0.87**
triangle-count	3.90	3.96	**4.03**	**2.32**	.072	.076	**9.92**	**2.21**
wc	.608	.620	**12.83**	**2.97**	.010	.010	**46.40**	**3.20**
min			**1.00**	**0.46**			**1.02**	**0.52**
geomean			**3.89**	**1.72**			**6.87**	**1.63**
max			**30.40**	**5.67**			**46.40**	**3.30**

The results of this comparison are shown in Table 2 and Figure 10. In Table 2, we report the time of MLton on each benchmark and compare against the times of MPL^s on 1 and 64 processors. The column labeled "Overhead" is the ratio MPL^s single-core time to MLton's time; this ratio summarizes the overhead of parallelism, including the amortization of **par** as well as parallel memory management overheads. The column labeled "Speedup" is the ratio of MLton's time to MPL^s's time on 64 cores; this ratio is the speedup of MPL^s on 64 cores in comparison to MLton. In Figure 10, we also plot these speedups across different numbers of processors.

In Table 2, we observe that MPL^s achieves low overheads on a single processor in comparison to MLton. Overheads are reasonable, ranging from 1.3x to 2.7x, with an average of 1.8x. In other words, users of MPL^s should typically need only 2-3 processors to see a performance improvement over the sequential elision, even when writing "fully parallel" programs.

On 64 processors, we observe that MPL^s achieves speedups between 18x and 49x in comparison to MLton, with an average of 27x. Furthermore, we observe in Figure 10 that the speedup over MLton increases approximately linearly as the number of processors increases. These results confirm that MPL^s preserves the scalability of these benchmarks. We conclude that MPL^s is able to provide predictable speedups and good scalability across a range of machine sizes.

6.6 Result #4: Low Overhead Relative to Manual Tuning

We investigate the performance impact of manual tuning for both MPL and MPL^s by comparing the fully parallel benchmarks against their manually tuned counterparts. The manually tuned benchmarks were originally written and tuned for best performance with MPL, and use a variety of manual granularity control strategies, especially constant thresholds hardcoded into the program source, as described in Section 6.1. We did not attempt to re-tune these benchmarks for $\mathsf{MPL}^{\mathsf{s}}$.

Table 4 reports the performance of the manually tuned benchmarks, and overheads of the fully-parallel benchmarks relative to their manually-tuned counterparts. Overheads are computed individually per system; for example, in the first bolded column, we use MPL to run two versions of the same benchmark (one fully parallel, and the other manually tuned), and report the performance gap. The second bolded column does the same, but for $\mathsf{MPL}^{\mathsf{s}}$. In other words, these columns report the performance gains a programmer could expect to achieve if they went through the effort of carefully optimizing their code by pruning parallelism. Lower ratios are preferable, as these indicate less reliance on manual granularity control for performance.

We immediately observe that MPL relies heavily on manual tuning for performance, with an average gap of 7x on 64 cores, and a maximum of 46x in one case. That is, with MPL, if the programmer does not manually control granularity, they risk a slowdown of as much as an order-of-magnitude (or more). In contrast, $\mathsf{MPL}^{\mathsf{s}}$ offers a gap of less than 2x on average. On a single core, the max is 5.7x; on 64 cores, the max is only 3.3x. We conclude that $\mathsf{MPL}^{\mathsf{s}}$ greatly reduces the need for programmers to manually tune the amount of parallelism in their programs. In future work, we believe that our approach in $\mathsf{MPL}^{\mathsf{s}}$ could be refined to further close the gap between fully-parallel and manually tuned performance.

7 DISCUSSION

Our approach in this paper was to extend the compiler with special support for automatic parallelism management. It is worth considering whether or not we could achieve a similar result without modifying the compiler. We attempted to do so (as we describe below), and found that the overheads were too high. The challenge is that any cost incurred per call to **par**, regardless of how small, will accumulate and significantly degrade performance. This is due to the sheer number of calls to **par**: in practice, we can expect on the order of 10-100 million **par**s per second (Section 6).

