Automatic Parallelism Management

SAM WESTRICK, Carnegie Mellon University, USA
MATTHEW FLUET, Rochester Institute of Technology, USA
MIKE RAINNEY, 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:

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

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 developed 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 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 \texttt{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., \texttt{int seq}, \texttt{string seq}, and \texttt{(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 \texttt{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 \texttt{par} into a \texttt{potentially parallel} function call—or \texttt{pcall}, for short—and represent \texttt{pcalls} explicitly as \texttt{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 \texttt{promoted} into an actual parallel task. In this way, we shift the overhead of task creation away from \texttt{par}, and instead rely on promotions (which we schedule separately) to create tasks. The programmer may therefore use \texttt{par} liberally to express all opportunities for parallelism, without harming performance.

Next, we develop a \texttt{parallelism management algorithm} to schedule promotions dynamically, during execution. Each promotion creates a task for an outstanding \texttt{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 \texttt{oldest} outstanding \texttt{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 \texttt{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 \texttt{work-efficient} and \texttt{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 \texttt{potentially parallel} function calls (\texttt{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 \texttt{pcall} has nearly zero cost.
- A semantics for an SSA intermediate language endowed with \texttt{pcall} (Section 3.3), and a technique for compiling high-level parallelism into \texttt{pcall} (Section 3.2).
- A \texttt{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 \texttt{pcalls} 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]
           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)

fun veryCheapFunction(y) = ... 
val Y = [1, 2, ..., 1000000000]
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.

par : (unit → α) × (unit → β) → α × β

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 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 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(expensiveFunction, X), but the same threshold is a bad choice for map(expensiveFunction, Y), because it would eliminate all parallelism at that callsite (note that the length of X is small).
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 parallel 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 pcalls 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 pcalls. 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.
2.1.2 Dynamic Promotions. With all potential parallelism marked in call-stacks, the scheduler is free to choose to promote an outstanding \texttt{pcall} at any moment. Selecting \texttt{pcalls} 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 \texttt{pcalls} 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 \texttt{pcall}.

3 \texttt{PCALL}: POTENTIALLY PARALLEL CALLS

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

In this section, we define the \texttt{pcall} primitive and give it a semantics. At a high level, \texttt{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 \texttt{pcall} with two behaviors, one of which is sequential and another which is parallel. In this way, \texttt{pcall} is \textit{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(x₁,…,xₙ) ▷ b, consist of: a function name f, a (possibly empty) list of arguments x₁,…,xₙ, 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(x₁,…,xₙ) ▷ (bseq,bsync,bspwn). The first return continuation, bseq, 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, bsync and bspwn, are used for the parallel behavior: bspwn points to the code that will be executed in parallel, by a spawned thread, and bsync points to the code that will synchronize with the spawned thread.

The intent is that the code path at bsync will eventually call getjoin to wait for the spawned thread to complete; similarly, the code path at bspwn will eventually call setjoin to pass a value back to the original thread. Importantly, on the sequential code path bseq, 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 ftop(…), gtop(…) ▷ 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 \( x = \ldots \) in let \( y = \ldots \) in let \( z = \ldots \) in
let \( f = \text{fn} () \Rightarrow \ldots x \ldots \) in
let \( g = \text{fn} () \Rightarrow \ldots y \ldots z \ldots \) in
let \( r = \text{par}(f, g) \) in
\[
\ldots r \ldots
\]
(a) Example source

\[
\text{fun } f_{\text{top}}(x) = \ldots x \ldots \\
\text{fun } g_{\text{top}}(y, z) = \ldots y \ldots z \ldots \\
\text{fun } \text{main}() = \\
\quad \ldots \\
\quad \text{block } \text{finish}_{\text{par}}(r_1, r_2) = \\
\quad \quad \{ r \leftarrow (r_1, r_2); \ldots \} \\
\quad \text{block } \text{after}_{\text{par}}() = \ldots r \ldots \\
\quad \text{in } \{ x \leftarrow \ldots; y \leftarrow \ldots; z \leftarrow \ldots; \}
\quad \text{par } f_{\text{top}}(x), g_{\text{top}}(y, z) \triangleright \text{finish}_{\text{par}} \}
\]
(b) Translation into SSA with \text{par} transfer, to be eliminated

c) After eliminating \text{par}, replacing with \text{pcall}, \text{getjoin}, and \text{setjoin}.

---

Fig. 4. Implementing parallelism using \text{pcall} in SSA.

\[
\text{par } f_{\text{top}}(\ldots), g_{\text{top}}(\ldots)
\]

eliminate \text{par}

\[
\text{pcall } f_{\text{top}}(\ldots)
\]

\[
\text{finish}_{\text{par}}
\]

\[
\text{after}_{\text{par}}
\]

---

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 \text{par}. The right (corresponding to Figure 4c) shows the control flow with \text{pcall}. Rectangles are basic blocks. Each block ends with a transfer, which has one or more successor blocks.

\[
\text{fun } f_{\text{top}}(x) = \ldots x \ldots \\
\text{fun } g_{\text{top}}(y, z) = \ldots y \ldots z \ldots \\
\text{fun } \text{main}() = \\
\quad \ldots \\
\quad \text{block } \text{seq}_{\text{enter}}(r_1^1) = \{ \text{call } g_{\text{top}}(y, z) \triangleright \text{seq}_{\text{exit}} \}
\quad \text{block } \text{seq}_{\text{exit}}(r_2^1) = \{ \text{goto } \text{finish}_{\text{par}}(r_1^1, r_2^1) \}
\quad \text{block } \text{sync}_{\text{enter}}(r_1^2) = \{ \text{getjoin } \triangleright \text{sync}_{\text{exit}} \}
\quad \text{block } \text{sync}_{\text{exit}}(r_2^2) = \{ \text{goto } \text{finish}_{\text{par}}(r_1^2, r_2^2) \}
\quad \text{block } \text{spwn}_{\text{enter}}() = \{ \text{call } g_{\text{top}}(y, z) \triangleright \text{spwn}_{\text{exit}} \}
\quad \text{block } \text{spwn}_{\text{exit}}(r_2^3) = \{ \text{setjoin}(r_2^3) \}
\quad \text{block } \text{finish}_{\text{par}}(r_1, r_2) = \{ r \leftarrow (r_1, r_2); \text{goto } \text{after}_{\text{par}}() \}
\quad \text{block } \text{after}_{\text{par}}() = \ldots r \ldots \\
\quad \text{in } \{ x \leftarrow \ldots; y \leftarrow \ldots; z \leftarrow \ldots; \\
\quad \text{pcall } f_{\text{top}}(x) \triangleright (\text{seq}_{\text{enter}}, \text{sync}_{\text{enter}}, \text{spwn}_{\text{enter}}) \}
\]

SSA-level functions $f_{\text{top}}$ and $g_{\text{top}}$ are the result of defunctionalizing the first-class functions $f$ and $g$. The continuation block $b$ has arity 2, expecting the results of $f_{\text{top}}$ and $g_{\text{top}}$ 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 \texttt{pcall}, as well as \texttt{getjoin} and \texttt{setjoin}. In the resulting code, we begin by calling $f_{\text{top}}(\ldots)$ using a \texttt{pcall} instead of a standard function call. There are then three possible continuations of \texttt{pcall}: seq\_enter, sync\_enter, and spwn\_enter, which are used to implement two different behaviors.

- **If the \texttt{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 \texttt{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 \texttt{pcall} by presenting an operational semantics for the language. The semantics is a small-step semantics of the form $F + \mathcal{P} \rightarrow \mathcal{P}$, where $F$ 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 $\rho$, and a join obligation $\sigma$. These components are managed by the calling convention of the language, and are used to keep track of both promoted and unpromoted \texttt{pcalls}, 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

\[\begin{align*}
\mathbb{F} & \ni \text{fun } f(\overline{x}) = \text{let } \_ = \text{in } [S; T] \\
\text{FuncLookup}(\overline{F}, f) & = ((\overline{x}), [S; T])
\end{align*}\]

\[\begin{align*}
\overline{B} & \ni \text{fun } f(\_\_\_) = \text{let } \overline{B} \in \_\_\_ \_\_ \text{ block } b(\overline{x}) = [S; T] \\
\text{BlockLookup}(\overline{F}, f, b) & = ((\overline{x}), [S; T])
\end{align*}\]