We previously attempted to implement **par** at the source level (in a library) by maintaining, per thread, a stack of promotable (or promoted) tasks. For the purposes of discussion here, we will refer to these as *promotion stacks*. The idea was for **par** to push and pop tasks from the promotion stack, and promotions would modify elements of the promotion stacks (i.e., mark tasks as promoted, allowing **par** to then check for this appropriately). The problem is that every call to **par** pays for pushing and popping tasks from the promotion stack, and also pays to check if a promotion has occurred. The cost of these operations, although small, becomes significant due to the high rate of calls to **par**. Optimizing these costs at the source level is difficult due to the high-level nature of Standard ML, where we do not have control over the exact memory representation and layout of the tasks and stack elements. As a result, operations on the promotion stack incur unnecessary memory costs (especially allocations for the tasks) and branching instructions.

We were able to avoid these costs by integrating with the compiler. In particular, we emphasize that our **pcall** primitive requires no branching instructions in the unpromoted case. Futhermore, the memory cost in the unpromoted case is exactly two additional call-stack slots (as described in Section 5.1.6 and shown in Figure 9).

8 RELATED WORK

Language support for task parallelism. This line of work dates back to the 1980s, as exemplified by multiLisp [Halstead 1984], to the 1990s with NESL [Blelloch 1996] and Cilk [Frigo et al. 1998], and from the 2000s to the present in several extensions of Java [Bocchino et al. 2009; Imam and Sarkar 2014; Lea 2000], parallel Haskell [Li et al. 2007; Marlow and Peyton Jones 2011; Peyton Jones et al. 2008], several forms of parallel ML [Arora et al. 2021, 2023; Fluet et al. 2011, 2007; Guatto et al. 2018;

Raghunathan et al. 2016; Sivaramakrishnan et al. 2014; Spoonhower 2009; Westrick et al. 2020], and X10 [Charles et al. 2005]. To express parallelism, these languages use several different kinds of primitives, such as fork-join, futures, or async-finish, which are closely related but also differ in their expressiveness [Acar et al. 2016]. We focus in this paper on the fork-join paradigm, specifically the **par** primitive, which is well-suited for expressing balanced divide-and-conquer style parallel algorithms. In future work, we hope to extend our approach for other forms of parallelism.

A common foundation for task-parallel languages is the greedy-scheduling idea developed by Brent's seminal work [Brent 1974]. This work was the first to establish a bound for scheduling a task-parallel program: for any program with work W and span S, using P processors and any a level-by-level scheduler, the running time is upper bounded by $\frac{W}{P} + S$. Later work generalized the bound so that it holds for any greedy scheduler [Arora et al. 2001; Eager et al. 1989]. From these early results came the randomized work-stealing scheduler of Blumofe and Leiserson [Blumofe and Leiserson 1999] and its generalization by Arora et al. [2001], which can generate greedy schedules for fork-join parallel programs, while also including certain scheduling costs, e.g., steals. Other considerations of task scheduling have been analyzed, such as data locality [Acar et al. 2015, 2002; Blelloch and Gibbons 2004; Chowdhury and Ramachandran 2008; Lee et al. 2015; Spoonhower et al. 2009], and responsiveness [Muller et al. 2020; Muller and Acar 2016; Muller et al. 2017, 2018, 2023, 2019]. All of this work assumes that the spawning of a thread has unit or asymptotically constant cost.

Lazy scheduling and clone optimization. Lazy scheduling was introduced as a way of mitigating task-related overheads in the early work of Mohr et al. [1991]. The idea was adapted for the work stealing scheduler of Cilk-5 [Frigo et al. 1998], thereby introducing a general technique known as of *clone optimization*. Clone optimization avoids the cost of spawning tasks when both branches of a parallel fork point are scheduled on the same processor. When such a case is detected, the optimization reuses the current stack and avoids a synchronization operation before executing the join continuation. This optimization is used many systems, including ours. However, we found that the clone optimization by itself is insufficient for efficiency, and that finding prompted our work on automatic parallelism management.

Backtracking-Based Load Balancing is a variant of lazy scheduling that minimizes task-related overheads of C++ programs [Hiraishi et al. 2009]. When it enters a fork point, the scheduler leaves behind bookkeeping data for potentially spawning a new task, like in our approach. The potential is realized, in contrast, by polling a core-local cell, and spawning a task if there is an outstanding request from a remote core.