Auxiliary Definitions: Oldest Promotable Frame

\[\begin{align*}
\text{NoPromotableFrames}(\mathcal{K}) & \quad k_p = \text{frame}(\text{promotable}(\_\_\_\_\_), \_\_\_\_\_\_) \\
\text{SplitOldestPromotable}(\mathcal{K}; k_p) & = \mathcal{K}; k_p; \bullet
\end{align*}\]

\[\begin{align*}
\text{SplitOldestPromotable}(\mathcal{K}) & = \mathcal{K}_1; k_p; \mathcal{K}_2 \\
\mathcal{K}' & = \mathcal{K}_2; k
\end{align*}\]

\[\text{NoPromotableFrames}(\bullet) \quad \text{NoPromotableFrames}(\mathcal{K}; \text{frame}(\text{none}, \_\_\_\_\_\_))\]

Execution \[\overline{F} + \mathcal{P} \rightarrow \mathcal{P}\]

\[\begin{align*}
\mathcal{T} & = \text{thread}(\mathcal{K}, \sigma, X, f, \{S_1; S_2; \ldots; T\}) \\
\mathcal{X}' & = \ldots \text{apply } S_1 \text{ to } X \ldots \\
\mathcal{T}' & = \text{thread}(\mathcal{K}, \sigma, \mathcal{X}', f, \{S_2; \ldots; T\}) \quad \text{\textbf{STMT}}
\end{align*}\]

\[\begin{align*}
\mathcal{T} & = \text{thread}(\mathcal{K}, \sigma, X, f, \{\text{call } g(x_1, \ldots, x_n) \triangleright b\}) \\
\text{FuncLookup}(\overline{F}, g) & = ((y_1, \ldots, y_n), [S; T]) \\
\mathcal{Y} & = [y_1 = X(x_1), \ldots, y_n = X(x_n)] \\
\mathcal{K}' & = \mathcal{K}; \text{frame}(\_\_\_\_\_, \sigma, X, f, b) \\
\mathcal{T}' & = \text{thread}(\mathcal{K}', \text{none}, \mathcal{Y}, g, [S; T]) \quad \text{\textbf{CALL}}
\end{align*}\]

\[\begin{align*}
\mathcal{T} & = \text{thread}(\mathcal{K}, \text{none}, X, f, \{\text{pcall } g(x_1, \ldots, x_n) \triangleright (\text{bsync}, \text{bspawn})\}) \\
\text{FuncLookup}(\overline{F}, g) & = ((y_1, \ldots, y_n), [S; T]) \\
\mathcal{Y} & = [y_1 = X(x_1), \ldots, y_n = X(x_n)] \\
\mathcal{K}' & = \mathcal{K}; \text{frame}(\text{promotable}(\text{bsync}, \text{bspawn}), \text{none}, X, f, \text{bsync}) \\
\mathcal{T}' & = \text{thread}(\mathcal{K}', \text{none}, \mathcal{Y}, g, [S; T]) \quad \text{\textbf{PCALL}}
\end{align*}\]

\[\begin{align*}
\mathcal{T}_1 & = \text{thread}(\mathcal{K}, \sigma, X, f, \{S_1; T_1\}) \\
\text{SplitOldestPromotable}(\mathcal{K}) & = \mathcal{K}_1; \text{frame}(\text{promotable}(\text{bsync}, \text{bspawn}), \text{none}, \mathcal{Y}, g, \_\_\_\_\_;) \mathcal{K}_2 \\
\mathcal{K}' & = \mathcal{K}_1; \text{frame}(\text{none}, \text{child}(t_2), f, \text{bsync}) \mathcal{K}_2; T_1' & = \text{thread}(\mathcal{K}', \sigma, X, f, \{\overline{S}_2; T_1\}) \\
\text{BlockLookup}(\overline{F}, g, \text{bspawn}) & = ((\_\_\_\_, \overline{S}_2; T_2)) T_2 & = \text{thread}(\bullet, \text{none}, \mathcal{Y}, g, \{\overline{S}_2; T_2\}) \quad \text{t2 fresh} \quad \text{\textbf{PREPROMOTE}}
\end{align*}\]

\[\begin{align*}
\mathcal{T}_1 & = \text{thread}(\mathcal{K}, \text{child}(t_2), X, f, \{\text{getjoin } \triangleright b\}) \\
\text{BlockLookup}(\overline{F}, b) & = ((\_\_\_, \overline{S}; T)) \mathcal{X}' = X[x \leftarrow \mathcal{Y}(y)] T_1' & = \text{thread}(\mathcal{K}, \text{none}, \mathcal{X}', f, \{\overline{S}; T\}) \\
\mathcal{T}_1 & = \text{thread}(\bullet, \text{child}(t_2), \mathcal{X}, f, \{\text{setjoin}(y)\}) \quad \text{\textbf{JOIN}}
\end{align*}\]

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

Proc. ACM Program. Lang., Vol. 8, No. POPL, Article 38. Publication date: January 2024.
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 $\rho = \text{none} \land \sigma = \text{none}$.