Lazy Binary Splitting (LBS) [Tzannes 2012; Tzannes et al. 2010, 2014] and Lazy Tree Splitting (LTS) [Bergstrom et al. 2012] are variants of lazy scheduling that optimize parallel do-all loops and tree traversals, respectively. LTS in particular focuses on traversals of tree data structures, which worked well for benchmarks in Manticore [Bergstrom et al. 2012; Fluet et al. 2011, 2007]. However, our setting is more general, supporting a wide variety of data structures (especially those based on mutable arrays). Both LBS and LTS follow the policy of promoting innermost parallelism, a policy with known limitations with respect to scalability [Tzannes et al. 2014]. For this reason, recent efforts have focused on variants that promote outermost parallelism. Although implementing outermost parallelism faces challenges, such as those we described in this paper, the outermost-promotion policy is, to the best of our knowledge, the only one backed by end-to-end time bounds. Given such bounds, one can guarantee work efficiency and preservation of scalability for all fork-join programs.

Heartbeat Scheduling. Our algorithm is inspired by heartbeat scheduling [Acar et al. 2018], with some key differences. The idea of heartbeat scheduling is to rely on a periodic pulse—i.e., a *heartbeat*—to trigger promotions. Specifically, at every heartbeat, every active thread tries to perform one promotion. By tuning the heartbeat interval, it is possible to guarantee that, for any fork-join program, the total work of promotion is bounded throughout execution. For example, if promotion takes 1 unit of time, then heartbeat scheduling can guarantee 1% overhead for promotions by triggering a heartbeat once every 100 units of time. Perhaps surprisingly, this policy prunes away parallelism by at most a constant factor, and therefore also guarantees span-efficiency (preservation of asymptotic parallelism).

The original heartbeat work demonstrated a prototype interpreter, written in C++, that operated over an explicit, hand-rolled AST structure of a client program. Its implementation drives the heartbeat pulse by software polling [Basu et al. 2021; Feeley 1993; Ghosh et al. 2020], which is similar to how our implementation implements beats. Subsequent work showed how to eliminate the interpretive overhead and gain finer control over work efficiency by using a Task-Parallel Assembly Language (TPAL) [Rainey et al. 2021], and used an alternative driver for the heartbeat, based on hardware interrupts.

We tried a version of heartbeat scheduling that, as described in Section 5, uses a hybrid of software polling and hardware interrupts, but found that the performance in some cases was poor (Section 6.4). The reason was that heartbeats need to occur at a very high rate to ensure good performance, which we were not able to achieve with our system. Specifically, on modern machines, an ideal heartbeat interval is in the range of 10-100 microseconds [Rainey et al. 2021]. Achieving this rate is difficult, even in a low-level setting [Basu et al. 2021; Ghosh et al. 2020; Hale and Dinda 2018; Rainey et al. 2021]. In our setting, the challenge is exacerbated: our language is high-level, with automatic memory management, which introduces unexpected delays during execution for additional work such as garbage collection.

Our work addresses this issue by reducing the pressure to have such a regular, high-frequency beat. In particular, our token accounting approach is capable of "simulating" frequent, regular heartbeats by decoupling promotions from heartbeats. Essentially, we use heartbeats only to track the amount of work performed, and then separately trigger promotions between heartbeats. Moreover, our algorithm retains performance guarantees (work- and span-efficiency). We expect that, with a reasonable implementation effort, we can improve performance by using a custom signal-delivery mechanism [Rainey et al. 2021].

Granularity control. Another approach to taming task-creation costs is given by granularity control, a family of algorithms in which the program ensures that each task holds onto a sizeable amount of useful work. Granularity control amortizes task-related costs by switching between parallel to serial modes of execution, and makes switching decisions based on *predictions* of *future* amounts of work, whereas alternatives, such as heartbeat scheduling and ours, switch based on *known* amounts of *past* work. At its most basic, granularity control may be performed manually by the application programmer [Intel 2011]. In manual granularity control, the programmer inserts into every parallel region a condition for switching between parallel and serial versions, and uses a hand tuning process to find suitable switching conditions. This approach faces severe limitations, as shown by Tzannes et al. [2010], owing largely to the difficulty of predicting execution time in an accurate and portable manner.

Starting from the late 1980s, there have been a number of approaches proposed to address the limitations of manual granularity control [Duran et al. 2008; Huelsbergen et al. 1994; Loidl and Hammond 1995; Pehoushek and Weening 1990; Shen et al. 1999; Weening 1989]. These techniques depend on certain assumptions, e.g., linear work complexity of all functions [Huelsbergen et al. 1994], or on sources of dynamically collected data, such as recursion-tree depth and various dynamic load conditions. Such data is typically limited and, as such, granularity decisions based on them risk decreasing parallelism adversely. Some alternative approaches base switching decisions on predictions of running times that are typically expressed in terms of some abstract quantity of computational steps, such as dynamically generated predictions based on user annotations in the source code [Lopez et al. 1996] or on statically generated predictions based on compiler-based analysis [Iwasaki and Taura 2016]. Such predictions are, however, vulnerable to the hardness of timing prediction on modern chip architectures.

Subsequent work on Oracle Guided Granularity Control addresses this issue by combining static and dynamic profiling data [Acar et al. 2019, 2011]. In the approach, each parallel region in the program is annotated with an *abstract cost function*, which provides a coarse estimation of the number of operations to be performed by the parallel region. This abstracted prediction is refined by an online prediction algorithm, which uses a profiler. The profiler takes samples from regions of the program as it runs and uses them, along with the results of the cost functions, to predict a number of cycles per unit of abstract work. This approach has the advantage that it has guaranteed time bounds for a well-defined subset of fork-join programs can be implemented entirely as a library. However, the approach requires specifying cost functions, which are not always easy or even possible to express properly, and its performance bounds only a certain somewhat well-behaved subset of fork-join program, whereas ours applies to all.

Another approach to granularity control, as developed recently by Rainey [2023], is to manually integrate heartbeat scheduling into a parallel kernel after a series of source-to-source refactoring steps (including transformations such as CPS-conversion and defunctionalization). This requires no changes to the compiler, but does essentially require the programmer to "manually compile" their program, which in general is infeasible, especially for large programs.

Embedding task parallelism in compiler IRs. Tapir [Schardl et al. 2017] is a recent proposal for embedding task parallelism in the middle end of LLVM. The purpose of the embedding is to enable LLVM's middle end to apply optimizations that previously were only applicable to purely serial regions of code. To this end, Tapir extends the LLVM IR with a few intrinsics for expressing potentials for parallel execution. In this regard, our IR extensions bear resemblance to Tapir, and although we do not analyze it, in principle, our IR extensions can unlock optimizations that were only possible in serial regions of code. The main contributions of Tapir differ from ours, however, in that our aim is for automatic management of parallelism.

9 CONCLUSION

In the current state of the art, parallel programmers are expected to manually hand-optimize code to control the cost of parallelism and ensure efficiency and scalability. Such optimizations are complex, require deep expertise, and result in code that is difficult to reason about and deploy, especially on different architectures.

Motivated by this challange and the effectiveness of language-level abstractions such as (automatic) memory management, we ask: can we manage parallelism in the run-time system automatically, so that its costs are controlled without harming performance? We answer this question in the affirmative by combining static and dynamic techniques for creating and managing parallelism. Specifically, we propose a *potentially parallel call* primitive and support this primitive in the run-time system by amortizing the cost of parallelism against actual (sequential) work. Our implementation, which extends the MPL language for Parallel ML, shows that the approach is practical and effective: (1) its implementation is not particularly onerous, (2) it significantly reduces the performance overhead of programming without manual granularity control, and (3) it performs well, delivering small overheads and good scalability (speedups).

DATA AVAILABILITY STATEMENT

The implementation and experiments for this paper are open-source and publicly available on GitHub at https://github.com/MPLLang/mpl and https://github.com/MPLLang/parallel-ml-bench.

ACKNOWLEDGMENTS

This research was supported by the following NSF grants (CCF-1901381, CCF-2115104, CCF-2119352, CCF-2107241) and by a gift from Intel.

REFERENCES

Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2019. Provably and Practically Efficient Granularity Control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 214–228. https://doi.org/10.1145/3293883.3295725

Umut A. Acar, Jatin Arora, Matthew Fluet, Ram Raghunathan, Sam Westrick, and Rohan Yadav. 2020. MPL: A High-Performance Compiler for Parallel ML. https://github.com/MPLLang/mpl.