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

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

When a function frame is promoted, the semantics sets $\sigma$ as well as a collection of stack slots $X$. In principle, any frame could be promoted, but we specialize language blindly returns to the label in the most recent frame, and ignores the promotion mark $\rho$.

**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

$$
\rho = \text{none} \land \sigma = \text{none}.
$$

**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 $\rho$ and join obligation $\sigma$ as described above, as well as a collection of stack slots $X$ (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 thread $(\mathcal{K}, \sigma, X, f, \{S_1; \ldots; S_n; T\})$ is currently executing the statements $\{S_1; \ldots; S_n; T\}$ inside function $f$, with current temporaries $X$, 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 `thread($\mathcal{K}, \sigma, X, f, \ldots$)` wishes to call a function, it does so by pushing `frame(none, \sigma, X, f, b)` onto its call-stack, where $b$ is the return block of the call.

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

**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 \( \text{frame}(\text{promotable}(b_{\text{sync}}, b_{\text{spwn}}), \text{none}, Y, g, \_)) \). The rule Promote spawns a new thread with identifier \( t_2 \), and then updates the selected frame to \( \text{frame}(\text{none}, \text{child}(t_2), Y, g, b_{\text{sync}}) \). Three components of the frame are updated: (1) the promotion mark is set to \text{none}, indicating that this frame is no longer promotable; (2) the join obligation is set to \text{child}(t_2), where \( t_2 \) is the identifier of the newly spawned thread of this promotion; (3) the return label is set to \( b_{\text{sync}} \), 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 \( Y \) (copied from the promoted frame), and immediately begins execution at the \( b_{\text{spwn}} \) 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 \( \sigma \) explicitly, we allow for the getjoin and setjoin primitives to synchronize with each other. In particular, in rule \text{Join}, we see that a thread labeled \( t_1 \) is waiting on a getjoin, and the join obligation of that thread is set to \text{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 \text{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 \text{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 \text{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. 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 \( \tau \) work, the total amount of work (including the cost of promotions) is \( 1 + \frac{C}{N} \tau W \).

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 \( \tau \) is the cost of a 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 \( \frac{C \tau}{N} \) should be set close to 0, where \( \tau \) 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\(^5\) (“Sugar MaPLe”). MPL\(^5\) 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\(^5\) 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/MPL₅ runtime-system functions as necessary.

In this section, we present an overview of several important aspects of the MPL₅ 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 MPL₅ and MLton are whole-program optimizing compilers, whole-program compilation is not required to implement the special behavior of pcalls.

5.1 Implementing the PCall SSA IR in MLton/MPL₅

Implementing a novel control-flow construct such as pcalls 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 pcalls 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 pcalls along with the three return continuations.

2. Middle-end changes, which explicitly represent pcalls in IRs and update analysis and optimization passes to handle pcalls.

3. Back-end changes, which implement the pcalls calling convention and emit pcalls frames.

Our claim is that the implementation effort required to support pcalls in a production compiler can be surprisingly low and our evidence is the implementation of MPL₅ 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 pcalls in MPL₅.

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 pcalls in MLton/MPL₅ 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 MPL₅’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 pcalls 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 [Češín 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
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 MPL² 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\((\text{none, none, } X, f, b)\) 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 \text{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 \text{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 provides the user with a function \( \text{par} \) of type \( (\text{unit} \rightarrow \alpha) \times (\text{unit} \rightarrow \beta) \rightarrow \alpha \times \beta \). One strategy would be to make \( \text{par} \) a compiler primitive and translate it to uses of \( \text{pcall}, \text{getjoin}, \) and \( \text{setjoin} \) primitives and/or IR constructs, as described in Section 3.2. However, \( \text{getjoin} \) and \( \text{setjoin} \) are really synchronization operations 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 \( \text{pcall} \) primitive and implement \( \text{par} \) using \( \text{pcall} \) in source SML code. The \( \text{pcall} \) primitive has the type \( (\alpha \rightarrow \beta) \times \alpha \times (\beta \rightarrow \gamma) \times (\beta \rightarrow \gamma) \times (\text{unit} \rightarrow \delta) \rightarrow \gamma \), where \( \text{pcall}(f, x, \text{seq}, \text{sync}, \text{spwn}) \) roughly corresponds to the \( \text{pcall} f(x) \triangleright (b_{\text{seq}}, b_{\text{sync}}, b_{\text{spwn}}) \) transfer from Section 3, except that the \( \text{pcall} \) primitive returns the result of either \( \text{seq} \) or \( \text{sync} \) (implicitly joining their control-flow), while the \( \text{pcall} \) transfer allows the control-flow of \( b_{\text{seq}} \) and \( b_{\text{sync}} \) to be arbitrary. The only assumption that the compiler makes about the \( \text{pcall} \) primitive is that the \( \text{spwn} \) argument function does not terminate with a value and instead exits the executing thread (as is the semantics of \( \text{setjoin} \) in Section 3).