Umut A. Acar, Guy Blelloch, Matthew Fluet, Stefan K. Muller, and Ram Raghunathan. 2015. Coupling Memory and Computation for Locality Management. In *Summit on Advances in Programming Languages (SNAPL)*.

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The Data Locality of Work Stealing. *Theory of Computing Systems* 35, 3 (2002), 321–347.

Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. 769–782.

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2011. Oracle Scheduling: Controlling Granularity in Implicitly Parallel Languages. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 499–518.

Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. 2016. Dag-calculus: A Calculus for Parallel Computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. 18–32.

Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space Efficient Parallel Functional Programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

Jatin Arora, Sam Westrick, and Umut A. Acar. 2023. Efficient Parallel Functional Programming with Effects. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1558–1583. https://doi.org/10.1145/3591284

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* 34, 2 (2001), 115–144.

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 598–632.

Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. 2021. Frequent Background Polling on a Shared Thread, Using Light-Weight Compiler Interrupts. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1249–1263. https://doi.org/10.1145/3453483.3454107

Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures*. 133–144.

Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2012. Lazy Tree Splitting. *J. Funct. Program.* 22, 4-5 (Aug. 2012), 382–438.

Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling irregular parallel computations on hierarchical caches. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 355–366.

Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively sharing a cache among threads. In *SPAA* (Barcelona, Spain).

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46 (Sept. 1999), 720–748. Issue 5.

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*

(Orlando, Florida, USA) *(OOPSLA '09)*. 97–116.

Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (1974), 201–206.

Henry Cejtin, Suresh Jagannathan, and Stephen T. Weeks. 2000. Flow-directed Closure Conversion for Typed Languages. In *European Symposium on Programming*. 56–71.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, 519–538.

Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*.

Rezaul Alam Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures* (Munich, Germany). ACM, New York, NY, USA, 207–216.

A. Duran, J. Corbalan, and E. Ayguade. 2008. An adaptive cut-off for task parallelism. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

Derek L. Eager, John Zahorjan, and Edward D. Lazowska. 1989. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing* 38, 3 (1989), 408–423.

Martin Elsman. 1999. Static Interpretation of Modules. In *International Conference on Functional Programming*. 208–219.

Marc Feeley. 1993. Polling efficiently on stock hardware. In *Proceedings of the conference on Functional programming languages and computer architecture* (Copenhagen, Denmark) *(FPCA '93)*. 179–187.

Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 1–11.

Jeremy T. Fineman. 2005. *Provably Good Race Detection That Runs in Parallel*. Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. *SIGPLAN Not.* 44, 6 (June 2009), 121–133. https://doi.org/10.1145/1543135.1542490

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (2011), 1–40.

Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming* (Nice, France) *(DAMP '07)*. 37–44.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.

Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. 2020. Compiler-Based Timing For Extremely Fine-Grain Preemptive Parallelism. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00057

Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. 81–93.

Kyle C. Hale and Peter A. Dinda. 2018. An Evaluation of Asynchronous Software Events on Modern Hardware. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 355–368. https://doi.org/10.1109/MASCOTS.2018.00041

Robert H. Halstead, Jr. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (Austin, Texas, United States) *(LFP '84)*. ACM, 9–17.

Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. 2009. Backtracking-based load balancing. *Proceedings of the 2009 ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* 44, 4 (February 2009), 55–64.

Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. 1994. Using the Run-time Sizes of Data Structures to Guide Parallel-thread Creation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming* (Orlando, Florida, USA) *(LFP '94)*. 79–90.

Shams Mahmood Imam and Vivek Sarkar. 2014. Habanero-Java library: a Java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*. 75–86.

Intel. 2011. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/.

Shintaro Iwasaki and Kenjiro Taura. 2016. A static cut-off for task parallel programs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 139–150.

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June*

18-23, 2017, Albert Cohen and Martin T. Vechev (Eds.). ACM, 157–170.

Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (San Francisco, California, USA) *(JAVA '00)*. 36–43.

I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-Fly Pipeline Parallelism. *TOPC* 2, 3 (2015), 17:1–17:42.

Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. 2007. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*. 107–118.

Hans-Wolfgang Loidl and Kevin Hammond. 1995. On the granularity of divide-and-conquer parallelism. In *Proceedings of the 1995 Glasgow Workshop on Functional Programming*. 1–10.

P. Lopez, M. Hermenegildo, and S. Debray. 1996. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation* 21 (June 1996), 715–734. Issue 4-6.

Simon Marlow and Simon L. Peyton Jones. 2011. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, Hans-Juergen Boehm and David F. Bacon (Eds.). ACM, 21–32.

John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing'91*. 24–33.

MLton n.d.. MLton web site. http://www.mlton.org.

E. Mohr, D. A. Kranz, and R. H. Halstead. 1991. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (1991), 264–280.

Stefan Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive Parallelism with Futures and State. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

Stefan K. Muller and Umut A. Acar. 2016. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 71–82.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2017. Responsive Parallel Computation: Bridging Competitive and Cooperative Threading. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 677–692.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Types and Cost Models for Responsive Parallelism. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '18)*.

Stefan K. Muller, Kyle Singer, Devyn Terra Keeney, Andrew Neth, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2023. Responsive Parallelism with Synchronization. *Proc. ACM Program. Lang.* 7, PLDI (2023), 712–735.

Stefan K. Muller, Sam Westrick, and Umut A. Acar. 2019. Fairness in Responsive Parallelism. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2019)*.

Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, Rudolf Eigenmann and Martin C. Rinard (Eds.). ACM, 167–178.

Joseph Pehoushek and Joseph Weening. 1990. Low-cost process creation and dynamic partitioning in Qlisp. In *Parallel Lisp: Languages and Systems*, Takayasu Ito and Robert Halstead (Eds.). Lecture Notes in Computer Science, Vol. 441. Springer Berlin / Heidelberg, 182–199.

Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*. 383–414.

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. ACM, New York, NY, USA, 392–406.

Mike Rainey. 2023. The best multicore-parallelization refactoring you've never heard of. arXiv:2307.10556 [cs.DC]

Mike Rainey, Kyle Hale, Ryan R. Newton, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut A. Acar. 2021. Task Parallel Assembly Language for Uncompromising Parallelism. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, New York, NY, USA. http://mike-rainey.site/papers/tpal-long.pdf

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Lecture Notes in Computer Science, Vol. 6418. Springer Berlin / Heidelberg, 368–383.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 531–542.

John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the 25th ACM National Conference*. 717–740.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*.

Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3018743.3018758

Kish Shen, Vitor Santos Costa, and Andy King. 1999. Distance: A new metric for controlling granularity for parallel execution. *Journal of Functional and Logic Programming* 1999 (1999), 1–23.

K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming* FirstView (6 2014), 1–62.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 387–400.

Daniel Spoonhower. 2009. *Scheduling Deterministic Parallel Programs*. Ph.D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf

Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. 2009. Beyond Nested Parallelism: Tight Bounds on Work-stealing Overheads for Parallel Futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. ACM, New York, NY, USA, 91–100.

Andrew P. Tolmach and Dino Oliva. 1998. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *Journal of Functional Programming* 8, 4 (1998), 367–412. citeseer.nj.nec.com/tolmach93from.html

Alexandros Tzannes. 2012. *Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming*. Ph.D. Dissertation. University of Maryland.

Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. 2010. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10*. 179–190.

Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. 2014. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. *TOPLAS* 36, 3, Article 10 (Sept. 2014), 51 pages.

Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 83–94.

Stephen Weeks. 2006. Whole-program compilation in MLton. In *ML '06: Proceedings of the 2006 workshop on ML* (Portland, Oregon, USA). ACM, 1–1.

Joseph S. Weening. 1989. *Parallel Execution of Lisp Programs*. Ph.D. Dissertation. Stanford University. Computer Science Technical Report STAN-CS-89-1265.

Sam Westrick. 2022. *Efficient and Scalable Parallel Functional Programming Through Disentanglement*. Ph.D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~swestric/22/thesis.pdf

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection With Near-Zero Cost. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2022)*.

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. 2020. Parallel determinacy race detection for futures. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 217–231. https://doi.org/10.1145/3332466.3374536

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, Andrew Herbert and Kenneth P. Birman (Eds.). ACM, 221–234.

Received 2023-07-11; accepted 2023-11-07