Using \( \text{pcall} \), we can give a simplified implementation of \( \text{par} \).

```sml
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

val _ = Signal.setHandler (Signal.SIGALRM, handler)
```

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

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 \( \text{spawnOldestPCallAndSetData}(t, d) \), 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 \( \text{join} \) value.

Note that while \( \text{getPCallData} \) and \( \text{spawnOldestPCallAndSetData} \) 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 \( \text{spawnOldestPCallAndSetData} \) flow out of \( \text{getPCallData} \). Moreover, \( \text{getPCallData} \) can be implemented efficiently in the compiler when frames and the call stack are made explicit.

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

---

1Essentially, \( \text{Id} \)-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 bseq, bsync, and bspwn as the continuations. The new top-level function takes f and x and performs the defunctionalized call of f x. Both bseq and bsync 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 sync rf and then join at the block receiving the result of the pcall primitive. bspwn 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 pcalled
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_sync, b_spwn), none, X, f, b_seq)`. The frame info corresponding to `RetAddr_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 `RetAddr_seq` records the set of live pointers as the `union` of the variables that are live at the beginning of `RetAddr_seq`, `RetAddr_sync`, and `RetAddr_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 `RetAddr_spwn/b_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 `RetAddr_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 `RetAddr_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 `RetAddr_sync` and `RetAddr_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 `RetAddr_sync` and `RetAddr_spwn` include the `DataPtr` slot among the set of live pointers along with the variables that are live at the beginning of `RetAddr_sync` and `RetAddr_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 `pcalled`, increments the stack-top pointer to the top of its frame. Moreover, returning from a `pcall` to either `RetAddr_seq` or `RetAddr_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 `RetAddr_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 `pcalled`. 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 `RetAddr_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 500\( \mu \)s, 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\textsuperscript{s}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 \texttt{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 \texttt{par}. This cost, although small, would nevertheless be prohibitively expensive if it were to happen at every \texttt{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\textsuperscript{s} (“Sugar MaPLe”). For our evaluation, we consider a benchmark suite consisting of both “fully parallel” benchmarks (Section 6.1) which use \texttt{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 \texttt{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\textsuperscript{s} against MPL [Acar et al. 2020], the compiler on which MPL\textsuperscript{s} is based. MPL features an eager implementation of \texttt{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\textsuperscript{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\textsuperscript{s}’s guaranteed amortization of the cost of \texttt{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\textsuperscript{s} is able to simultaneously amortize the overhead of \texttt{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\textsuperscript{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 (\texttt{map}, \texttt{scan}, \texttt{reduce}, \texttt{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.

\textit{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 \texttt{par}, thus amortizing its overhead, and (2) algorithmic grains, whose purpose is
Table 1. Times (seconds), performance improvements of MPL\textsuperscript{5} relative to MPL, and analysis of number of pars and promotions, for the fully parallel benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$T_i$</th>
<th>$T_{64}$</th>
<th>$\frac{MPL}{MPL^5}$</th>
<th>$\frac{MPL}{MPL^5}$</th>
<th>$\frac{MPL}{MPL^5}$</th>
<th>#par</th>
<th>promote</th>
<th>#par</th>
<th>promote</th>
<th>work($\mu$s) promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfs</td>
<td>7.47</td>
<td>4.32</td>
<td>1.73</td>
<td>.281 .115</td>
<td>.244</td>
<td>36 M</td>
<td>76 K</td>
<td>473</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>bignum-add</td>
<td>4.39</td>
<td>1.48</td>
<td>2.97</td>
<td>.218 .028</td>
<td>.779</td>
<td>32 M</td>
<td>17 K</td>
<td>1833</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>delaunay</td>
<td>7.38</td>
<td>7.56</td>
<td>0.98</td>
<td>.275 .290</td>
<td>.95</td>
<td>2.0</td>
<td>M .18 M</td>
<td>11.2</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>grep</td>
<td>9.55</td>
<td>3.87</td>
<td>2.47</td>
<td>.466 .076</td>
<td>5.34</td>
<td>60 M</td>
<td>45 K</td>
<td>1320</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>linefit</td>
<td>10.7</td>
<td>2.58</td>
<td>4.15</td>
<td>.518 .066</td>
<td>7.85</td>
<td>67 M</td>
<td>36 K</td>
<td>1890</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>mandelbrot</td>
<td>4.52</td>
<td>2.87</td>
<td>1.57</td>
<td>.163 .049</td>
<td>3.33</td>
<td>17 M</td>
<td>29 K</td>
<td>603</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>map-heavy</td>
<td>3.63</td>
<td>3.69</td>
<td>0.98</td>
<td>.058 .060</td>
<td>.97</td>
<td>3.9 K</td>
<td>3.0 K</td>
<td>1.26</td>
<td>1210</td>
<td></td>
</tr>
<tr>
<td>map-light</td>
<td>17.3</td>
<td>3.93</td>
<td>4.40</td>
<td>.773 .078</td>
<td>9.91</td>
<td>.13 B</td>
<td>45 K</td>
<td>2996</td>
<td>88</td>
<td></td>
</tr>
<tr>
<td>msort</td>
<td>7.14</td>
<td>4.99</td>
<td>1.43</td>
<td>.222 .093</td>
<td>2.39</td>
<td>19 M</td>
<td>58 K</td>
<td>333</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>nearest-nbns</td>
<td>1.39</td>
<td>1.25</td>
<td>1.11</td>
<td>.031 .026</td>
<td>1.19</td>
<td>1.5 M</td>
<td>18 K</td>
<td>83.5</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>nqueens</td>
<td>8.23</td>
<td>3.07</td>
<td>2.68</td>
<td>.431 .053</td>
<td>8.13</td>
<td>55 M</td>
<td>31 K</td>
<td>1791</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>primes</td>
<td>21.1</td>
<td>5.75</td>
<td>3.67</td>
<td>.942 .124</td>
<td>7.60</td>
<td>14 B</td>
<td>73 K</td>
<td>1970</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>sparse-mxv-csr</td>
<td>15.5</td>
<td>4.32</td>
<td>3.59</td>
<td>.547 .088</td>
<td>6.22</td>
<td>78 M</td>
<td>48 K</td>
<td>1633</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>suffix-array</td>
<td>6.58</td>
<td>3.79</td>
<td>1.74</td>
<td>.318 .084</td>
<td>3.79</td>
<td>29 M</td>
<td>36 K</td>
<td>792</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>triangle-count</td>
<td>15.7</td>
<td>9.20</td>
<td>1.71</td>
<td>.714 .168</td>
<td>4.25</td>
<td>54 M</td>
<td>95 K</td>
<td>566</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>wc</td>
<td>7.80</td>
<td>1.84</td>
<td>4.24</td>
<td>.464 .032</td>
<td>14.50</td>
<td>54 M</td>
<td>19 K</td>
<td>2901</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

| 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 many 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 MPL/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 \#par/#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 pars 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 $16\mu s$ 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
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 \texttt{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 (\texttt{delaunay}). In the case of \texttt{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 \texttt{par} occurs, with no delay.

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

We compare against MLton, on which MPL\textsuperscript{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 \texttt{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\textsuperscript{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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>(P = 1)</th>
<th>(P = 64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfs</td>
<td>0.94</td>
<td>1.51</td>
</tr>
<tr>
<td>bignum-add</td>
<td>1.00</td>
<td>1.25</td>
</tr>
<tr>
<td>delaunay</td>
<td>1.01</td>
<td>3.23</td>
</tr>
<tr>
<td>grep</td>
<td>1.00</td>
<td>1.14</td>
</tr>
<tr>
<td>linefit</td>
<td>0.98</td>
<td>1.05</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>0.99</td>
<td>1.02</td>
</tr>
<tr>
<td>map-heavy</td>
<td>1.00</td>
<td>1.03</td>
</tr>
<tr>
<td>map-light</td>
<td>0.97</td>
<td>1.03</td>
</tr>
<tr>
<td>msort</td>
<td>1.06</td>
<td>1.17</td>
</tr>
<tr>
<td>nearest-nbrs</td>
<td>1.00</td>
<td>1.77</td>
</tr>
<tr>
<td>nqueens</td>
<td>1.02</td>
<td>1.08</td>
</tr>
<tr>
<td>primes</td>
<td>0.97</td>
<td>1.12</td>
</tr>
<tr>
<td>sparse-mxv-csr</td>
<td>1.41</td>
<td>1.10</td>
</tr>
<tr>
<td>suffix-array</td>
<td>0.97</td>
<td>1.44</td>
</tr>
<tr>
<td>triangle-count</td>
<td>0.98</td>
<td>1.00</td>
</tr>
<tr>
<td>wc</td>
<td>0.96</td>
<td>1.03</td>
</tr>
</tbody>
</table>

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 \(MPL^3\). Smaller overheads are preferable. (See Table 1 for the reported fully-parallel timings.)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>(T_1) (manual)</th>
<th>(T_1) (fully-par)</th>
<th>(T_8) (manual)</th>
<th>(T_8) (fully-par)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfs</td>
<td>3.03</td>
<td>3.15</td>
<td>2.47</td>
<td>1.37</td>
</tr>
<tr>
<td>bignum-add</td>
<td>3.31</td>
<td>3.20</td>
<td>1.33</td>
<td>0.46</td>
</tr>
<tr>
<td>delaunay</td>
<td>7.07</td>
<td>7.33</td>
<td>1.04</td>
<td>1.03</td>
</tr>
<tr>
<td>grep</td>
<td>2.00</td>
<td>2.02</td>
<td>4.78</td>
<td>1.92</td>
</tr>
<tr>
<td>linefit</td>
<td>0.35</td>
<td>0.45</td>
<td>30.40</td>
<td>5.67</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>1.62</td>
<td>1.63</td>
<td>2.79</td>
<td>1.76</td>
</tr>
<tr>
<td>map-heavy</td>
<td>3.62</td>
<td>3.66</td>
<td>1.00</td>
<td>1.01</td>
</tr>
<tr>
<td>map-light</td>
<td>0.92</td>
<td>0.96</td>
<td>18.80</td>
<td>3.95</td>
</tr>
<tr>
<td>msort</td>
<td>3.84</td>
<td>4.26</td>
<td>1.86</td>
<td>1.17</td>
</tr>
<tr>
<td>nearest-nbrs</td>
<td>1.15</td>
<td>1.17</td>
<td>1.21</td>
<td>1.07</td>
</tr>
<tr>
<td>nqueens</td>
<td>1.40</td>
<td>1.38</td>
<td>5.88</td>
<td>2.22</td>
</tr>
<tr>
<td>primes</td>
<td>1.82</td>
<td>1.86</td>
<td>11.59</td>
<td>3.09</td>
</tr>
<tr>
<td>sparse-mxv-csr</td>
<td>1.71</td>
<td>1.85</td>
<td>9.06</td>
<td>2.34</td>
</tr>
<tr>
<td>suffix-array</td>
<td>4.55</td>
<td>4.65</td>
<td>1.45</td>
<td>0.82</td>
</tr>
<tr>
<td>triangle-count</td>
<td>3.90</td>
<td>3.96</td>
<td>4.03</td>
<td>2.32</td>
</tr>
<tr>
<td>wc</td>
<td>.608</td>
<td>.620</td>
<td>12.83</td>
<td>2.97</td>
</tr>
</tbody>
</table>

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 \(MPL^3\) on 1 and 64 processors. The column labeled “Overhead” is the ratio \(MPL^3\) single-core time to \(MLton\)’s time; this ratio summarizes the overhead of parallelism, including the amortization of \(\text{par}\) as well as parallel memory management overheads. The column labeled “Speedup” is the ratio of \(MLton\)’s time to \(MPL^3\)’s time on 64 cores; this ratio is the speedup of \(MPL^3\) 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 \(MPL^3\) 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 \(MPL^3\) 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 \(MPL^3\) 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 \(MPL^3\) preserves the scalability of these benchmarks. We conclude that \(MPL^3\) 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 \(MPL^3\) 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 MPL³.

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 MPL³. 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, MPL³ 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 MPL³ 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 MPL³ 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 pars 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. Furthermore, 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 \texttt{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 challenge 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, 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).


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